ALIBABA CLOUD

阿里云

云原生多模数据库Lindorm

云原生多模数据库Lindorm（合集）

文档版本：20220713

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.有奖问卷调研 24
2.动态与公告 25
2.1. 产品公告 25
2.1.1. Lindorm容量型云存储价格变动 25
2.2. 版本说明 25
2.2.1. 宽表引擎版本说明 25
2.2.2. 时序引擎版本说明 27
2.2.3. 文件引擎版本说明 31
2.2.4. 搜索引擎版本说明 32
2.3. 安全公告 33
2.3.1. 【通知】Apache Log4j2远程代码执行漏洞（CVE-2021-44228.. 33
3.产品简介 34
3.1. 什么是云原生多模数据库Lindorm 34
3.2. Lindorm产品选型 34
3.3. 产品系列 35
3.4. 功能特性 36
3.5. 应用场景 37
3.6. 产品架构 43
3.7. 产品优势 47
3.7.1. Lindorm VS HBase VS Cassandra 47
3.7.2. Lindorm VS OpenTSDB 49
3.7.3. Lindorm VS ElasticSearch VS Solr 50
3.7.4. Lindorm VS HDFS 51
3.8. 常见问题 52
3.8.1. Lindorm连接问题与解决方法 52
3.8.2. 使用ping命令检测ECS与Lindorm之间的连接 52

3.8.3. 使用telnet命令检测Lindorm端口连通性 53
3.8.4. Lindorm连接出现报错与解决方法 54
3.9. 技术支持 54
3.10. 引擎类型 55
3.11. 存储类型 55
3.12. 基本概念 56
4.产品计费 61
4.1. 计费方式 61
4.2. 产品价格 61
4.3. 到期和欠费说明 63
4.4. 续费说明 64
4.5. 变更配置计费说明 64
4.6. 退费说明 65
5.快速入门 66
5.1. 快速入门概览 66
5.2. 创建实例 66
5.3. 设置白名单 69
5.4. 连接并使用实例 70
6.用户指南 71
6.1. 实例管理 71
6.1.1. 创建实例 71
6.1.2. 变更实例配置 73
6.1.2.1. 变更实例规格 73
6.1.2.2. 变更实例节点 74
6.1.3. 升级小版本 75
6.1.4. 开启或关闭删除保护 75
6.1.5. 释放实例 76
6.1.6. 设置可维护时间段 76

6.1.7. 查看连接地址 76
6.1.8. 创建多可用区实例 81
6.2. 监控与报警 83
6.2.1. 查看监控信息 83
6.2.2. 报警规则 83
6.2.2.1. 创建报警规则 83
6.2.2.2. 管理报警 85
6.3. 标签管理 87
6.3.1. 新建标签 87
6.3.2. 绑定已有标签 87
6.3.3. 根据标签筛选实例 87
6.3.4. 解绑或删除标签 88
6.4. 添加白名单和安全组 88
6.4.1. 设置白名单 88
6.4.2. 添加安全组 89
6.5. Lindorm服务关联角色 89
6.6. 计费管理 91
6.6.1. 按量付费转包年包月 91
6.6.2. 开通自动续费 91
6.6.3. 手动续费包年包月实例 93
6.7. 开通冷存储 93
7.宽表引擎（兼容HBase或Cassandra） 95
7.1. 引擎简介 95
7.2. 通过Lindorm宽表SQL连接宽表引擎 95
7.2.1. Lindorm宽表SQL Driver说明 95
7.2.2. 访问实例 95
7.2.3. 通过Lindorm宽表SQL Java API连接并使用宽表引擎 96
7.2.4. 通过Lindorm宽表SQL 非Java API连接并使用宽表引擎 98

7.2.5. 通过Lindorm-cli连接并使用宽表引擎 99
7.2.6. 通过Java框架访问Lindorm宽表引擎示例 101
7.3. 使用开源客户端访问Lindorm宽表引擎 101
7.3.1. 通过HBase API连接并使用宽表引擎 101
7.3.1.1. HBase SDK 安装 101
7.3.1.2. 通过HBase Java API连接并使用宽表引擎 101
7.3.1.3. 通过HBase非Java API连接并使用宽表引擎 103
7.3.1.4. 通过HBaseue Shell连接并使用宽表引擎 104
7.3.1.5. 使用限制 106
7.3.2. 通过Cassandra CQL连接并使用宽表引擎 106
7.3.2.1. Cassandra CQL Driver 安装 106
7.3.2.2. 连接集群 106
7.3.2.3. 通过Lindorm-cqlsh连接并使用宽表引擎 106
7.3.2.4. 通过Cassandra CQL Java Driver连接并使用宽表引擎 110
7.3.2.5. 通过Cassandra CQL多语言Driver连接并使用宽表引擎 111
7.3.2.6. 使用限制 116
7.3.2.7. Lindorm CQL语法介绍 117
7.3.2.7.1. 名词解释 117
7.3.2.7.2. 数据类型 117
7.3.2.7.3. DESCRIBE 119
7.3.2.7.4. DDL 121
7.3.2.7.5. DML 129
7.3.2.7.6. DCL 134
7.3.2.7.7. 函数说明 139
7.3.2.7.8. JSON支持 142
7.3.3. 通过Phoenix SQL连接并使用宽表引擎（已下线） 143
7.3.3.1. PhoenixSQL SDK安装 143
7.3.3.2. 开通宽表SQL 143

7.3.3.3. 连接集群 146
7.3.3.4. 通过Phoenix SQL Java API连接并使用宽表引擎 146
7.3.3.5. 通过Phoenix SQL非Java API连接并使用宽表引擎 147
7.3.3.6. 通过Phoenix Sqlline连接并使用宽表引擎 147
7.3.3.7. 使用限制 148
7.3.3.8. PhoenixSQL与Spark的选择 148
7.3.4. 通过S3协议连接并使用宽表引擎 148
7.3.4.1. S3兼容协议使用文档 148
7.3.4.2. 开通S3协议兼容功能 148
7.3.4.3. 获取连接地址 149
7.3.4.4. 通过S3 Java API连接并使用宽表引擎 149
7.3.4.5. 通过S3非Java API连接并使用宽表引擎 153
7.3.4.6. 使用限制以及兼容API 156
7.4. 企业特性 157
7.4.1. 冷存储 157
7.4.1.1. 冷存储介绍 157
7.4.1.2. 配置冷存储 158
7.4.1.3. 查看冷存储容量 159
7.4.1.4. 扩容冷存储 160
7.4.2. 冷热分离 160
7.4.3. 管理动态列 165
7.4.4. 高性能原生二级索引 167
7.4.5. 搜索索引 171
7.4.5.1. 特性介绍 171
7.4.5.2. 开通搜索索引 172
7.4.5.3. 使用手册 173
7.4.5.3.1. 管理搜索索引 174
7.4.5.3.2. 分区索引 176

7.4.5.3.3. 通过搜索索引查询宽表数据 178
7.4.5.3.4. 高级查询语法 180
7.4.5.4. 使用限制 181
7.4.5.5. 开源客户端访问（兼容云数据库HBase增强版） 182
7.4.5.5.1. 通过开源客户端访问并使用搜索索引 182
7.4.5.5.2. 列映射的配置项说明 184
7.4.5.5.3. 管理列映射 185
7.4.5.5.4. 通过构建索引完成全量数据同步 187
7.4.6. 宽表Serverless 188
7.4.7. 备份恢复 190
7.4.7.1. 开通备份恢复 190
7.4.7.2. 自动备份Lindorm数据 191
7.4.7.3. 恢复备份数据至当前实例 192
7.4.8. 开启透明数据加密（内测中） 193
7.4.9. 多可用区部署 194
7.4.10. 审计日志 197
7.4.10.1. 开通审计日志 197
7.4.10.2. 查询审计日志 197
7.5. 数据生态 199
7.5.1. 数据导入与迁移 199
7.5.2. 使用DataWorks或者DataX导入数据 200
7.5.3. 通过Spark访问Lindorm 202
7.5.4. 通过Hive访问Lindorm 205
7.5.5. 通过Flink访问Lindorm 207
7.5.6. 通过Grafana访问Lindorm 208
7.5.7. 通过Redash访问Lindorm 210
7.5.8. 通过DataWorks将MongoDB数据迁移至Lindorm宽表 212
7.6. 性能白皮书 215

7.6.1. 测试环境 216
7.6.2. 性能测试工具-Shell 216
7.6.3. 性能测试工具-图形工具 219
7.6.4. 测试方法 223
7.6.5. 测试结果分析 228
7.7. 集群管理系统 229
7.7.1. 登录集群管理系统 229
7.7.2. 分组管理 230
7.7.3. Namespace管理 233
7.7.4. 表管理 233
7.7.5. 用户和权限管理 234
7.7.5.1. 管理用户 234
7.7.5.2. 管理ACL权限 234
7.7.6. 数据查询 236
7.8. 开发指南 237
7.8.1. Lindorm宽表SQL语法手册 237
7.8.1.1. 名词解释 237
7.8.1.2. 数据类型 238
7.8.1.3. 函数说明 239
7.8.1.3.1. Count函数 239
7.8.1.3.2. Sum函数 239
7.8.1.3.3. Avg函数 239
7.8.1.3.4. Max函数和Min函数 239
7.8.1.3.5. As函数 239
7.8.1.4. 使用限制说明 239
7.8.1.5. DDL 240
7.8.1.5.1. 概览 240
7.8.1.5.2. DESCRIBE/SHOW/USE 241

7.8.1.5.3. CREATE DATABASE 241
7.8.1.5.4. DROP DATABASE 241
7.8.1.5.5. CREATE TABLE 242
7.8.1.5.6. ALTER TABLE 243
7.8.1.5.7. TRUNCATE TABLE 244
7.8.1.5.8. DROP TABLE 244
7.8.1.5.9. CREATE INDEX 244
7.8.1.5.10. ALTER INDEX 245
7.8.1.5.11. BUILD INDEX 246
7.8.1.5.12. SHOW INDEX 246
7.8.1.5.13. DROP INDEX 246
7.8.1.5.14. OFFLINE TABLE 247
7.8.1.5.15. ONLINE TABLE 247
7.8.1.5.16. CREATE SEARCH INDEX 247
7.8.1.5.17. DROP SEARCH INDEX 249
7.8.1.5.18. ALTER SEARCH INDEX 249
7.8.1.5.19. SHOW SEARCH INDEX 252
7.8.1.6. DML 252
7.8.1.6.1. SELECT 252
7.8.1.6.2. UPSERT 253
7.8.1.6.3. DELETE 254
7.8.1.7. DCL 255
7.8.1.7.1. 概述 255
7.8.1.7.2. SHOW USERS 255
7.8.1.7.3. CREATE USER 255
7.8.1.7.4. ALTER USER 256
7.8.1.7.5. DROP USER 256
7.8.1.7.6. GRANT PERMISSION 256

7.8.1.7.7. REVOKE PERMISSION 257
7.8.1.8. EXPLAIN 258
7.8.2. Lindorm SQL ORDER BY介绍 258
7.8.3. 主键设计的问题和示例 259
7.8.4. 统计表行数的方法 260
8.时序引擎 264
8.1. 引擎简介 264
8.2. 产品功能 264
8.3. 数据模型 265
8.4. 与旧版TSDB的比较 268
8.5. 快速入门 268
8.5.1. 入门概览 268
8.5.2. 通过Lindorm-cli连接访问Lindorm时序引擎 269
8.5.3. 通过JDBC Driver连接并访问Lindorm时序引擎 273
8.5.4. 通过Java Native SDK连接并使用Lindorm时序引擎 275
8.6. 数据管理 279
8.6.1. 数据库管理 279
8.6.2. 用户及权限管理 281
8.6.3. 冷数据归档 285
8.6.4. 数据的持续查询 287
8.7. 开发指南 289
8.7.1. 时序引擎应用开发简介 289
8.7.2. 关于时序数据的Schema约束 290
8.7.3. 通过JDBC Driver连接并访问时序引擎 291
8.7.4. SQL接口应用指南 297
8.7.5. Java Native SDK 299
8.7.5.1. 快速入门 299
8.7.5.2. 客户端初始化 303

8.7.5.3. 接口说明 305
8.7.5.4. 异常处理 310
8.7.5.5. 版本说明 311
8.7.6. SQL语法参考 312
8.7.6.1. 数据类型 312
8.7.6.2. 常用时序函数 312
8.7.6.3. DDL 321
8.7.6.3.1. CREATE DATABASE 321
8.7.6.3.2. CREATE TABLE 322
8.7.6.3.3. CREATE USER 323
8.7.6.3.4. CREATE CONTINUOUS QUERY 323
8.7.6.3.5. ALTER DATABASE 324
8.7.6.3.6. ALTER TABLE 325
8.7.6.3.7. ALTER SYSTEM 326
8.7.6.3.8. ALTER USER 327
8.7.6.3.9. DROP DATABASE 327
8.7.6.3.10. DROP USER 328
8.7.6.3.11. DROP CONTINUOUS QUERY 328
8.7.6.3.12. DESCRIBE DATABASE 328
8.7.6.3.13. DESCRIBE TABLE 329
8.7.6.3.14. SHOW 329
8.7.6.3.15. SHOW PARAMETER 332
8.7.6.3.16. SHOW CONTINUOUS QUERIES 332
8.7.6.3.17. USE 333
8.7.6.4. DML 333
8.7.6.4.1. 数据写入 333
8.7.6.4.2. 基本查询 335
8.7.6.4.3. 降采样查询 336

8.7.6.4.4. 最新值查询 340
8.7.6.5. DCL 342
8.7.6.5.1. GRANT 342
8.7.6.5.2. REVOKE 343
8.7.7. 问题排查 344
8.7.7.1. 常见错误码参考 344
8.8. 数据生态 345
8.8.1. 使用Telegraf写入数据到Lindorm 345
8.8.2. 通过Grafana访问Lindorm时序引擎 348
8.8.3. InfluxDB®行协议写入 353
8.8.4. 对接Prometheus 354
8.8.5. 使用Flink写入时序引擎 355
8.9. 最佳实践 358
8.9.1. 如何设计时序数据表 358
8.9.2. 如何高效地写入数据 360
8.9.3. 如何高效地查询时序数据 361
8.10. 性能白皮书 364
8.10.1. 测试环境与工具 364
8.10.2. 写入测试结果 365
8.10.3. 查询测试结果 372
9.搜索引擎 377
9.1. 引擎简介 377
9.2. 连接并使用搜索引擎 377
9.2.1. 通过Search Shell连接并使用搜索引擎 377
9.2.2. 通过SQL连接并使用搜索引擎 379
9.3. 用户指南 381
9.3.1. 开通指南 381
9.3.2. 查看连接地址 381

9.3.3. 使用示例 382
9.3.4. 集群容量规划 383
9.3.5. 配置冷存储 384
9.4. 集群管理 384
9.4.1. 登录集群管理系统 384
9.4.2. 管理索引表 385
9.5. 性能白皮书 387
9.5.1. 测试环境 387
9.5.2. 测试结果 388
9.6. 开发指南 389
9.6.1. 更新配置集 389
9.6.2. 索引数据可见性 390
9.6.3. 分库分表（Alias功能） 392
9.6.4. 预排序 396
9.6.5. Lindorm搜索SQL语法手册 396
9.6.5.1. 名词解释 396
9.6.5.2. 数据类型 396
9.6.5.3. 函数说明 397
9.6.5.3.1. Count函数 397
9.6.5.3.2. Sum函数 397
9.6.5.3.3. Avg函数 397
9.6.5.3.4. Max函数和Min函数 397
9.6.5.3.5. As函数 397
9.6.5.4. 使用限制说明 398
9.6.5.5. DDL 398
9.6.5.5.1. DESCRIBE/SHOW 398
9.6.5.5.2. CREATE TABLE 398
9.6.5.5.3. DROP TABLE 401

9.6.5.6. DML 401
9.6.5.6.1. SELECT 401
9.6.5.6.2. UPSERT 402
9.6.5.6.3. DELETE 403
9.7. 数据生态 403
9.7.1. 连接Lindorm Search和Grafana 403
10.文件引擎（兼容HDFS） 405
10.1. 引擎简介 405
10.2. 连接并使用文件引擎 405
10.2.1. 通过HDFS Shell连接并使用文件引擎 405
10.2.2. 通过开源HDFS客户端连接并使用文件引擎 406
10.2.3. 通过HDFS FUSE连接并使用文件引擎 408
10.3. 用户指南 409
10.3.1. 如何选择文件引擎规格 409
10.3.2. 开通指南 409
10.3.3. 数据存储目录结构说明 410
10.3.4. 设置数据冷热存储策略 411
10.3.5. 透明压缩 411
10.4. 开发指南 413
10.4.1. 使用Spark访问 413
10.4.2. 使用Hive访问 417
10.4.3. 使用Flink访问 421
10.4.4. 使用Presto访问 425
10.4.5. 使用DLA访问 429
10.4.6. 使用E-Mapreduce访问 431
10.4.7. 使用MaxCompute访问 438
10.5. 最佳实践 446
10.5.1. Ambari与文件引擎集成 446

10.5.2. CDH6与文件引擎集成 456
11.计算引擎 473
11.1. 引擎简介 473
11.2. 开通与变配 473
11.3. 快速入门 474
11.4. 开发指南 475
11.4.1. 使用SQL访问Lindorm计算引擎 475
11.4.2. JAR作业开发实践 477
11.4.3. Python作业开发实践 480
11.4.4. 流式作业开发实践 482
11.4.5. 作业配置说明 485
11.4.6. 本地运行并调试Spark作业 486
11.5. 数据访问 488
11.5.1. 使用须知 488
11.5.2. 使用Hive Metastore管理计算引擎元信息 489
11.5.3. 访问Lindorm宽表引擎数据的介绍 491
11.5.4. 访问Lindorm CDC数据的介绍 494
11.6. 作业调度运维 496
11.6.1. 通过Lindorm控制台管理作业 496
11.6.2. 通过DMS调度Lindorm Spark任务 497
11.6.3. SparkUI界面介绍 504
11.6.4. 查看计算引擎作业运行日志 504
12.流引擎 507
12.1. 引擎简介 507
12.2. 快速入门 508
12.2.1. 入门概览 508
12.2.2. 开通流引擎 508
12.2.3. 查看连接地址 509

12.2.4. 通过Lindorm-cli连接并使用Lindorm流引擎 510
12.2.5. 通过开源Kafka客户端写入Lindorm流引擎数据 512
12.3. 流引擎SQL语法介绍 514
12.3.1. 数据类型 514
12.3.2. DDL 515
12.3.2.1. CREATE STREAM 515
12.3.2.2. CREATE EXTERNAL TABLE 516
12.3.2.3. DROP STREAM 516
12.3.2.4. DROP EXTERNAL TABLE 517
12.3.2.5. SHOW STREAMS 517
12.3.2.6. SHOW EXTERNAL TABLES 517
12.3.2.7. DESCRIBE 518
12.3.3. DQL 518
12.3.3.1. SELECT 518
12.3.4. DML 521
12.3.4.1. CREATE CQ 521
12.3.4.2. LIST CQ 521
12.3.4.3. TERMINATE CQ 522
12.4. 最佳实践 522
12.4.1. 对流引擎数据进行实时统计 522
13.Ganos时空引擎 526
13.1. 引擎简介 529
13.2. 开通指南 531
13.3. 时空几何 531
13.3.1. 基本概念 531
13.3.2. 快速入门 532
13.3.3. 创建索引表 533
13.3.4. 创建时空对象 535

13.3.5. 写入时空对象 537
13.3.6. 查询时空对象 537
13.3.7. 删除时空对象 540
13.3.8. 与Spark集成分析 541
13.3.9. 与GeoServer集成 541
13.4. 时空栅格 541
13.5. 时空轨迹 542
13.5.1. 功能简介 542
13.5.2. 基本概念 542
13.5.3. 使用指南 543
13.5.4. 代码示例 545
14.数据导入导出 547
14.1. Lindorm数据生态大图 547
14.2. LTS（原BDS）服务介绍 548
14.3. 开通并登录LTS 549
14.4. 数据源管理 550
14.4.1. 添加DTS数据源 550
14.4.2. 添加HBase数据源 551
14.4.3. ODPS数据源 553
14.4.4. RocketMQ数据源 555
14.4.5. 添加Kafka数据源 556
14.4.6. 添加Lindorm宽表数据源 557
14.4.7. RDS数据源添加 558
14.4.8. 添加HDFS数据源 559
14.5. 实时数据订阅 559
14.5.1. 实时数据订阅概述 559
14.5.2. 通过Pull模式创建数据订阅通道 560
14.5.3. 使用Flink SQL消费实时变更数据示例 563

14.5.4. 通过Push模式创建数据订阅 566
14.5.5. 数据消费格式 569
14.6. 宽表引擎数据迁移 573
14.6.1. 不停机迁移 573
14.6.1.1. 全增量数据同步 573
14.6.1.2. 网络打通 575
14.6.1.3. 公网迁移专项 576
14.6.1.4. Phoenix全增量迁移 577
14.6.2. 数据导入 580
14.6.2.1. RDS全增量同步 580
14.6.2.2. RDS全量导入 585
14.6.2.3. RDS增量导入 588
14.6.2.4. 导入日志服务增量数据 592
14.6.2.5. 通用批量导入服务 594
14.6.2.6. 将Cassandra数据导入至Lindorm宽表 600
14.6.3. 数据导出 602
14.6.3.1. 全量导出MaxCompute 602
14.6.3.2. 增量归档MaxCompute 604
14.6.4. 最佳实践 606
14.6.4.1. 低成本历史库 606
14.7. 时序引擎数据迁移 609
14.7.1. 利用DataWorks将数据迁移到Lindorm时序引擎 609
14.7.2. 从关系型数据库到Lindorm时序引擎的数据迁移 618
14.7.3. 从 Prometheus 到 Lindorm时序引擎的数据迁移 622
14.8. 搜索引擎数据迁移 625
14.8.1. 增量导入 626
14.8.2. 全量导入 626
14.9. 文件引擎数据迁移 626

14.9.1. 从自建HDFS迁移数据 626
14.9.2. 从OSS迁移数据 627
15.SDK参考 630
15.1. OpenAPI-SDK下载 630
15.2. Java SDK for Lindorm OpenAPI使用说明 630
15.3. 宽表引擎 632
15.3.1. HBase API 632
15.3.1.1. 升级HBase Java SDK 632
15.3.1.2. 通过HBase非Java SDK访问Lindorm宽表引擎 633
15.3.1.3. 通过Maven方式升级HBase客户端 634
15.3.1.4. 通过导入JAR文件升级HBase客户端 638
15.3.2. Cassandra CQL Driver 639
15.3.2.1. Cassandra CQL Java Driver 安装和升级 639
15.4. 时序引擎 639
15.4.1. 通过JDBC Driver连接并访问时序引擎 639
15.4.2. 通过Native SDK连接并访问时序引擎 645
15.5. 搜索引擎 649
15.5.1. 通过开源Solr API访问搜索引擎 649
16.行业案例 651
16.1. 车联网 651
16.1.1. 上海新能源汽车车辆基础数据 651
16.1.2. 某网约车公司车辆轨迹数据 652
16.1.3. 某新能源汽车造车新势力 653
16.2. 航运物流 654
16.2.1. 申通快递迁移Oracle到Lindorm 654
16.2.2. 亿海蓝-航运大数据的可视化案例 655
16.3. 医疗健康 657
16.3.1. 某国内领先的健康体检公司迁移SQL Server到Lindorm 657

16.4. 广告行业 658
16.4.1. 某国际智能营销服务公司广告投放数据迁移至阿里云Lindorm 658
16.4.2. 某互动广告运营商实时竞价数据迁移至阿里云Lindorm 660
16.4.3. 某全球领先媒介投资管理公司迁移Bigtable到Lindorm 662
16.5. 金融行业 663
16.5.1. 收钱吧订单查询&搜索 663
16.5.2. 云数据库Lindorm助力数禾科技业务系统实现真正的流批一体 665
16.5.3. 某互联网保险公司保险订单迁移MySQL到Lindorm 666
16.5.4. 云数据库Lindorm助力蚂蚁链IoT数据降本 667
16.6. 游戏行业 669
16.6.1. 某历史养成类游戏开发公司实时计算和数据仓库方案 669
16.6.2. 某手游开发公司在离线一体分析方案 670
16.6.3. 某中生代游戏公司广告业务HBase搬站 671
16.6.4. 云数据库Lindorm助力趣加互娱低成本上云 672
16.7. 教育行业 674
16.7.1. 某在线教育领导品牌推荐业务迁移至阿里云Lindorm 674
16.8. 通用行业 676
16.8.1. Napatech案例 676
16.8.2. 图扑案例 677
16.8.3. 东软案例 680
17.解决方案 685
17.1. HDFS云原生上云方案 685
17.2. 互联网账单解决方案 685
17.3. 大数据用户画像解决方案 686
17.4. 车联网数据存储处理方案 687
17.5. 低成本RDS历史库 688
17.6. Lindorm for Cassandra应用实践依赖 690
18.实践教程 694

18.1. 从云HBase标准版/自建开源HBase迁移到Lindorm宽表引擎 694
18.2. 通过LTS批量加载离线数据到Lindorm 703

1.有奖问卷调研

Lindorm文档体验调研正在进行中。花3分钟参与问卷调研，反馈您对Lindorm文档的点赞、吐槽与建议，有机会获得阿里云精美周边礼品。
点此填写问卷，赢取礼品

2.动态与公告

2.1. 产品公告

2.1.1. Lindorm容量型云存储价格变动

由于成本变动，Lindorm容量型云存储价格由0.11元/GB调整为0.12元/GB。

2.2. 版本说明

2.2.1. 宽表引擎版本说明

为提升用户体验，云原生多模数据库Lindorm会不定期地发布版本，用于丰富云产品功能或修复已知缺陷。您可以参阅本文了解Lindorm宽表引擎的版本更新说明，选择在业务低峰期升级实例的宽表引擎版本。

如何查看或升级实例的宽表引擎版本

您可以通过控制台查看当前的宽表引擎版本，如下图所示，具体操作及升级版本的注意事项，请参见升级版本。



> **注意**
>
> - 系统会自动检测实例的宽表引擎版本，如果**小版本升级**按钮不存在，表示该实例的宽表引擎版本已经是最新的。
> - 由于各地域版本发布进度可能有所差异，宽表引擎版本发布情况以当前控制台显示为准。

版本说明

版本号	类型	说明
2.2.21	功能优化	• 优化写入时DirectMemory消耗。 • 优化负载均衡策略。 • 优化Compaction的参数，并支持自适应调节。 • 优化索引构建速度。 • 优化SQL的读写性能。
2.2.20	功能优化	• 优化大kv的写入性能。 • Lindorm SQL支持读取多版本和自定义时间戳写入功能。 • 修复其他性能问题。
2.2.19	新功能	Lindorm SQL支持动态列功能，支持SQL层监控功能。
	功能优化	• 修复Lindorm SQL表属性修改后元数据更新滞后的问题。 • 修复稳定性问题。
2.2.18	功能优化	• 优化Lindorm SQL的性能。 • 修复Lindorm SQL中Date类型和Time类型的时区问题。 • 修复稳定性问题。

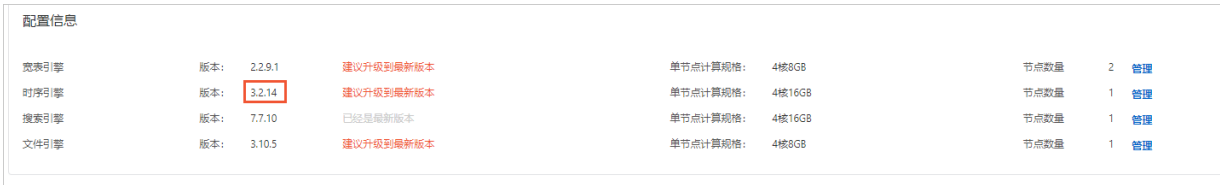
版本号	类型	说明
2.2.17	功能优化	• 修复Lindorm SQL的BUG。 • 优化冷存储Region的上线速度，使用冷存储的用户，建议将宽表引擎版本升级至2.2.17以上。
2.2.16	功能优化	• 完善Lindorm SQL的语法，使用SQL的用户建议将宽表引擎版本升级至2.2.16以上。 • 优化刷盘逻辑，避免Region过多时出现内存不足问题。 • 修复其他BUG。
2.2.15	功能优化	优化Lindorm SQL的使用，修复某种情况下建表失败和强一致表访问失败的问题。建议升级至此版本以上。
2.2.12	功能优化	• 优化compaction策略，提高效率。 • 优化Scan返回的数据量。
2.2.11	新特性	• 支持Lindorm SQL。 • 支持高性能原生二级索引。
2.2.10	功能优化	修复冷存储在网络抖动时遇到请求报错的问题。
2.2.9	新特性	多维索引SearchIndex功能支持动态生成分词器的配置，并修复部分缺陷问题。
2.2.8	功能优化	优化zstd压缩时的内存占用问题。
2.2.7	新特性	支持Lindorm数据按顺序导出，需要同时使用LTS。
2.2.5	新特性	支持Lindorm增量数据的导出，需要同时使用LTS。
2.2.1	功能优化	优化多维索引SearchIndex功能，并修复部分缺陷问题。
2.1.30	功能优化	优化写热点情况下flush region的选择算法问题。
2.1.26	功能优化	优化小规格实例中BucketCache的内存overhead的问题。
2.1.25	功能优化	减少了Reverse Scan时取回的冗余数据量。
2.1.23	功能优化	优化小内存机型的内存管理问题。
2.1.21	功能优化	建议升级至此版本以上。 • 优化写入吞吐的问题。 • 优化Reverse Scan的性能问题。 • 优化冷热分离功能中热存储数据归档至冷存储的速度问题。
2.1.17	功能优化	优化多维索引SearchIndex功能的使用。
2.1.15	功能优化	优化多维索引SearchIndex功能，并修复部分缺陷问题。
2.1.12	功能优化	优化多维索引SearchIndex功能，并修复bulkload过程中遇到的问题。
2.1.10	新特性	支持维索引SearchIndex功能。
	功能优化	优化冷热分离功能。
2.1.9	新特性	支持多维索引SearchIndex功能，并修复部分缺陷问题。
2.1.8	新特性	支持使用冷存储和冷热分离功能。

2.2.2. 时序引擎版本说明

为提升用户体验，云原生多模数据库Lindorm会不定期地发布版本，用于丰富云产品功能或修复已知缺陷。您可以参阅本文了解Lindorm时序引擎的版本更新说明，选择在业务低峰期升级实例的时序引擎版本。

如何查看或升级实例的时序引擎版本

您可以通过控制台查看当前的时序引擎版本，如下图所示，具体操作及升级版本的注意事项，请参见升级版本。



> **注意**
>
> - 系统会自动检测实例的时序引擎版本，如果**小版本升级**按钮不存在，表示该实例的时序引擎版本已经是最新的。
> - 由于各地域版本发布进度可能有所差异，时序引擎版本发布情况以当前控制台显示为准。

版本说明

小版本号	发布日期	类型	说明
3.4.17	2022年06月17日	新特性	支持设置写入的最大时间（即当前时间和偏移时间），超过偏移时间后写入的数据会被丢掉。
		功能优化	• 更改数据最小时间校验逻辑。 • 修复SQL负科学计数法写入报错的问题。 • 修复InfluxDB行协议写入科学计数法报错的问题。 • 修复连续查询操作执行失败的问题。 • 修复 `COUNT(1)` 在嵌套子查询操作中报错的问题。
3.4.16	2022年05月31日	新特性	• 降采样查询支持基于本地时区的降采样。 • 最新值查询语法支持多列查询。 • 支持查询和管理正在执行的SQL。 • PromQL访问方式支持元数据查询。
3.4.15	2022年05月16日	新特性	降采样查询支持Offset能力。
		功能优化	• SQL查询不带时间条件时默认开始时间设置为0。 • 修复在降采样查询中COUNT函数和其他算子混合使用时报类型转换错误的问题。 • 修复在PrepareStatement语句中使用Limit不支持Long类型的问题。 • 修复存储锁住后无法手动删除数据的问题。 • 修复SQL接口（api/v2/sql）对于长链接连续发起查询操作造成SQL执行异常的问题。 • 修复TTL设置为0不生效的问题。
3.4.14	2022年04月25日	新特性	• TSQL访问区分读写线程池。 • 连续查询语句支持开启记录执行日志。
		功能优化	• 修复最新值查询执行UNION操作报错的问题。 • 修复TSQL在连接（通过JDBC或者SQL接口）断开后未调用close的问题。 • 修复行协议在并发更新Schema操作时出现冲突的问题。

小版本号	发布日期	类型	说明
3.14.13	2022年04月14日	新特性	• 支持SHOW CREATE TABLE语句。 • 连续查询语句支持自定义窗口长度选项。 • 执行DESCRIBE TABLE语句的返回数据支持按照创建顺序显示。
		功能优化	• 修复prepared statement不支持降采样查询的问题。 • 行协议写入时新增约束条件：tag列和field列不能有相同的列名。 • 修复连续查询在开启权限管理后无法正常运行的问题。
3.14.12	2022年03月30日	新特性	• 通过设置不同的TTL支持不同的预降采样。 • BIGINT类型的列写入DOUBLE类型的数据时支持默认转换成BIGINT类型。
		功能优化	• 修复最新值查询BOOLEAN类型的数据时返回结果异常问题。 • SQL断开连接后支持取消查询操作。
3.4.11	2022年03月16日	新特性	• SQL语句支持用户自定义时间列名。 • SQL语句支持修改时序表的列类型。
		功能优化	• 修复部分查询场景distinct报错的问题。 • 修复最新值查询在扩容后可能会出现返回多条数据的问题。 • 修复降采样查询中不支持常量折叠计算的问题。 • 修复正则查询降采样数据时间条件不生效的问题。
3.4.10	2022年02月28日	新特性	• SQL查询语句tags支持通过正则表达式过滤。 • 支持non_negative_rate和non_negative_delta时序函数。 • 增加table/database/CQ/column的长度限制。
		功能优化	• 优化大量数据点查询遍历的性能问题。 • 修复列数据类型相同的两张时序表无法结合的问题。
3.4.9	2022年02月14日	新特性	• DESCRIBE TABLE语法支持返回PRIMARY KEY。 • SQL语句区分大小写。 • SQL语句支持写入NULL的数据源（tag）或者数据内容（field）。 • 支持SQL管控预降采样规则，并支持表级别的降采样规则。 • 完善SQL语句错误返回信息的规范，例如对返回错误码以及对应的描述进行了规范。
		功能优化	• 修复最新值查询会返回时间范围之外的数据。 • 优化创建连续查询（Continuous Query，简称CQ）时，interval参数需要反引号的问题。
3.4.8	2022年02月07日	功能优化	修复HASH策略的BUG问题。
3.4.7	2022年01月26日	新特性	• TSQL写入语句限制至少一个Field列。 • TSQL支持创建或者删除预降采样规则。 • TSQL支持使用降采样为0。 • TSQL RATE函数和DELTA函数支持嵌套使用。 • PromQL支持basic鉴权，并支持指定DB参数。 • 行协议写入支持强Schema约束。 • 增加表长度限制，限制长度为128个字节。

小版本号	发布日期	类型	说明
		功能优化	• 修复表反复删除可能会出现数据漏删除的问题。 • 修复写入整数到两个列失败的问题。 • 修复嵌套子查询单列降采样时报错的问题。
3.4.6	2022年01月11日	新特性	• InfluxDB行协议支持指定Schema策略，支持按需创建Schema。 • TSQL DATABASE/TABLE区分大小写。 • 优化TSQL查询性能。 • TSQL支持GROUP BY查询。 • TSQL支持预降采样查询。 • TSQL支持ALTER TABLE语法。 • TSQL插入操作中默认Schema一致性校验。 • /api/v2/sql接口支持chunked流式交互查询。
		功能优化	• 修复cache flush异常会导致flush中断的问题。 • 修复执行Continuous Query时出现JDBC前缀不匹配导致执行失败的问题。 • 修复查询和写入SQL对时间字符串处理不一致的问题。 • 修复sample by fill与percentile使用会抛异常的问题。 • 修复ACL关闭时创建用户会出现NPE的问题。
3.4.5	2021年12月24日	新特性	• MySQL支持鉴权。 • TSQL提高简单查询的性能。 • TSQL支持分位数函数。 • API支持field查询。 • 优化时间分区功能，支持关闭时间分区功能。 • 行协议和API接口支持通过修改SQL可配置Schemaless模式，默认为schemaless。
		功能优化	• 修复删除表操作后重新写入数据会丢失数据的问题。 • 修复api/v2/write接口格式错误时出现的NPE问题。 • 修复/api/mquery接口查询多值降采样错误。 • 修复历史文件的存储容量没有达到最大上限时无法迁移至冷存储的问题。 • 修复查询操作可能出现的栈溢出问题。 • 修复string类型的field会导致compaction异常的问题。
3.4.4	2021年12月14日	新特性	• SQL支持IF EXISTS语法。 • SQL支持SHOW PRIVILEGES和SHOW PARAMETER语法。 • 降采样interval支持毫秒（ms）。 • CREATE TABLE语法必须指定TIME和FIELD列，并支持指定分区规则。 • 限制最多Field列数为1024。 • 删除默认结束时间为当前时间的逻辑。 • 支持RATE函数查询和DELTA函数查询。 • 优化pipeline处理逻辑，提升查询性能。 • DESCRIBE TABLE支持返回列的类型。 • 优化字符串类型数据的读写压缩算法，提升写入性能。 • 预降采样规则avg默认拆成为cunt或者sum。

小版本号	发布日期	类型	说明
		功能优化	• 修复字符串降采样查询失败的问题。 • 修复调用commit方法或者rollback方法报错的问题。 • 修复执行插入数据语句不指定时间列报错不友好的问题。 • 修复SQL执行JOIN语句报错的问题。 • 修复预降采样内存数据Count错误的问题。
3.4.3	2021年12月1日	新特性	• Grafana插件支持Database和用户认证功能。 • 预降采样使用独立的Flush线程，能够提升预降采样性能。 • DESCRIBE DATABASE语法支持返回Database属性。 • 取消TTL和冷热分界线之间的限制。 • 创建实例时默认关闭时间分区功能。 • 新增了表写入数据量的统计，支持通过Lindorm-cli查询。 • 字符串处理修改为字节数组，并支持proxy对于字符串的压缩。 • 创建Database支持指定分区数量。
		功能优化	• 修复时间分区切分导致的Flush阻塞问题。 • 修复Database在创建或者删除时的稳定性问题。 • 修复更新Database时出现NPE（Null Pointer Exception）的问题。 • 修复执行插入数据语句时间列时间范围不对时报错不准确的问题。 • 修复在跨分区使用SQL查询出现重复数据的问题。 • 修复API查询时，endtime边界不包含的问题。 • 修复无法写入空字符串的问题。 • 修复预降采样混合类型出现错误的问题。
3.4.2	2021年11月15日	新特性	• TSQL latest/downsample使用流式pipeline。 • 流式pipeline支持BIGINT数据类型。 • TSQL增加鉴权语法，并支持宽表SQL。 • Database支持skip_wal选项。 • TSQL降采样算法下推TSCore优化。 • PromQL支持多值查询。
		功能优化	• 修复TSQL对中文过滤条件的支持度问题。 • 修复预降采样AVG查询匹配单个文件的问题。 • 修复写入数据时删除表可能会导致查询异常的问题。 • 修复WAL回滚时或文件更换时可能出现的问题。 • 修复最新值查询会出现NPE（Null Pointer Exception）的问题。 • 修复Schema更新问题导致TSQL写入错误的问题。 • 修复suggest会返回重复的metric的问题。 • 修复Database name不支持小于3个字符的问题。 • 修复TSQL已经创建的连接对于新创建的Database的可见性问题。
3.3.4 （稳定版本）	2021年10月28日	功能优化	修复最新值查询会出现NPE（Null Pointer Exception）的问题。
		新特性	• 优化TSQL查询，新增全新流式引擎设计。 • TSFile文件内部字典化，提升压缩率。 • TSQL去掉查询TSCore的Schema逻辑，提升TSQL查询性能，SDK写入的数据需要提前创建表才能使用TSQL访问。 • TSQL支持DML操作错误提示。

小版本号	发布时间	类型	说明
3.4.1	2021年10月22日	功能优化	• 修复使用details会导致SDK无法收到response的问题。 • 修复大量时间线存在情况下可能会导致TSCore Schema内存溢出的问题。 • 修复查询多个分区场景下，大查询操作会导致TSCore FullGC的问题。 • 修复多个TSCore可能导致无法查出数据的问题。
3.4.0	2021年10月13日	新特性	• TSQL支持BIGINT类型。 • TSQL插入语句支持写入NULL值。
		功能优化	提升并行查询数据效率。
3.3.3	2021年09月08日	新特性	• SDK支持查询预降采样。 • OpenTSDB协议支持GROUP_BY操作。
		功能优化	• 修复WAL的Flush线程在OOM时退出导致写入卡住的问题。 • 修复预降采样刷盘失败后WAL日志堆积的问题。
3.3.2	2021年08月25日	新特性	默认开启时间分区，默认周期为30天。
		功能优化	• 优化单值数据写入多值接口报错信息不准确的问题。 • 修复默认分区存在无法执行TTL的问题。 • 修复覆盖写场景下Flush后导致重复时间点数据的问题。
3.3.1	2021年08月12日	新特性	• 支持配置通用压缩算法，默认不开启通用压缩。 • 行协议支持科学计数法表示。 • 更新特殊字符校验策略。
		功能优化	• 优化compaction策略，提高效率。 • 优化JVM参数。 • 修复插值对齐问题。 • 优化在集群间查询中断场景下RPC的处理流程。 • 修复删除表后，新创建的同名表无法写入数据的问题。
3.3.0	2021年07月28日	新特性	支持PromQL查询。
		功能优化	• 修复SDK出现连接关闭的问题。 • 修复TSQL无法查询单值的问题。 • 修复TSQL别名设置无法生效的问题。

2.2.3. 文件引擎版本说明

为提升用户体验，云原生多模数据库Lindorm文件引擎（底层存储）会不定期地发布版本，用于丰富云产品功能或修复已知缺陷。您可以参阅本文了解Lindorm文件引擎（底层存储）的版本更新说明，选择在业务低峰期升级实例的文件引擎（底层存储）版本。

如何查看或升级实例的文件引擎版本

您可以通过控制台查看当前的文件引擎（底层存储）版本，如下图所示，具体操作及升级版本的注意事项，请参见升级版本。



注意

- 版本升级过程中会滚动重启数据库所有节点的底层存储服务，在重启期间会有轻微的抖动，建议您在业务低峰期操作。
- 系统会自动检测实例的文件引擎版本，如果**小版本升级**按钮不存在，表示该实例的文件引擎版本已经是最新的。
- 由于各地域版本发布进度可能有所差异，文件引擎版本发布情况以当前控制台显示为准。

版本说明

小版本号	类型	说明
3.10.10.1	功能优化	优化系统稳定性。
3.10.7	功能优化	优化冷存储功能和容量型存储的稳定性。
3.10.6	功能优化	优化系统稳定性。

2.2.4. 搜索引擎版本说明

为提升用户体验，云原生多模数据库Lindorm搜索引擎会不定期地发布版本，用于丰富云产品功能或修复已知缺陷。您可以参阅本文了解Lindorm搜索引擎的版本更新说明，选择在业务低峰期升级实例的搜索引擎版本。

如何查看或升级实例的搜索引擎版本

您可以通过控制台查看当前的搜索引擎版本，如下图所示，具体操作及升级版本的注意事项，请参见升级版本。



注意

- 版本升级过程中会滚动重启搜索引擎所有节点，在此期间会自动移动索引分片到可用节点，应用访问会有轻微抖动，建议您在业务低峰期操作。
- 如果搜索引擎节点数量只有1个，升级过程可能中会导致应用访问中断，建议提前扩容，确保服务高可用。
- 系统会自动检测实例的搜索引擎版本，如果**小版本升级**按钮不存在，表示该实例的搜索引擎版本已经是最新的。
- 由于各地域版本发布进度可能有所差异，搜索引擎版本发布情况以当前控制台显示为准。

版本说明

小版本号	类型	说明
8.10.1	新特性	• 支持更多SQL的。下推算子 • 支持冷存储功能，具体请参见配置冷存储。
7.7.13	安全升级	合入安全补丁，增强系统安全。
7.7.12	新特性	支持使用开源的Solr JDBC访问搜索引擎，具体请参见通过SQL连接并使用搜索引擎。
7.7.11	新特性	• 支持实时检索，写入数据后立即可查。 • 支持动态设置数据的存活时间（Time To Live，简称TTL）。 • 支持对嵌套类型的文档创建索引。
7.7.10	新特性	• 预排序支持提前终止和加速查询的功能。 • 新增获取索引表大小的接口。

2.3. 安全公告

2.3.1. 【通知】Apache Log4j2远程代码执行漏洞（CVE-2021-44228/CVE-2021-45046）对云原生多模数据库Lindorm的影响

近日，阿里云计算有限公司发现阿帕奇Log4j2组件存在远程代码执行漏洞。该漏洞对云原生多模数据库Lindorm的搜索引擎有影响，对宽表引擎、时序引擎和文件引擎无影响。

漏洞描述

更多信息，请参见【漏洞通告】Apache Log4j2 远程代码执行漏洞（CVE-2021-44228/CVE-2021-45046）。

影响范围

云原生多模数据库Lindorm的搜索引擎7.7.13以下版本受此漏洞影响。

修复方式

如果您已经开通云原生多模数据库Lindorm的搜索引擎，并且搜索引擎版本为7.7.13以下版本，可以通过以下两种方法修复Apache Log4j2远程代码执行漏洞：

- 将云原生多模数据库Lindorm搜索引擎版本升级至7.7.13及以上版本，升级版本的方法请参见升级小版本。
- 请提交工单，云原生多模数据库Lindorm产品团队为您修复漏洞。

3.产品简介

3.1. 什么是云原生多模数据库Lindorm

本文介绍云原生多模数据库Lindorm的产品概要。

产品介绍

Lindorm是面向物联网、互联网、车联网等设计和优化的云原生多模超融合数据库，支持宽表、时序、文本、对象、流、空间等多种数据的统一访问和融合处理，并兼容SQL、HBase/Cassandra/S3、TSDB、HDFS、Solr、Kafka等多种标准接口和无缝集成三方生态工具，适用于日志、监控、账单、广告、社交、出行、风控等场景，Lindorm也是为阿里巴巴核心业务提供支撑的数据库之一。

Lindorm基于存储计算分离、多模共享融合的云原生架构，具备弹性、低成本、简单易用、开放、稳定等优势，适合元数据、日志、账单、标签、消息、报表、维表、结果表、Feed流、用户画像、设备数据、监控数据、传感器数据、小文件、小图片等数据的存储和分析，其核心能力包括：

- 多模超融合：支持宽表、时序、对象、文本、队列、空间等多种数据模型，模型之间数据互融互通，具备数据接入、存储、检索、计算、分析等一体化融合处理与服务的能力，帮助应用开发更加敏捷、灵活、高效。
- 高性价比：支持千万级高并发吞吐、毫秒级访问延迟，并通过多级存储介质、智能冷热分离、自适应特征压缩，大幅减少存储成本。
- 云原生弹性：支持计算资源、存储资源独立弹性伸缩，并提供按需即时弹性、按使用量付费的Serverless服务。
- 开放兼容：兼容SQL、HBase/Cassandra/S3、TSDB、HDFS、Solr、Kafka等多种标准接口，支持与Hadoop、Spark、Flink、Kafka等系统无缝打通，并提供简单易用的数据交换、处理、订阅等能力。

多模介绍

Lindorm支持宽表、时序、对象、文件、队列、空间等多种数据模型，提供标准SQL和开源接口两种方式，模型之间数据互融互通，帮助应用开发更加敏捷、灵活、高效。多模型的核心能力主要由以下几大数据引擎提供，包括：

- 宽表引擎
  负责宽表与对象数据的管理和服务，具备全局二级索引、多维检索、动态列、TTL等能力，适用于元数据、订单、账单、画像、社交、feed流、日志等场景，兼容SQL、HBase、Cassandra（CQL）、S3等标准接口。
  支持千万级高并发吞吐，支持百PB级存储，吞吐性能是开源HBase（Apache HBase）的3~7倍，P99时延为开源HBase（Apache HBase）的1/10，平均故障恢复时间相比开源HBase（Apache HBase）提升10倍，支持冷热分离，压缩率比开源HBase（Apache HBase）提升一倍，综合存储成本为开源HBase（Apache HBase）的1/2。
- 时序引擎
  负责时序数据的管理和服务，主要面向工业、IoT、监控等领域的量测数据、监控数据以及设备运行数据提供基于SQL的管理、写入、查询能力。针对时序数据设计的压缩算法，压缩率可达15:1。支持海量数据的多维查询和聚合计算，同时也支持时序数据的预降采样和持续查询。
- 搜索引擎
  负责多模数据的检索分析加速，其基于列存、倒排等核心技术，具备全文检索、聚合计算、复杂多维查询等能力，适用于日志、账单、画像等场景，兼容SQL、开源Solr等标准接口。
- 文件引擎
  负责目录文件数据的管理和服务，并提供宽表、时序、搜索引擎底层共享存储的服务化访问能力，从而加速多模引擎底层数据文件的导入导出及计算分析效率，兼容开源HDFS标准接口。
- 计算引擎
  计算引擎与Lindorm存储引擎深度融合，基于云原生架构提供的分布式计算服务，满足用户在数据生产、交互式分析、机器学习和图计算等场景的计算需求，兼容开源Spark标准接口。
- 流引擎
  云原生多模数据库Lindorm流引擎是面向流式数据处理的引擎，提供了流式数据的存储和轻计算功能，帮助您轻松实现流式数据存储至云原生多模数据库Lindorm，构建基于流式数据的处理和应用。

3.2. Lindorm产品选型

创建云原生多模数据库Lindorm实例前，您需要结合业务场景、工作负载、价格等因素选择符合预期的方案。本文从引擎类型、存储类型、节点规格和数量三个方面帮助您进行产品选型。

选择引擎类型

云原生多模数据库Lindorm支持多种引擎类型，兼容多种开源标准接口，同时提供SQL查询、时序处理、文本检索分析等能力。由于不同引擎类型适用不同的业务场景，您可以根据业务场景需求选择一种或多种引擎类型，具体介绍请参见引擎类型。

选择存储类型

云原生多模数据库Lindorm依赖于LindormStore云原生存储系统，实现数据存储和计算分离解耦。一个云原生多模数据库Lindorm实例的多个引擎可以共享存储容量。不同存储类型的数据访问延时不同，具体介绍请参见存储类型。

选择节点规格和数量

选择引擎类型的节点规格和节点数量时，一般情况下需要考虑业务的读写请求量。 当每条请求的数据量大小在0~10 KB以内并且没有复杂模式的查询时，节点规格和数量的选型建议如下表：

TPS和QPS（次/秒）	节点规格和数量	业务场景
0~1000	建议选择2台4核16 GB。	适合低负载场景访问。不推荐选择最低规格，即4核8 GB，因为4核8 GB规格的内存容量过小，在业务高峰或者大KV访问期间容易出现内存溢出导致任务被终止等不稳定现象。
1000~20000	建议选择2~3台8核32 GB。	适合中低负载场景访问，与8核16 GB规格相比价格不高，但内存容量翻倍，稳定性好。
20000以上	选择节点规格：8核32 GB、16核32 GB、16核64 GB、32核64 GB或者32核128 GB。 节点数量根据业务的访问模式和访问流量选择。	大内存节点规格适用于以下两个场景： • 如果是在线型业务，建议选择大内存机型来提高性能。 • 如果需要运行MR、Spark等离线大负载任务或者TPS和QPS非常高，建议选择大内存机型。

> **说明** 目前云原生多模数据库Lindorm支持的节点规格都是独享规格。独享规格是指完全独享的CPU和内存，性能长期稳定，不会因为物理机上其它实例的行为而受到影响。

选择引擎类型的节点规格和节点数量时，不仅需要考虑业务的读写请求量，而且还要综合考虑多方面因素。如果您在创建云原生多模数据库Lindorm实例时选型方面需要帮助，请提交工单。

常见问题

Q：在满足业务需求的前提下，购买云原生多模数据库Lindorm实例时应该选择高规格少节点数量的实例还是低规格多节点数量的实例？

A：建议选择高规格少节点数量的实例。云原生多模数据库Lindorm支持水平无限扩展，在访问比较均衡的情况下，如果出现高负载、高延迟、业务抖动等情况时，可以通过水平增加节点数量解决。但是当节点遇到瞬时的大请求或者偶发的热点流量时，单机的规格决定了抗热点的能力，此时单机的规格越高，抗抖动能力越强，业务整体的稳定性会更好。

如果您需要升级Lindorm实例的节点规格，可以通过云原生多模数据库Lindorm控制台升级规格，请参见变更实例规格。

3.3. 产品系列

云原生多模数据库Lindorm包括四个系列：Lindorm、Lindorm Serverless、Lindorm单节点和Lindorm Tunnel通道服务。本文介绍各产品系列的适用场景和查看方法。

产品系列说明

创建云原生多模数据库Lindorm实例时，可以根据各产品系列的使用场景选择**商品类型**。

系列	说明	支持的引擎类型和存储类型	适用场景
Lindorm	Lindorm独立集群的标准化部署方式，各个引擎有各自独立的部署节点，适合对集群资源使用有较高要求的业务。	• 引擎类型：宽表、时序、搜索、文件、计算和流引擎。 • 存储类型：标准型云存储、性能型云存储、容量型云存储、本地SSD盘和本地HDD盘。	• 大中型企业的生产环境。 • 车联网、物联网、IoT工业互联网、风控、账单、广告、Feeds流、数据大屏等。
Lindorm Serverless	Lindorm Serverless是基于Lindorm内核构建的Serverless模式服务，提供超低起步价格，是一种弹性、按需计费的产品形态。	• 引擎类型：宽表引擎。 • 存储类型：标准型云存储和性能型云存储。	• 个人开发者学习。 • 中小型企业的开发测试环境。 • 中小型企业的小规模生产环境。

系列	说明	支持的引擎类型和存储类型	适用场景
Lindorm单节点	Lindorm单节点可以快速创建、升降配的单机模式集群。价格较便宜，适合用来测试或体验Lindorm功能。	• 引擎类型：宽表、时序、搜索、文件和计算引擎。 • 存储类型：高效云盘。	• 个人开发者学习。 • 小型企业的开发测试环境。
Lindorm Tunnel通道服务	Lindorm Tunnel通道服务（简称LTS）是面向Lindorm业务场景特点深度定制的数据生态服务。支持简单易用的数据交换、处理、订阅等能力，满足数据迁移、实时订阅、数湖转存、数仓回流、单元化多活、备份恢复等需求，实现面向云原生多模数据库Lindorm的一站式数据生态服务。更多信息，请参见LTS服务介绍。	不涉及	• 不停服数据迁移。 • 在线和离线业务分离。 • 主备容灾。 • RDS历史库建设。

查看产品系列

您可以通过云原生多模数据库Lindorm控制台的**实例列表**页面查看实例所属的系列，如下图所示：



控制台的服务类型与实例所属的系列对应关系如下表：

控制台的服务类型	系列
Lindorm	Lindorm
Lindorm Serverless版	Lindorm Serverless
Lindorm单机版	Lindorm单节点
Lindorm TS	Lindorm Tunnel通道服务

3.4. 功能特性

Lindorm是一款适用于任何规模、多种模型的云原生数据库服务，其基于存储计算分离、多模共享融合的云原生架构设计，具备弹性、低成本、稳定可靠、简单易用、开放、生态友好等优势。

云原生弹性

- 基于存储计算分离的全分布式架构，支持计算资源和存储资源的独立弹性伸缩。
- 存储资源支持秒级在线扩缩，计算资源（宽表引擎、时序引擎、搜索引擎）支持分钟级在线伸缩。
- 提供按需即时弹性、按使用量付费的Serverless服务，自适应弹性伸缩，无需人工容量管理。

多模超融合

- 多模型之间支持数据互通，搜索引擎可无缝作为宽表引擎、时序引擎的索引存储，加速多维检索与分析。
- 具备数据接入、存储、检索、计算、分析等一体化融合处理与服务的能力

- 支持统一的SQL访问，以及跨多模引擎关联查询。
- 无需二次开发，内置数据转换、同步、订阅等能力

低成本

- 提供性能型、标准型、容量型多种存储规格，可满足不同场景的性价比选择。
- 多种引擎共享统一的存储池，减少存储碎片，降低使用成本。
- 容量型存储单价低于基于ECS本地盘自建。
- 内置深度优化的压缩算法，数据压缩率高达10:1以上，相比snappy提高50%以上。
- 内置面向数据类型的自适应编码，数据无需解码，即可快速查找。
- 支持智能冷热分离，多种存储规格混合使用，大幅降低数据存储综合成本。

高性能

- 宽表引擎：支持千万级并发吞吐，支持百PB级存储，吞吐性能是开源HBase的3-7倍，P99时延为开源HBase的1/10。
- 时序引擎：写入性能和查询性能是InfluxDB的1.3倍，是OpenTSDB的5-10倍。
- 搜索引擎：基于Lucene引擎深度优化，综合性能比开源Solr/ES提升30%。

易用

- 兼容多种开源标准接口，包括HBase/Cassandra/Phoenix、OpenTSDB、Solr，业务可以无缝迁移。
- 云托管服务，免运维。
- 图形化系统管理和数据访问，操作简单。

高可用

- 系统采用分布式多副本架构，集群自动容灾恢复，并提供99.9%以上的SLA保障。
- 支持跨可用区、强一致的容灾能力，具备金融级可用性标准。
- 支持全球多活部署。

高可靠

- 底层多副本存储，99.99999999%的数据可靠性。
- 提供企业级备份能力。
- 在阿里部署上万台，支持过10年天猫双十一，久经验证。

开放生态

- 支持与MySQL、HBase、Cassandra等系统的平滑在线数据搬迁。
- 可轻松与Spark、Flink、Hadoop、MaxCompute等计算引擎无缝对接。
- 支持无缝订阅Kafka、SLS等日志通道的数据，并具备快速处理能力。
- 可以实时订阅Lindorm的增量变更数据，自定义消费。

3.5. 应用场景

Lindorm是阿里云自研的云原生多模型数据库，面向海量多模型数据的低成本存储分析，构建万物互联时代的数据底座。Lindorm支持宽表模型、时序模型，提供自研的宽表引擎、时序引擎和搜索引擎，兼容HBase、Phoenix、OpenTSDB、Solr等多种开源标准接口，提供SQL查询、时序处理、检索分析等能力，满足结构化、半结构化的存储和分析需求，同时支持在线业务和离线业务。

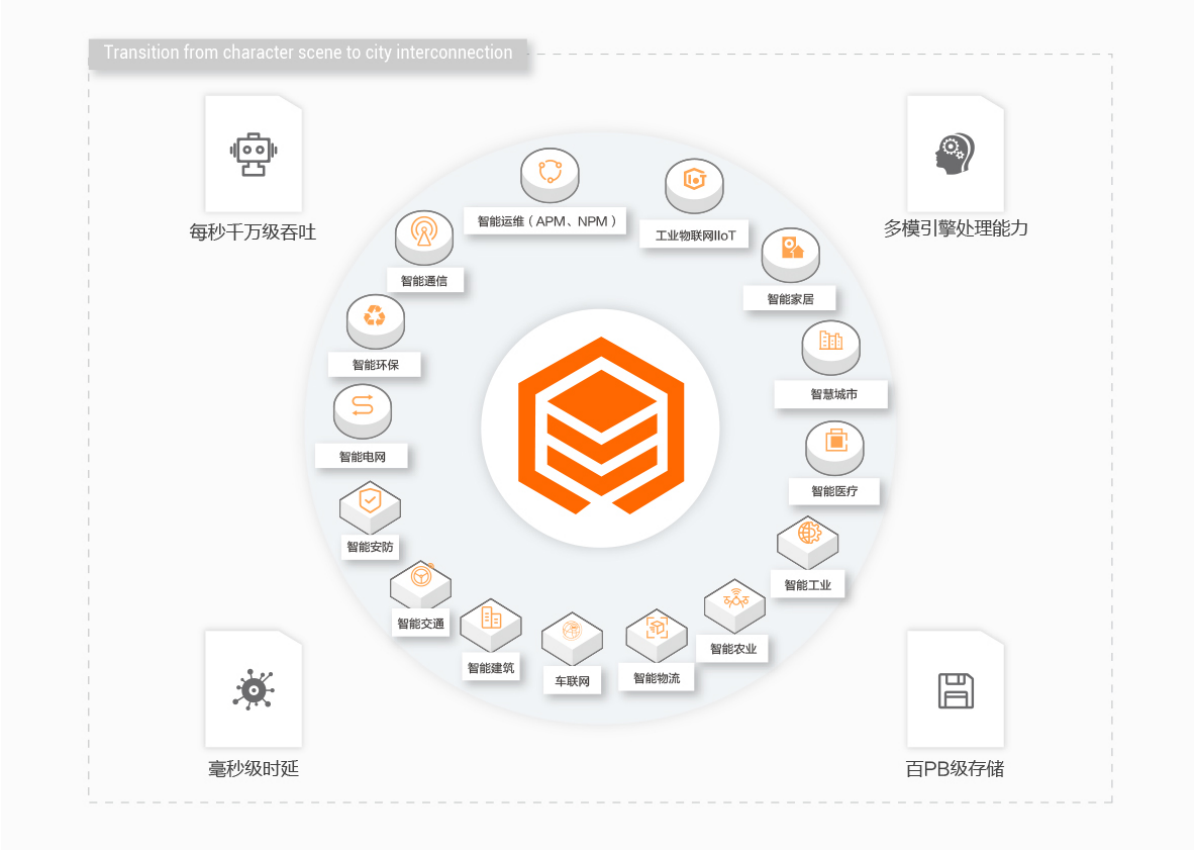
Lindorm可以提供单个毫秒响应的性能，支持水平扩展到PB级存储和千万级QPS，在IoT、淘宝、支付宝、菜鸟等众多阿里巴巴核心服务中起到了关键支撑的作用。

阿里巴巴集团内部最佳实践

- Lindorm在阿里巴巴集团内部成熟业务中得到广泛使用。

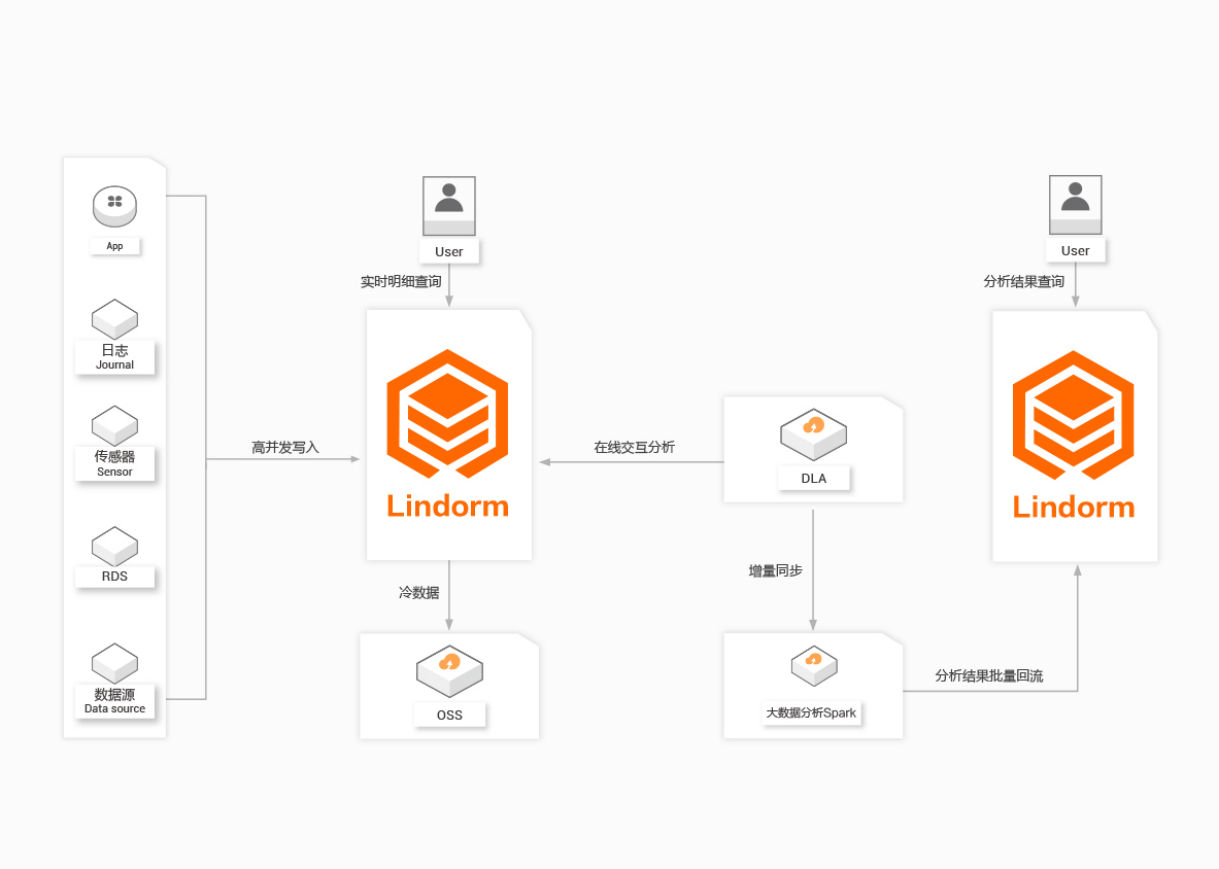


- Lindorm基于自研的云原生多模架构支撑IoT业务飞速发展。



大数据场景：海量数据存储与分析

Lindorm支持海量数据的低成本存储、快速批量导入和实时访问，具备高效的增量及全量数据通道，可轻松与Spark、MaxCompute等大数据平台集成，完成数据的大规模离线分析。

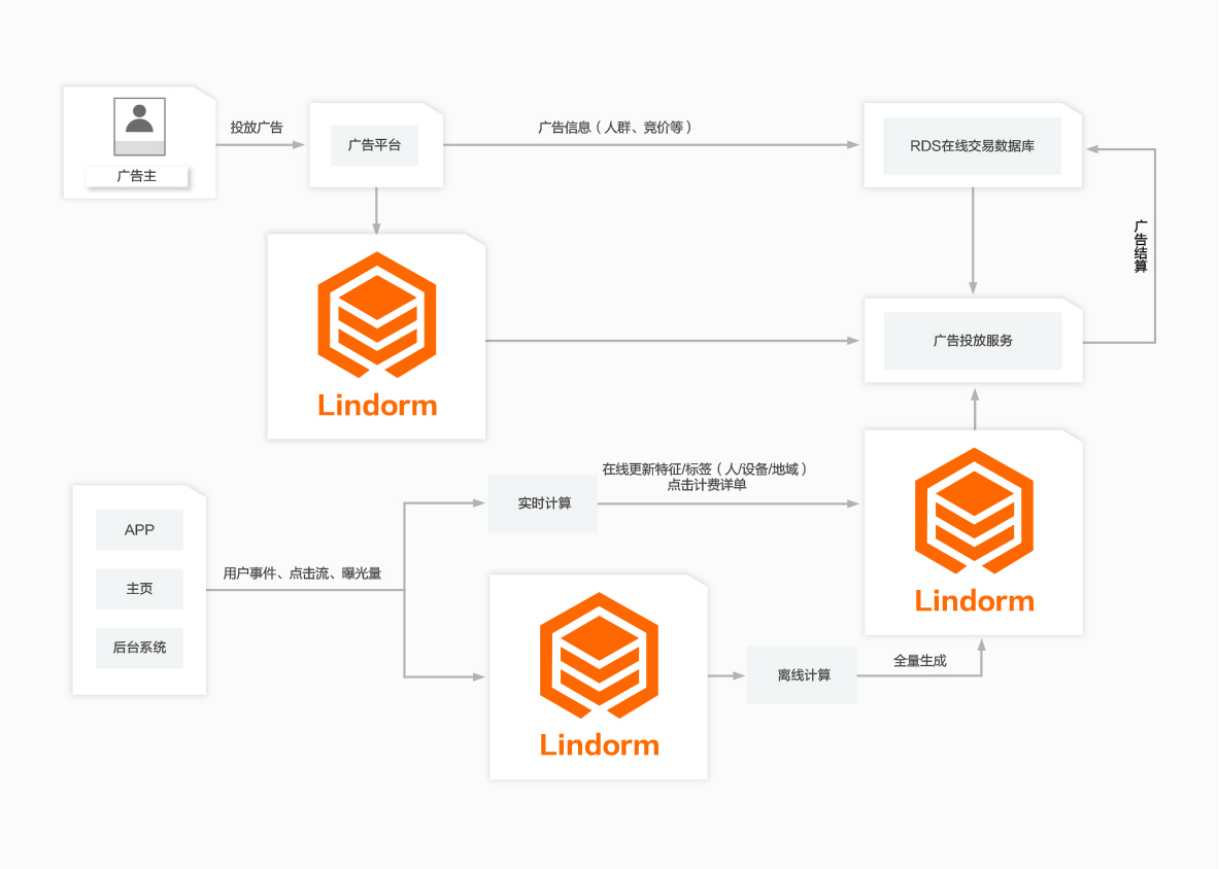


优势

- 低成本：高压缩比，数据冷热分离，支持HDD/OSS存储。
- 数据通道：通过BDS构建云Lindorm与异构计算系统的高效、易用的数据链路。
- 快速导入：通过BulkLoad将海量数据快速导入Lindorm，效率比传统方式提升一个数量级。
- 高并发：水平扩展至千万级QPS。
- 弹性：存储计算分离架构，支持独立伸缩，自动化扩容。

广告场景：海量广告营销数据的实时存储

使用Lindorm存储广告营销中的画像特征、用户事件、点击流、广告物料等重要数据，提供高并发、低延迟、灵活可靠的能力，帮助您快速构建领先的实时竞价、广告定位投放等系统服务。

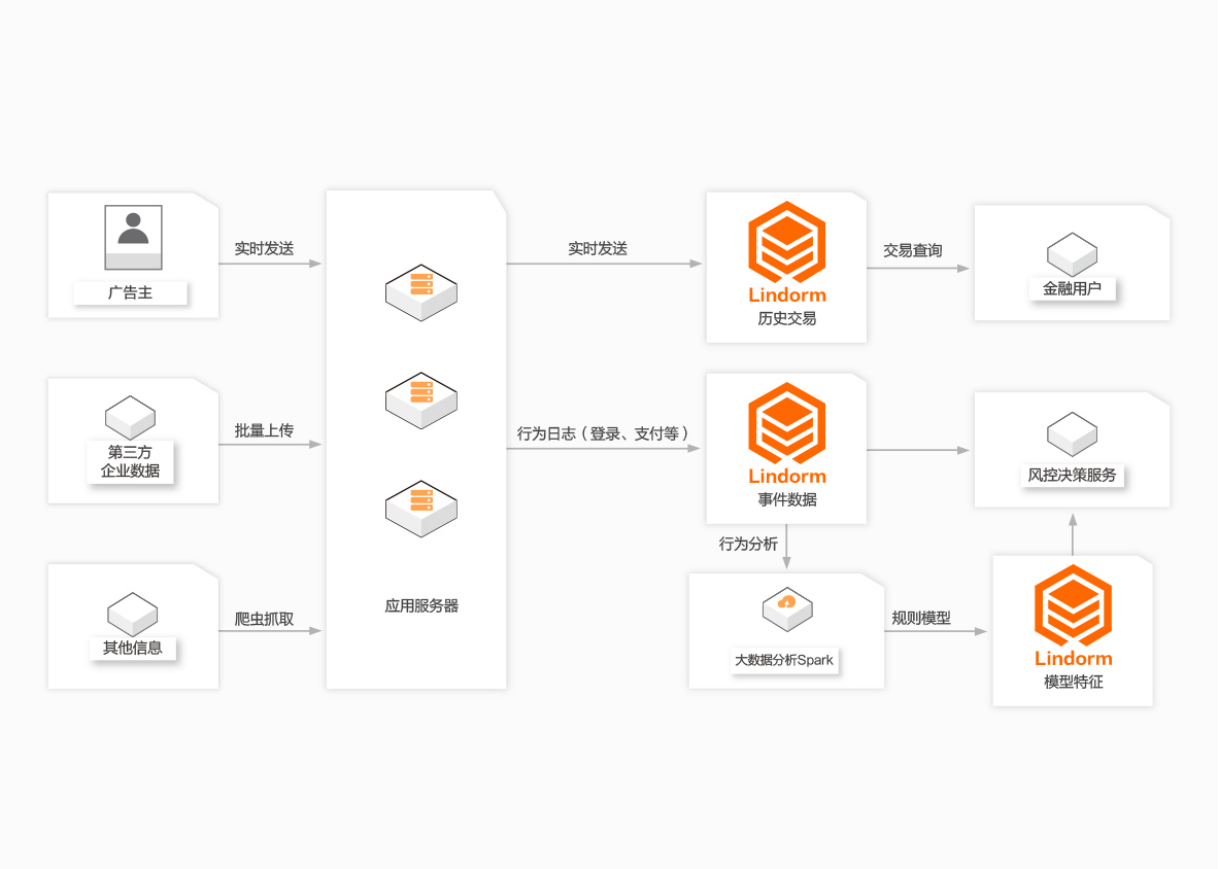


优势

- 低延迟：单个毫秒响应，支持双集群请求并发加速。
- 高并发：水平扩展至千万级QPS。
- 使用灵活：动态列，自由增减特征/标签属性；TTL，数据自动过期。
- 低成本：高压缩比，数据冷热分离，支持HDD/OSS存储。
- 数据通道：通过BDS构建云Lindorm与异构计算系统的高效、易用的数据链路。
- 高可用：主备双活容灾，请求自动容错，满足99.99% SLA。

金融&零售：海量订单记录与风控数据的实时存储

使用Lindorm存储金融交易中的海量订单记录，金融风控中的用户事件、画像特征、规则模型、设备指纹等重要数据，提供低成本、高并发、灵活可靠的能力，帮助您构建领先的金融交易与风控服务。

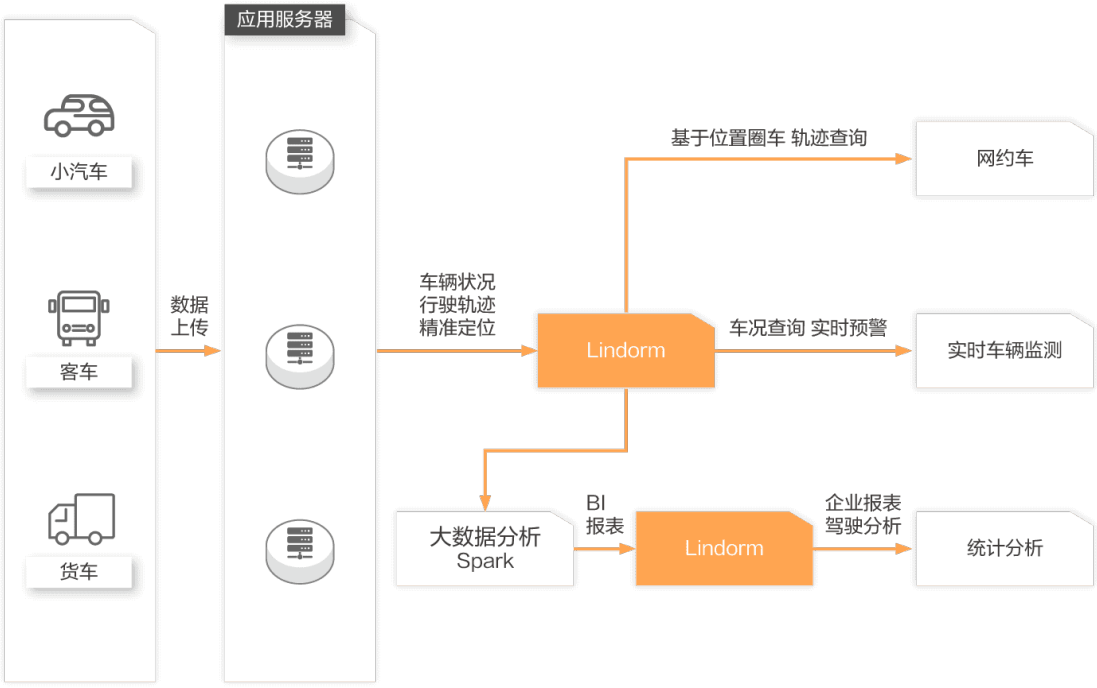


优势

- 低成本：高压缩比，数据冷热分离，支持HDD/OSS存储
- 高并发：水平扩展至千万级QPS
- 使用灵活：动态列，自由增减特征/标签属性；TTL，数据自动过期；多版本
- 低延迟：单个毫秒响应，支持双集群请求并发加速
- 数据通道：通过BDS构建Lindorm与异构计算系统的高效、易用的数据链路
- 高可用：主备双活容灾，请求自动容错，满足99.99% SLA

车联网：车辆轨迹与状况数据的高效存储处理

使用Lindorm存储车联网中的行使轨迹、车辆状况、精准定位等重要数据，提供低成本、弹性、灵活可靠的能力，帮助您构建领先的网约车、物流运输、新能源车检测等场景服务。

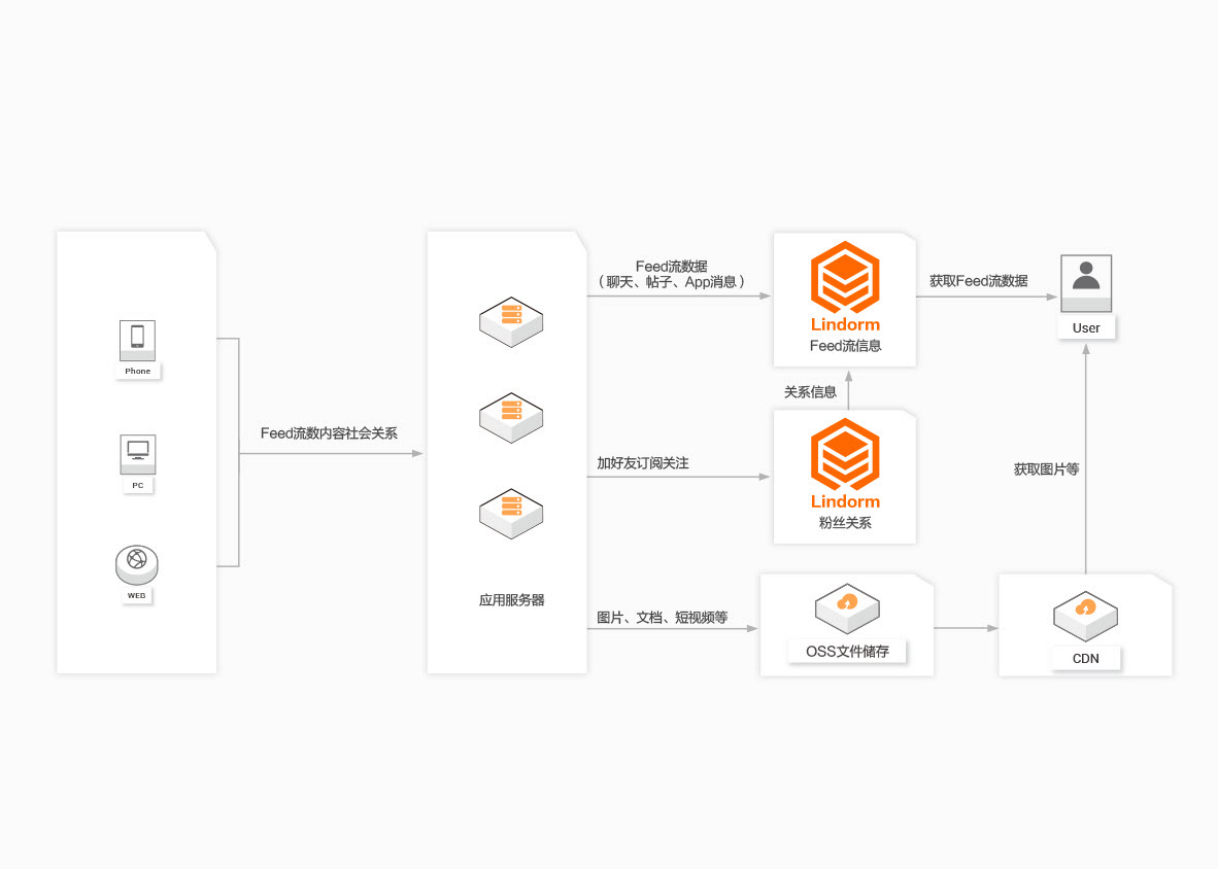


优势

- 低成本：高压缩比，数据冷热分离，支持HDD/OSS存储。
- 弹性伸缩：存储计算分离架构，支持独立伸缩，自动化扩容。
- 使用灵活：动态列，自由增减特征/标签属性；TTL，数据自动过期；多版本。
- 低延迟：单个毫秒响应，支持双集群请求并发加速。
- 数据通道：通过BDS构建Lindorm与异构计算系统的高效、易用的数据链路。
- 高可用：主备双活容灾，请求自动容错，满足99.99% SLA。

互联网社交：高效、稳定的社交Feed流信息存储

使用Lindorm存储社交场景中的聊天、评论、帖子、点赞等重要数据，提供易开发、高可用、延迟的能力，帮助您快速构建稳定可靠的现代社交Feed流系统。



优势

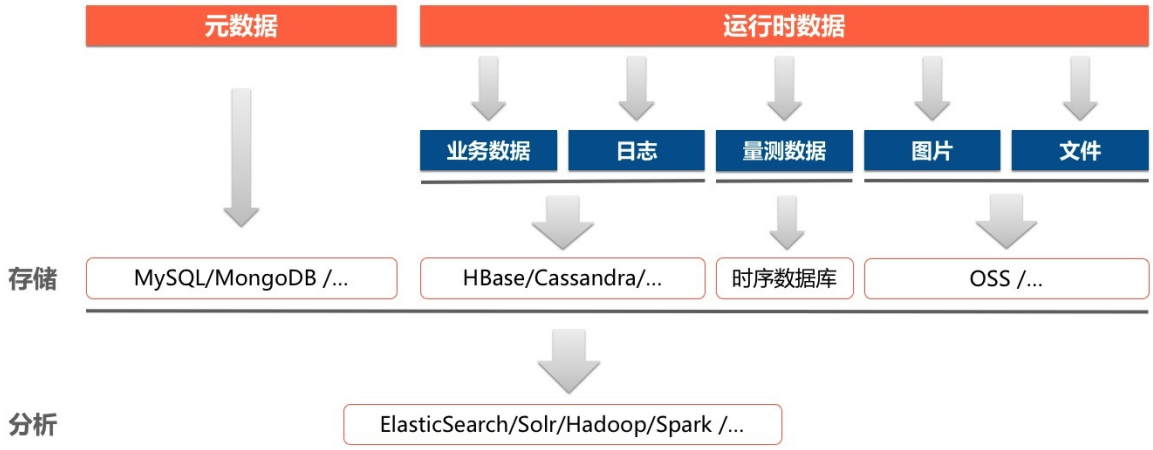
- 易开发：提供社交IM场景专属的FeedStream功能，开发效率和运行性能提升一个数量级。
- 高可用：主备双活容灾，请求自动容错，满足99.99% SLA。
- 低延迟：单个毫秒响应，支持双集群请求并发加速。
- 低成本：高压缩比，数据冷热分离，支持HDD/OSS存储。
- 弹性：存储计算分离架构，支持独立伸缩，自动化扩容链路。

3.6. 产品架构

本文介绍云原生多模数据库Lindorm的产品架构，包括业务背景和总体结构。

业务背景

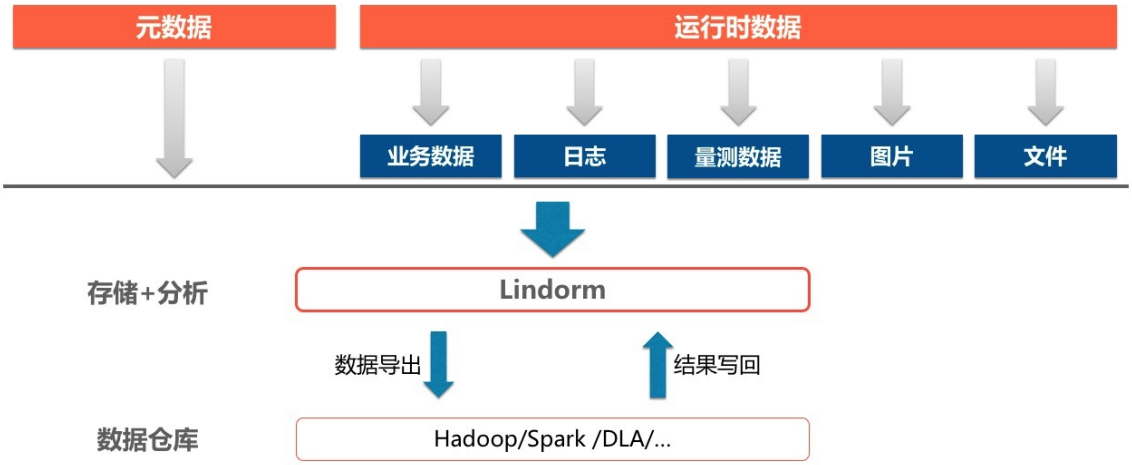
伴随着信息技术的飞速发展，各行各业在业务生产中产生的数据种类越来越多，有结构化的业务元数据、业务运行数据、设备或者系统的量测数据、日志，也有半结构化的业务运行数据、日志、图片或者文件等。按照传统方案，为了满足多种类型数据的存储、查询和分析需求，在设计IT架构时，需要针对不同种类的数据，采用不同的存储分析技术，如下图：



这种技术方案，是一种典型的技术碎片化的处理方案。针对不同的数据，使用不同的数据库来处理。有如下几个弊端：

- 涉及的技术组件多且杂
- 技术选型复杂
- 数据存取、数据同步的链路长

这些弊端会对信息系统建设带来巨大的问题，对技术人员要求高、业务上线周期长、故障率高、维护成本高。更进一步，技术碎片化导致技术架构割裂，不利于技术架构的演进和发展，最终导致技术架构无法跟进业务前进的步伐。举个简单的例子，当业务发展，需要支持跨可用区高可用、全球同步或者降低存储成本时，各技术组件都需要独立演进和发展，耗时耗人耗力，是一件非常痛苦的事情。并且随着业务的发展，数据的类型会越来越多，对不同种类数据的差异化处理需求会日渐增加，会导致数据存储碎片化更加严重。
当前信息化技术发展面临的一个主要矛盾是"日益多样的业务需求带来的多种类型数据与数据存储技术架构日趋复杂成本快速上升之间的矛盾"。伴随5G、IoT、智能网联车等新一代信息技术的逐步普及应用，这个矛盾会越来越突出。为了解决这个问题，阿里云自研了云原生多模数据库Lindorm，满足多模型数据的统一存储、查询和分析需求。如下图所示，与传统方案相比，Lindorm系统极大地简化数据存储技术架构设计，大幅度提升系统稳定性，降低建设成本投入。

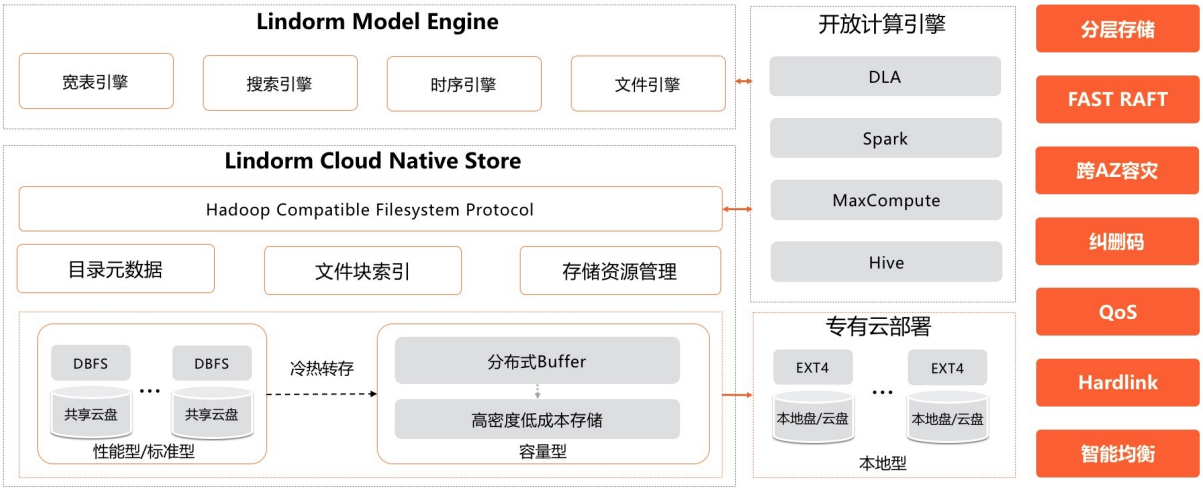


总体架构

Lindorm创新性地使用存储计算分离、多模共享融合的云原生架构，以适应云计算时代资源解耦和弹性伸缩的诉求。其中云原生存储引擎LindormStore为统一的存储底座，向上构建各个垂直专用的多模引擎，包括宽表引擎、时序引擎、搜索引擎、文件引擎。在多模引擎之上，Lindorm既提供统一的SQL访问，支持跨模型的联合查询，又提供多个开源标准接口（HBase/Phoenix/Cassandra、OpenTSDB、Solr、HDFS），满足存量业务无缝迁移的需求。最后，统一的数据Stream总线负责引擎之间的数据流转和数据变更的实时捕获，以实现数据迁移、实时订阅、数湖转存、数仓回流、单元化多活、备份恢复等能力。

存储引擎

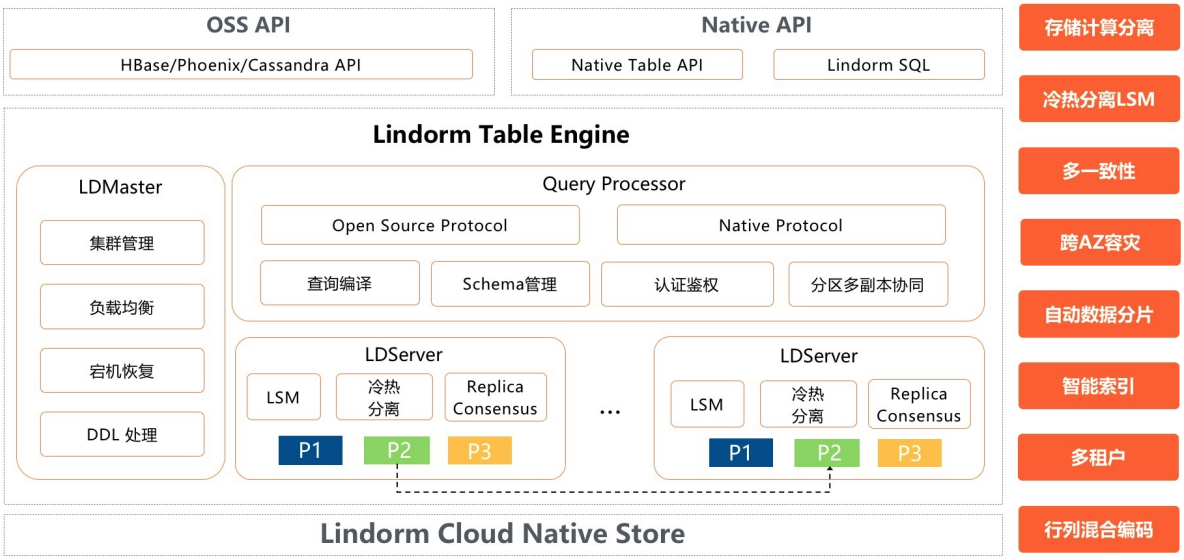
LindormStore是面向云基础存储设施设计、兼容HDFS协议的分布式存储系统，并同时支持运行在本地盘环境，以满足部分专有云、专属大客户的需求，向多模引擎和外部计算系统提供统一的、与环境无关的标准接口，整体架构如下：



LindormStore提供性能型、标准型、容量型等多种规格，并且面对真实场景的数据冷热特点，支持性能型/标准型、容量型多种存储混合使用的形态，以适应真实场景的数据冷热特点，可结合多模引擎的冷热分离能力，实现冷热存储空间的自由配比，让用户的海量数据进一步享受云计算的低成本红利。

宽表引擎

LindormTable是面向海量半结构化、结构化数据设计的分布式NoSQL系统，适用于元数据、订单、账单、画像、社交、feed流、日志等场景，兼容HBase、Phoenix(SQL)、Cassandra等开源标准接口。其基于数据自动分区+分区多副本+LSM的架构思想，具备全局二级索引、多维检索、动态列、TTL等查询处理能力，支持单表百万亿行规模、高并发、毫秒级响应、跨机房强一致容灾，高效满足业务大规模数据的在线存储与查询需求。面向海量半结构化、结构化数据设计的分布式NoSQL系统，能够兼容HBase、Phoenix(SQL)、Cassandra等开源标准接口。整体架构如下：



LindormTable的数据持久化存储在LStore中，表的数据通过自动Sharding分散到集群中的多台服务器上，并且每一个分区可以拥有1至N个副本，这N个副本拥有主、从两种角色，主从副本可以加载在不同的Zone，从而保障集群的高可用和强一致。针对不同的一致性模式，主从副本之间的数据同步和读写模式如下：

- 强一致模式。只有主副本提供读写，数据会异步回放到从副本，主副本所在节点故障，从副本晋升为主副本。晋升之前会保障数据同步完成，从副本拥有所有最新数据，整体过程由Master协调负责。
- 最终一致模式。主从副本都提供读写，数据会相互同步，保证副本之间的数据最终一致。

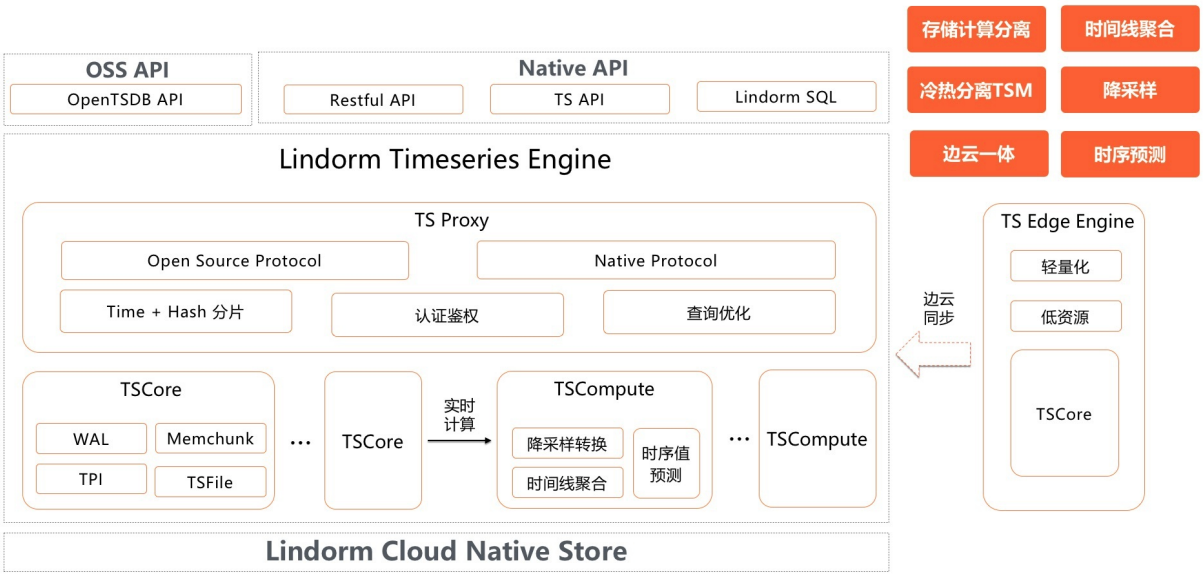
LindormTable的多副本架构基于PACELC理念设计，每一个数据表都支持单独支持设置一致性模式，从而拥有不同的可用性和性能。在最终一致模式下，服务端会对每一个读写请求在一定条件下触发多副本并发访问，从而大幅提升请求的成功率和减少响应毛刺。该并发机制建立在内部的异步访问框架上，相比于启动多线程，额外资源消耗可以忽略不计。对于并发访问的触发条件，主要包括两个类型：

- 限时触发，对于每一个请求，都可以单独设置一个GlitchTimeout，当请求运行时间超过该值未得到响应后，则并发一个请求到其他N-1个副本，最终取最快的那个响应。
- 黑名单规避，服务端内部会基于超时、抛错、检测等机制，主动拉黑存在慢、停止响应等问题的副本，使得请求能够主动绕开受软硬件缺陷的节点，让服务最大可能保持平滑。比如在掉电断开的场景下，在节点不可服务至失去网络心跳往往会存在一两分钟的延迟，利用LTable的这种多副本协同设计，可以大幅提升服务的可用性。

LindormTable的LSM结构面向冷热分离设计，支持用户的数据表在引擎内自动进行冷热分层，并保持透明查询，其底层结合LStore的冷热存储混合管理能力，大幅降低海量数据的总体存储成本。
LindormTable提供的数据模型是一种支持数据类型的松散表结构。相比于传统关系模型，LindormTable除了支持预定义字段类型外，还可以随时动态添加列，而无需提前发起DDL变更，以适应大数据灵活多变的特点。同时，LindormTable支持全局二级索引、倒排索引，系统会自动根据查询条件选择最合适的索引，加速条件组合查询，特别适合如画像、账单场景海量数据的查询需求。

时序引擎

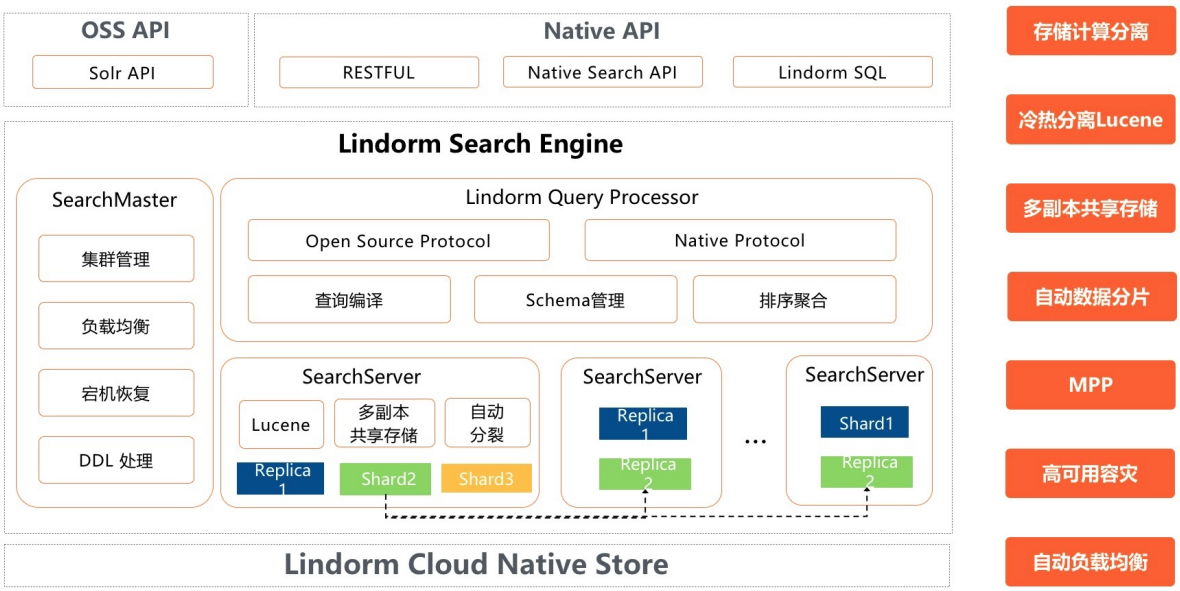
LindormTS是面向海量时序数据设计的分布式时序引擎，兼容开源OpenTSDB标准接口，其基于时序数据特点和查询方式，采用Timerange+hash结合的分区算法，时序专向优化的LSM架构和文件结构，支持海量时序数据的低成本存储、预降采样、聚合计算、高可用容灾等，高效满足IoT/监控等场景的测量数据、设备运行数据的存储处理需求，整体架构如下：



TSCore是时序引擎中负责数据组织的核心部分，其整体思想与LSM结构相似，数据先写入Memchunk，然后Flush到磁盘，但由于时序数据天然的顺序写入特征，定向专用的时序文件TSFile的结构设计为以时间窗口进行切片，数据在物理和逻辑上均按时间分层，从而大幅减少Compaction的IO放大，并使得数据的TTL、冷热分离等实现非常高效。
TSCompute是负责时序数据实时计算的组件，重点解决监控领域常见的降采样转换和时间线聚合需求，通过Lindorm Stream进行数据订阅，并完全基于内存计算，所以，整体非常的轻量、高效，适合系统已预置的计算功能。针对部分灵活复杂的分析需求，用户仍可以通过对接Spark、Flink等系统实现，从而支撑更多场景和适应业务变化。

搜索引擎

LindormSearch是面向海量数据设计的分布式搜索引擎，兼容开源Solr标准接口，同时可无缝作为宽表、时序引擎的索引存储，加速检索查询。其整体架构与宽表引擎一致，基于数据自动分区+分区多副本+Lucene的结构设计，具备全文检索、聚合计算、复杂多维查询等能力，支持水平扩展、一写多读、跨机房容灾、TTL等，满足海量数据下的高效检索需求，具体如下：



LindormSearch的数据持久化存储在LindormStore中，通过自动Sharding的方式分散到多台SearchServer中，每一个分片拥有多个副本，支持一写多读，提升查询聚合的效率，同时这些副本之间共享存储，有效消除副本之间的存储冗余。
在Lindorm系统中，**LindormSearch**既可以作为一种独立的模型，提供半结构化、非结构化数据的松散文档视图，适用于日志数据分析、内容全文检索；也可以作为宽表引擎、时序引擎的索引存储，对用户保持透明，即宽表/时序中的部分字段通过内部的数据链路自动同步搜索引擎，而数据的模型及读写访问对用户保持统一，用户无需关心搜索引擎的存在，跨引擎之间的数据关联、一致性、查询聚合、生命周期等工作全部由系统内部协同处理，用简单透明的方式发挥多模融合的价值。

文件引擎

LindormFile是基于LindormStore轻量封装的分布式文件模型服务，其核心是提供安全认证、限流保护、多网络访问等服务化能力，从而使得外部系统可以直接访问多模引擎的底层文件，大幅提升备份归档、计算分析等场景的效率。同时，用户可以在离线计算系统直接生成底层数据格式的物理文件，通过LindormFile入口把物理文件快速导入到LindormStore中，以减少对在线服务的影响。对于部分存储计算的混合场景，用户可以将计算前的源文件存在LindormFile，计算后的数据结果存在LindormTable，结合Spark、DLA等大规模计算引擎，实现简单高效的数据湖分析能力。

计算引擎

计算引擎是基于云原生架构提供的分布式计算服务，计算节点运行在阿里云Serverless Kubernetes（简称ASK）容器服务中。计算引擎支持社区版计算模型以及编程接口，同时深度融合Lindorm存储引擎特性，充分利用底层数据存储特征以及索引能力，高效地完成分布式作业任务。在数据生产、交互式分析和机器学习等场景中，提供高性能计算服务。Spark作业执行中，计算引擎提供了作业管理接口，您可以通过Spark Web UI（简称SparkUI）界面对Spark作业进行完整的监控运维。

3.7. 产品优势

3.7.1. Lindorm VS HBase VS Cassandra

Lindorm宽表引擎是面向海量半结构化、结构化数据设计的分布式存储，兼容HBase、Phoenix(SQL)、Cassandra等开源标准接口，下文将对它们之间的异同做一个对比，以便用户有更好的了解。

特性		云原生多模数据库Lindorm	开源HBase	开源Cassandra
	数据模型	支持宽表、时序、搜索、文件等多种数据模型，并且宽表支持多端、多API。	仅宽表	仅宽表
	访问API	包括HBase API、Cassandra CQL、Phoenix SQL，并且多端数据互通。	HBase API或Phoenix SQL	Cassandra CQL
	SQL	JDBC标准，兼容Phoenix，具备更好的稳定性与性能。	依赖外部Phoenix支持	简单SQL方言
	数据类型	丰富。	只支持byte[]	丰富

核心功能特性		云原生多模数据库Lindorm	开源HBase	开源Cassandra
	TTL	企业级TTL，支持表、行、Cell等多种粒度。	支持表和Cell级	只支持表级
	强一致	支持强一致、最终一致等多一致性等级。	支持	不支持
	全局二级索引	内置，查询透明、高性能、按需冗余非索引列。	依赖外部组件，复杂	支持，但数据非强一致
	多维检索	与搜索引擎LindormSearch智能集成，支持海量数据的存储、多维查询、全文检索等统一访问能力，详情请参见特性介绍。	不支持	不支持
性能	吞吐性能	单机吞吐是开源HBase的7倍，详情请参见测试结果分析。	无	无
	请求毛刺	P99延迟是开源HBase的1/10，详情请参见测试结果分析。	无	无
成本	存储成本	支持性能型、标准型、容量型等多种存储规格，成本比云盘自建低80%。	基于云盘、本地盘自建，成本高且不弹性	基于云盘、本地盘自建，成本高且不弹性
	存计分离	是，存储和计算分别支持伸缩。	否	否
	数据压缩	内置深度优化的压缩算法，数据压缩率高达10:1以上，相比snappy提高50%以上。	支持snappy/LZ4/LZO，压缩率不高	支持snappy/LZ4，压缩率不高
	编码	面向数据类型的自适应编码，压缩率高，并且无需解码，即可快速查找。	支持DIFF，压缩效果一般，并且编码后的数据无法检索	无
	冷热分离	冷热数据自动分层，其中冷数据使用高压缩和廉价存储，减少80%成本，热数据可提升访问性能15%，详情请参见冷热分离。	不支持	不支持
扩展性与弹性	最小规模	1个节点，并且提供Serverless模式，最小规模为0。	至少3个节点	至少3个节点
	扩展性	强，支持水平伸缩至几千节点。	强，支持水平伸缩至几千节点	中，支持水平伸缩，百个节点往上有瓶颈
	弹性	通过Serverless，实现自适应伸缩，按需即时弹性。	中，手动扩容，分钟级	弱，扩容需要搬迁数据，小时级
可靠性	主备双活	成熟，支持自动容灾切换、双集群请求并发等高级能力，支持与自建HBase/Cassandra构建混合主备。	无优化，不支持切换	支持，但需要三副本
	跨机房强一致	跨机房部署，支持机房级故障的自动恢复，并保证数据的强一致。	不支持	不支持
	备份恢复	支持100TB+规模的数据备份至OSS，并提供与规模无关的RTO（小于30分钟）、按需备份、指定时间点恢复等高级能力，详情请参见开通备份恢复。	支持，能力弱	支持，能力弱
	全球多活	支持，全球多地多单元部署，数据按需同步。	不支持	支持，能力一般
多租户与安全	认证与ACL	支持易用的账号密码认证+ACL，使用请参见管理用户。	不支持	支持
	资源隔离	提供Resource Group特性，支持租户间的资源物理隔离。	不支持	不支持
	Quota	支持租户的全局Quota，包括请求、存储等。	只支持单Server	不支持

安全特性		云原生多模数据库Lindorm	开源HBase	开源Cassandra
	静态加密	支持，密匙KMS托管，数据和Log全加密。	支持，较弱	不支持
	RPC黑名单	支持，可限制RPC调用。	不支持	不支持
	审计	计划中。	不支持	不支持
高级特性	Feed流API	提供社交IM场景专属的FeedStream功能，开发效率和运行性能提升一个数量级。	不支持	不支持
	表回收站	数据表被删除后进入回收站，支持找回，防止误删。	不支持	不支持
	级联Split	Region可以连续Split，无需等待Compaction，可大幅提升扩展和负载均衡能力。	不支持	不支持
	离散TTL	支持保留多个时间区段的数据。	不支持	不支持
运维诊断	运维工具	界面化集群管理工具，支持表，Namespace，Group，ACL等管理，请参见登录集群管理系统。	HBase Shell	黑屏工具
	数据查询	集群管理系统内支持图形化SQL交互查询，请参见数据查询，也支持使用开源工具HBase Shell/CQLsh。	HBase Shell	CQLsh
生态体系	数据搬迁	支持与HBase/Cassandra各个版本之间的在线、跨版本、自动化、高效搬迁，应用零影响、零改造，请参见LTS（原BDS）服务介绍。	只能离线迁移	只能离线迁移
	MySQL数据同步	通过LTS（原BDS）服务介绍，支持MySQL数据到Lindorm的全量导入和增量同步。	自己用工具，不支持在线增量	自己用工具，不支持在线增量
	Spark分析	产品化深度集成，支持Spark SQL分析Lindorm、Lindorm数据增量同步到Spark，离线分析结果回流到Lindorm等。	无优化，数据集成需要较大开发	无优化，数据集成需要较大开发
	MaxCompute	产品化集成，支持Lindorm数据增量归档到MC。	数据集成需要较大开发	数据集成需要较大开发
	日志服务（SLS）	通过LTS（原BDS）服务介绍，支持实时订阅SLS数据到Lindorm。	数据集成需要较大开发	数据集成需要较大开发
服务能力	可用性SLA	提供SLA保障，单集群99.9%，双集群高可用99.99%。	无	无
	运维成本	全托管，无需复杂的数据库运维投入。	无	无
	技术团队	由多名Apache社区PMC和Committer组成的专家队伍提供技术服务支持。	无	无
	实践经验	支持9年天猫双十一，阿里部署上万台。	无	无

3.7.2. Lindorm VS OpenTSDB

云原生多模数据库Lindorm时序引擎是一款高性能、低成本、稳定可靠的在线时序数据库引擎服务，提供高效读写、高压缩比存储、时序数据聚合计算等能力。时序引擎高度兼容OpenTSDB协议，采用自研的索引，数据模型，流式聚合等技术手段提供更强大的时序能力。本文从多方面介绍Lindorm时序引擎和OpenTSDB的区别，方便您了解和使用。

特性		Lindorm时序引擎	OpenTSDB
	服务可用性	99.9%	需自行保障，自行搭建集群，自建组件依赖。

特性		Lindorm时序引擎	OpenTSDB
运维管控	数据可靠性	99.9999%	需自行保障，自行搭建集群，自建组件依赖。
	软硬件投入	无软硬件投入，按需付费。	数据库服务器成本相对较高。
	维护成本	托管服务	需招聘专职TSDB DBA人员来维护，人力成本高。
	部署扩容	即时开通，快速部署，弹性扩容。	需硬件采购、机房托管、机器部署等工作，周期较长。
	依赖组件	零运维	依赖AysncHBase、HBase等，运维成本高。
	配置调优参数	默认参数采用最佳实践	SALT、连接数，同步刷盘参数，Compaction等等。
	建表语句	建表语句托管，用户透明。	需要运维人员静态建表语句。
	监控报警体系	完整的自监控链路	依赖外部搭建
功能	数据模型	同时支持多值模型和单值模型	仅支持单值模型
	SDK	Java SDK	开源SDK不支持查询
	数据类型多样性	支持数值、布尔、字符串等多种数据类型。	支持数值类型
	SQL查询能力	支持SQL的分析查询	不支持
	管理控制台	支持丰富的详情展示、数据管理、时序洞察等。	内置简单的图形展示
	中文支持	支持英文字符和中文字符	仅支持英文字符
	单一维度（tags可选择）	tags是可选参数	tags是必选参数
	TagKey个数	可支持16个	最多8个
	集成能力	同Flink，物联网平台无缝对接，生态丰富。	开源产品，与云产品集成能力弱。
成本	数据压缩	时序领域专用压缩，压缩率高。	通用压缩，压缩率低。
稳定性	数据读取	读写线程池分离，易于管理连接，读写稳定。	读写耦合，容易造成连接数耗尽，读写失败概率大。
	聚合器	流式聚合，内存管理粒度细，可控性强。	内存物化聚合，容易造内存OOM。

3.7.3. Lindorm VS ElasticSearch VS Solr

云原生多模数据库Lindorm搜索引擎是面向海量数据设计的分布式搜索存储，兼容开源Solr标准接口，本文从多方面介绍Lindorm搜索引擎和ElasticSearch的区别，方便您更好的了解和使用。

特性		Lindorm搜索引擎	开源ElasticSearch	开源Solr
	数据模型	支持宽表、时序、搜索、文件等多种，并且搜索可以无缝作为其他引擎的索引存储。	仅搜索	仅搜索
	访问API	包括Cassandra CQL、Phoenix SQL、Solr API。	ES API	Solr API

特性		Lindorm搜索引擎	开源ElasticSearch	开源Solr
	TTL	企业级TTL，支持表、行等多种粒度。	只支持表级	只支持表级
	存储检索统一访问	与Lindorm宽表、时序引擎无缝融合，形成多模统一存储检索能力。	N/A	N/A
性能成本	吞吐性能	单机吞吐是开源Solr的130%~200%。	N/A	N/A
	存储成本	支持性能型、标准型、容量型等多种存储规格，最低成本比云盘自建低80%。	基于云盘、本地盘自建，成本高且不弹性。	基于云盘、本地盘自建，成本高且不弹性。
	存计分离	是，存储和计算各自独立伸缩。	否	否
	数据压缩	内置深度优化的压缩算法，数据压缩率高达10:1以上，相比snappy提高50%以上。	N/A	N/A
	冷热分离	基于时间属性，数据自动分表，其中冷数据使用高压缩和廉价存储，减少成本，热数据提升访问性能。	不支持	不支持
弹性	弹性	强，存计分离，一键扩容，存储秒级生效，计算分钟级生效。	弱，扩容需要搬迁数据，小时级。	弱，扩容需要搬迁数据，小时级。
	一写多读	数据分片支持一写多读，读副本水平在线扩展，秒级生效。	支持，但增加读副本需要搬数据，小时级生效。	支持，但增加读副本需要搬数据，小时级生效。
生态体系	数据搬迁	支持Solr/ES集群数据的在线、自动化、高效搬迁到Lindorm，应用零影响、零改造，请参见LTS（原BDS）服务介绍。	只能离线迁移	只能离线迁移
	MySQL数据同步	通过LTS（原BDS）服务介绍，支持MySQL数据到Lindorm的全量导入和增量同步。	自己用工具，不支持在线增量。	自己用工具，不支持在线增量。
	Spark分析	产品化深度集成，支持Spark SQL分析Lindorm、Lindorm数据增量同步到Spark，离线分析结果回流到Lindorm等。	无优化，数据集成需要较大开发。	无优化，数据集成需要较大开发。
	日志服务	通过LTS（原BDS）服务介绍，支持实时订阅SLS数据到Lindorm。	数据集成需要较大开发	数据集成需要较大开发
服务能力	可用性SLA	提供SLA保障，单集群99.9%，双集群高可用99.99%。	N/A	N/A
	运维成本	全托管，无需复杂的数据库运维投入。	N/A	N/A
	技术团队	由多名Apache社区PMC&Committer组成的专家队伍提供技术服务支持。	N/A	N/A
	实践经验	支持9年天猫双十一，阿里部署上万台。	N/A	N/A

3.7.4. Lindorm VS HDFS

云原生多模数据库Lindorm文件引擎是一个云原生的文件存储服务，兼容开源HDFS协议，本文介绍Lindorm文件引擎和开源HDFS的区别，方便您了解和使用。

特性		Lindorm文件引擎	自建HDFS
功能定位		分布式文件系统	分布式文件系统
HDFS兼容	HDFS通信协议	支持	支持
	基础读写接口	完整支持	完整支持

特性		Lindorm文件引擎	自建HDFS
	高级管理接口	完整支持	完整支持
成本	存储单价（以实际为准）	**最低0.11元/GB/月**	最低单价0.15元/GB/月
	弹性	**在线平滑伸缩**	起步门槛高，扩容步长大。
	计算分离	**支持，与计算引擎分离，独自伸缩。**	不支持，与计算引擎混合部署。
	冷热存储	**多级存储，智能转存。**	不支持
	透明压缩	支持	不支持
扩展性	节点数	N/A	0~1000
	存储量	**0~1 EB**	0~10 PB
	文件数	**千亿级**	千万级
性能吞吐		高	高
生态		开源大数据生态Hadoop/Spark等、阿里云数据生态。	开源大数据生态Hadoop/Spark等
易用性		**免运维，维护简单。**	有状态服务，维护较复杂。

3.8. 常见问题

3.8.1. Lindorm连接问题与解决方法

本文介绍云原生多模数据库Lindorm连接问题与解决方法。
在ECS实例上通过内网地址访问Lindorm实例需要确保ECS与Lindorm之间可以相互连接。如果二者无法互访，可能是下列原因引起的。

问题原因	解决方法
Lindorm的白名单中未加入ECS的内网地址。	将ECS的内网IP添加至Lindorm的白名单，具体操作方法请参见设置白名单。
ECS的安全组规则阻塞了对Lindorm内网地址和端口的访问。	添加安全组规则，使其允许访问，具体操作方法请参见添加安全组规则。
ECS与Lindorm的网络类型不同。	网络类型不同有以下几种情况： • 经典网络下无法访问VPC环境。 • 同一个VPC网络下跨交换机，请添加安全组规则和网络ACL规则，具体操作方法请参见添加安全组规则和添加网络ACL规则。 • 不同VPC网络下默认是无法访问，请配置高速通道。更多信息，请参见如何选择私网类产品？。 • 非阿里云环境访问。例如使用您公司内网环境，请开通公网访问或通过专线接入。

如果您排查以上原因后仍然无法连接Lindorm实例，请提交工单。

3.8.2. 使用ping命令检测ECS与Lindorm之间的连接

新建Lindorm实例后，或者使用中的Lindorm实例突然无法连接时，请检查ECS与Lindorm之间的连接。您可以先在ECS的命令行中使用ping命令检测ECS与Lindorm是否能够连通。

操作步骤

1. 查看连接地址。
2. 登录ECS系统并在命令行中使用如下命令。

```
ping <host>
```

> **说明**
> - 命令中的 `<host>` 为步骤1查询的连接地址。
> - Windows系统和Linux系统中都可以使用该命令。

3. 查看测试结果，连接地址以Lindorm时序引擎地址为例。
   - Windows系统会在4次ping之后显示测试结果，如下图。

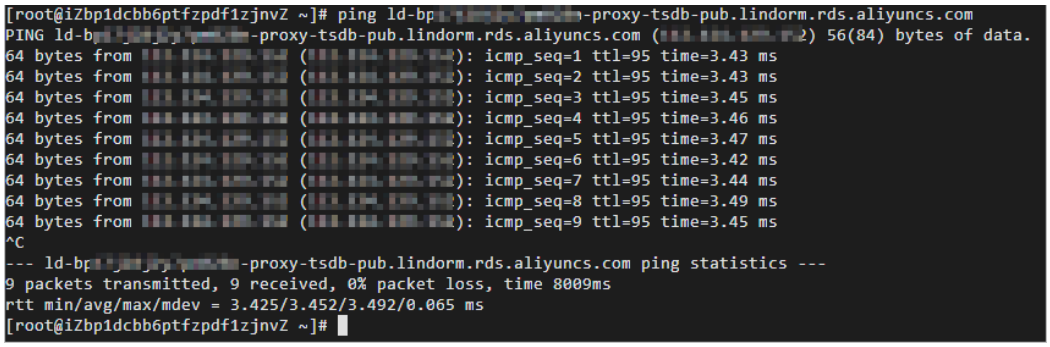
     > **说明** 如果需要持续检测连通性请使用 `ping <host> -t` 命令。

     Windows服务器执行ping命令



   - Linux系统执行该命令后将会持续发送ping请求，您可以按下键盘上的Ctrl键加C键停止执行并统计结果，示例如下。

     Linux服务器执行ping命令



结果分析

- 如果所有请求都成功返回结果（参见上方示例图），则连接正常。
- 如果未收到正常回复，则连接异常，建议参照Lindorm连接问题与解决方法检查ECS与Lindorm的设置（例如白名单或者安全组）。

3.8.3. 使用telnet命令检测Lindorm端口连通性

本文介绍如何使用telnet命令检测Lindorm端口的连通性。

前提条件

ECS系统中已经安装了Telnet（Linux）或开启了Telnet客户端（Windows）。

背景信息

如果Lindorm服务出现了连接问题，并且使用ping命令检测ECS与Lindorm之间的连接成功，您需要进一步使用telnet命令检测服务端口是否可用。

操作步骤

1. 查看连接地址。
2. 登录ECS系统并在命令行中使用如下命令。

```
telnet <host> <port>
```

> **说明**
> - 命令中的 `<host>` 为步骤1查询的连接地址，`<port>` 为连接地址的端口号。
> - Windows系统和Linux系统中都可以使用该命令。

3. 查看测试结果，连接地址以Lindorm时序引擎地址为例。

   Linux系统telnet成功示例



结果分析

如果Lindorm连接存在问题，但可以在ECS上使用telnet连接到Lindorm实例，则ECS本身与Lindorm之间的连接无异常，请排查其它因素，例如客户端、业务代码，以及业务环境导致的Lindorm服务阻塞等问题。您可以参见Lindorm连接问题与解决方法以获得更多帮助信息。

3.8.4. Lindorm连接出现报错与解决方法

本文介绍云原生多模数据库Lindorm连接时出现报错与解决方法。
如果客户端出现类似如下报错，解决方法请参见下表。

报错类型	导致报错的可能原因与解决方法
Retry exhausted when update config from seedserver	• 如果通过ECS访问Lindorm实例，确保ECS与Lindorm之间网络可以相互连接，具体排查方法请参见Lindorm连接问题与解决方法。 • 可能Lindorm实例的连接地址端口使用错误，例如HBase兼容客户端的连接地址端口号为30020。
Failed to connect to jdbc:lindorm:table:url=****	• 如果通过ECS访问Lindorm实例，确保ECS与Lindorm之间网络可以相互连接，具体排查方法请参见Lindorm连接问题与解决方法。 • 可能Lindorm实例的连接地址端口使用错误，例如Lindorm宽表SQL的连接地址端口号为30060，获取方法请参见获取连接地址。
DoNotRetryIOException: Detect inefficient query: SELECT*FROM xxxx This query may be a fulltable scan and thus may have unpredictable performance.	执行SELECT语句被Lindorm宽表引擎识别为低效查询，所以服务器拒绝执行，解决方法请参见常见问题。

3.9. 技术支持

关于云原生多模数据库Lindorm产品和服务的任何问题，请您通过以下渠道寻求支持和解决。

支持方式

- 提交工单
  i. 登录工单系统。
  ii. 在搜索栏中，输入**云数据库HBase**。
  iii. 选择对应的问题分类，并单击**创建工单**。
- 钉钉群
  您可以使用钉钉搜索钉钉群号35977898加入钉钉群进行反馈。

> **说明** 此钉钉群仅用于用户之间的交流，如果您有业务问题或者产品疑难问题请提交工单。

3.10. 引擎类型

云原生多模数据库Lindorm内含宽表引擎、时序引擎、搜索引擎、文件引擎、计算引擎和流引擎，兼容HBase/Cassandra/S3、OpenTSDB、Solr、HDFS、Kafka等多种开源标准接口，同时提供SQL查询、时序处理、文本检索分析等能力。
为应对动态变化的业务访问量，各引擎支持独立按需弹性扩展，宽表引擎和时序引擎均提供高并发吞吐能力。

引擎选择

不同引擎类型适用不同的场景，可按需选择一种或多种引擎，具体介绍请参见下表。

引擎类型	兼容接口	适用场景	介绍
宽表引擎	兼容SQL、HBase API、Cassandra CQL、S3 API	元数据、订单、账单、画像、社交、feed流、日志等场景。	面向海量半结构化、结构化数据设计的分布式宽表引擎，具备全局二级索引、多维检索、动态列、TTL等能力，支持千万级高并发吞吐，支持百PB级存储，吞吐性能是开源HBase的3-7倍，P99时延为开源HBase的1/10，支持冷热分离，压缩率比开源HBase提升一倍，综合存储成本为开源HBase的1/2。
时序引擎	提供HTTP API接口，并兼容OpenTSDB API	IoT、监控等场景存储、处理量测数据、设备运行数据等时序数据。	面向海量时序数据设计的分布式时序引擎，支持SQL查询。针对时序数据设计的压缩算法，提高压缩率。支持海量多维的时间线查询和时间线聚合，支持降采样。支持弹性扩展。
搜索引擎	兼容SQL、Solr接口	海量日志、文本、文档等数据，适用于日志、账单、画像等场景。	采用存储和计算分离架构设计的分布式搜索引擎，可无缝作为宽表、时序引擎的索引存储，加速检索查询，具备全文检索、聚合计算、复杂多维查询等能力，支持水平扩展、一写多读、跨机房容灾、TTL等，满足海量数据下的高效检索需求。
文件引擎	兼容HDFS接口	企业级数据湖存储、Hadoop平台存储底座、历史数据归档压缩等场景。	云原生HDFS。通信协议级兼容HDFS，可使用开源HDFS客户端直接访问，功能100%兼容HDFS标准，无缝接入所有HDFS开源生态与云计算生态。基于HDFS深度定制开发，具备低成本、EB级数据存储、分钟级存储弹性扩容、带宽水平弹性扩展等能力，适合构建基于HDFS的企业级低成本数据湖存储、通过存储计算分离降低总体成本。
计算引擎	兼容Spark接口	海量数据生产、交互式分析、计算学习和图计算等场景。	计算引擎是基于云原生架构提供的分布式计算服务，支持社区版计算模型以及编程接口，同时深度融合Lindorm存储引擎特性，利用底层数据存储特征以及索引能力，高效地完成分布式作业任务。
流引擎	兼容SQL、Kafka接口	IoT数据处理、应用日志处理、物流时效分析、出行数据处理等场景。	流引擎是面向流式数据处理的引擎，提供了流式数据的存储和轻计算功能，帮助您轻松实现将流式数据存储至云原生多模数据库Lindorm，满足基于流式数据的处理和应用需求。

3.11. 存储类型

云原生多模数据库Lindorm依赖于LindormStore云原生存储系统，实现了数据存储与计算分离解耦。存储容量独立计费，支持不停机在线扩容。Lindorm实例的存储容量在同实例内多个引擎间共享。

存储类型说明

云原生多模数据库Lindorm目前提供了三种主要的存储类型：容量型存储、标准型存储、性能型存储。其适应场景如下：

存储类型	访问延迟	适用场景	推荐引擎

存储类型	访问延迟	适用场景	推荐引擎
容量型存储	15ms ~ 3s	监控日志、历史订单、音视频归档、数据湖存储、离线计算等低频访问数据。 **说明** 容量型存储使用高密度磁盘存储阵列，提供极低成本存储能力，和高吞吐读写能力，但随机读能力较弱，适用于写多读少场景或大数据计算场景。	宽表引擎、时序引擎、文件引擎
标准型存储	3ms ~ 5ms	Feed流数据、聊天、实时报表、在线计算等实时访问数据。	宽表引擎、时序引擎、搜索引擎、文件引擎
性能型存储	0.2ms ~ 0.5ms	广告竞价投放、用户画像、人群圈选、实时搜索、风控大脑等低延迟访问数据。	宽表引擎、时序引擎、搜索引擎、文件引擎

说明 访问延迟仅表示存储访问延迟，不代表端到端访问延迟。

3.12. 基本概念

本文介绍云原生多模数据库Lindorm帮助文档中的相关名词和术语的解释。

A

安全组

安全组是一种虚拟防火墙，用于控制安全组中的ECS实例的出入流量。在Lindorm实例的白名单中添加安全组后，该安全组中的ECS实例就可以访问Lindorm实例。更多信息，请参见添加安全组。

按量付费

后付费，按小时扣费。适合短期业务需求，用完可以立即释放实例，节省费用。更多信息，请参见计费方式。

B

包年包月

预付费，在新建实例时需要支付费用。适合长期业务需求，价格比按量付费更实惠，且购买时长越长，折扣越多。更多信息，请参见计费方式。

本地SSD盘

与数据库引擎位于同一节点的SSD盘。将数据存储于本地SSD盘，可以降低I/O延时。更多信息，请参见存储类型。

C

CPU使用率

在一段时间内，系统中运行的程序、进程等所占用的CPU资源。

D

多模

云原生多模数据库Lindorm支持宽表、时序、对象、文件、队列、空间等多种数据模型，提供标准SQL和开源接口两种方式，各数据模型之间数据互融互通，使得应用开发更加敏捷、灵活和高效。

DMS

数据管理（Data Management Service，简称DMS）是一种集数据管理、结构管理、用户授权、安全审计、数据趋势、数据追踪、BI图表、性能与优化和服务器管理于一体的数据管理服务。更多信息，请参见什么是数据管理DMS。

DTS

数据传输服务（Data Transmission Service，简称DTS）支持关系型数据库（RDBMS）、非关系型数据库（NoSQL）、数据多维分析（OLAP）等数据源间的数据交互，集数据同步、迁移、订阅、集成、加工于一体，帮助您构建安全、可扩展、高可用的数据架构。更多信息，请参见什么是数据传输服务DTS。

独享规格

完全独享的CPU和内存，性能长期稳定，不会因为物理机上其它实例的行为而受到影响。独享规格的顶配是独占一台物理机的所有资源。

地域

云原生多模数据库Lindorm实例的服务器所处的地理位置。用户需要在购买Lindorm实例时指定地域，购买后不支持更改。

动态列

云原生多模数据库Lindorm宽表引擎支持的特性。动态列是业务无需预定义就可以在使用过程中根据需求动态增加的列。更多信息，请参见管理动态列。

多可用区

云原生多模数据库Lindorm支持将一个实例部署在多个可用区，这种部署方案可以使实例具备更高的容灾能力。如果Lindorm宽表的一致性是最终一致，Lindorm实例会在多个可用区之间自动路由请求，从而提高业务的单可用区抗抖动能力。更多信息，请参见多可用区部署。

E

ECS

云服务器ECS（Elastic Compute Service，简称ECS）是一种简单高效、处理能力可弹性伸缩的计算服务。帮助您构建更稳定、安全的应用，提升运维效率，降低IT成本，使您更专注于核心业务创新。更多信息，请参见什么是云服务器ECS。

ESSD云盘

阿里云ESSD（Enhanced SSD）云盘结合25 GE网络和RDMA技术，为您提供单盘高达100万的随机读写能力和单路低时延性能。更多信息，请参见ESSD云盘。

二级索引

云原生多模数据库Lindorm宽表在Tabular模型下支持对表创建多个二级索引。二级索引是非主键索引，在非主键匹配的查询场景下，可以降低应用的开发复杂性、保证数据的一致性、提高写入效率。更多信息，请参见高性能原生二级索引。

G

Ganos时空引擎

阿里云推出的一款管理空间几何数据、时空轨迹的时空大数据引擎系统。该系统兼容开源GeoMesa、GeoServer等生态，内置了高效的时空索引算法、空间拓扑几何算法等，结合云原生多模数据库Lindorm宽表引擎强大的分布式存储能力以及DAL Spark分析平台，可以广泛应用于空间或时空存储、查询、分析和数据挖掘场景。更多信息，请参见Ganos时空引擎介绍。

I

I/O

Input和Output的缩写，即对磁盘进行的读写动作。

IOPS

Input/Output Operations Per Second，以4 KB为单位，每秒进行块设备读写操作的次数。

IP白名单

允许访问Lindorm实例的IP地址清单。设置IP白名单可以让Lindorm实例得到高级别的访问安全保护，建议您定期维护白名单。设置白名单的方法请参见设置白名单。

J

计算引擎

与Lindorm存储引擎深度融合，基于云原生架构提供的分布式计算服务，满足用户在数据生产、交互式分析、机器学习和图计算等场景的计算需求，兼容开源Spark标准接口。更多信息，请参见计算引擎介绍。

JDBC

Java Database Connectivity，是Java语言中用来规范客户端程序如何访问数据库的应用程序接口。

经典网络

实例之间不通过网络进行隔离，只能依靠实例自身的白名单策略来阻挡非法访问。

K

宽表引擎

负责宽表与对象数据的管理和服务，具有全局二级索引、多维检索、动态列、TTL等能力，适用于元数据、订单、账单、画像、社交、feed流、日志等场景，兼容SQL、HBase、Cassandra（CQL）、S3等标准接口。更多信息，请参见宽表引擎介绍。

宽表SQL

支持多种语言Driver，包括Java、Python、Go等，通过Lindorm宽表SQL多语言Driver可以访问Lindorm实例。更多信息，请参见Lindorm宽表SQL Driver说明。

可用区

在某个地域内拥有独立电力和网络的物理区域。同一地域的不同可用区之间没有实质性区别。

L

流引擎

面向流式数据处理的引擎，提供了流式数据的存储和轻计算功能，帮助您轻松实现流式数据存储至云原生多模数据库Lindorm，构建基于流式数据的处理和应用。更多信息，请参见流引擎介绍。

逻辑I/O

由应用程序发给文件系统的读写操作。

冷存储

一种存储介质，可以存储数据库中的冷数据，从而降低存储成本。更多信息，请参见冷存储介绍。

冷热分离

Lindorm宽表引擎实现了数据的冷热分离能力，系统会自动根据用户设置的冷热分界线自动将表中的冷、热数据分类归档到冷、热存储中。更多信息，请参见冷热分离。

LTS

LTS（Lindorm Tunnel Service）是面向Lindorm业务场景特点深度定制的数据生态服务。支持简单易用的数据交换、处理、订阅等能力，满足用户的数据迁移、实时订阅、数湖转存、数仓回流、单元化多活、备份恢复等需求，实现面向Lindorm的一站式数据生态服务。更多信息，请参见LTS（原BDS）服务介绍。

M

MaxCompute

大数据计算服务MaxCompute（原名ODPS）是一种快速、完全托管的TB、PB级数据仓库解决方案，提供了完善的数据导入方案以及多种经典的分布式计算模型，能够快速地解决海量数据计算问题。通过数据集成服务，可将Lindorm数据导入MaxCompute，实现大规模的数据计算。更多信息，请参见什么是MaxCompute。

MongoDB

提供稳定可靠、弹性伸缩、完全兼容MongoDB协议的数据库服务。数据结构多样时，可以选择将结构化数据存储在Lindorm，将非结构化数据存储在MongoDB，满足业务的多样化存储需求。更多信息，请参见什么是云数据库MongoDB版。

O

OSS

对象存储服务（Object Storage Service，简称OSS）是阿里云提供的海量、安全、低成本、高可靠的云存储服务。更多信息，请参见什么是对象存储OSS。

OpenAPI

应用编程接口（Open Application Programming Interface，简称OpenAPI）是云原生多模数据库Lindorm开放的应用编程接口，提供给第三方开发者使用。Lindorm提供的OpenAPI请参见API概览。

Q

QPS

数据库每秒执行的SQL数（Queries Per Second，简称QPS），包含INSERT、SELECT、UPDATE、DETELE、COMMIT等。

R

RAM

访问控制RAM（Resource Access Management）是阿里云提供的管理用户身份与资源访问权限的服务。更多信息，请参见什么是访问控制。

RDS

阿里云关系型数据库RDS（Relational Database Service）是一种稳定可靠、可弹性伸缩的在线数据库服务。基于阿里云分布式文件系统和SSD盘高性能存储，RDS支持MySQL、SQL Server、PostgreSQL和MariaDB TX引擎，并且提供了容灾、备份、恢复、监控、迁移等方面的全套解决方案，帮助用户解决数据库运维的烦恼。更多信息，请参见云数据库RDS简介。

Redis

云数据库Redis版提供持久化的内存数据库服务。当业务访问量较大时，ECS 、Lindorm和Redis的组合可以支持更多的读请求，同时减少响应时间。更多信息，请参见什么是云数据库Redis版。

S

时序引擎

面向海量时序数据设计的分布式时序引擎，支持SQL查询和压缩算法、支持海量多维的时间线查询和时间线聚合、降采样、弹性扩展等功能。更多信息，请参见时序引擎介绍。

搜索引擎

负责多模数据的检索分析加速，基于列存、倒排等核心技术，具备全文检索、聚合计算、复杂多维查询等能力，适用于日志、账单、画像等场景，兼容SQL、开源Solr等标准接口。更多信息，请参见搜索引擎介绍。

搜索索引

云原生多模数据库Lindorm宽表引擎除了支持高性能原生二级索引，也支持一种新的索引类型，称为搜索索引（SearchIndex），搜索索引主要面向复杂的多维查询场景，能够覆盖分词、模糊查询、聚合分析、排序翻页等场景。更多信息，请参见搜索索引介绍。

SSL

SSL（Secure Sockets Layer）是Netscape公司所提出的安全保密协议，在浏览器和Web服务器之间构造安全通道来进行数据传输，采用RC4、MD5、RSA等加密算法实现安全通讯。国际互联网工程任务组（IETF）对SSL 3.0进行了标准化，标准化后更名为安全传输层协议（TLS）。

删除保护

删除保护可以防止您在操作过程中将承载有重要业务的实例误释放。开启实例删除保护后，将无法释放该实例，如果需要释放，需要先关闭实例删除保护，然后释放实例。开启删除保护的方法请参见开启或关闭删除保护。

顺序I/O

按照顺序向磁盘进行读写操作。

随机I/O

随机向磁盘进行读写操作。

T

TDE

透明数据加密（Transparent Data Encryption，简称TDE），云原生多模数据库Lindorm支持透明数据加密功能，即对数据文件执行实时I/O加密和解密。数据在写入磁盘之前进行加密，从磁盘读入内存时进行解密。TDE不会增加数据文件的大小，开发人员无需更改任何应用程序，即可使用TDE功能。更多信息，请参见开启透明数据加密。

TPS

数据库每秒执行的事务数（Transactions Per Second，简称TPS）。

V

VPC

专有网络（Virtual Private Cloud，简称VPC）是一种隔离的网络环境，安全性和性能均高于传统的经典网络。更多信息，请参见什么是专有网络。

W

文件引擎

负责目录文件数据的管理和服务，并提供宽表、时序、搜索引擎底层共享存储的服务化访问能力，从而加速多模引擎底层数据文件的导入导出及计算分析效率，兼容开源HDFS标准接口。更多信息，请参见文件引擎介绍。

物理I/O

由文件系统发给磁盘的读写操作。

X

小版本升级

云原生多模数据库Lindorm发布新的引擎小版本时，您可以通过控制台一键将实例引擎升级至最新的小版本。升级小版本可以完善老版本的功能，同时也为您提供更多的新功能。更多信息，请参见升级小版本。

4.产品计费

4.1. 计费方式

本文介绍云原生多模数据库Lindorm的收费项目和计费方式。

收费项目与计费标准

资源模式	商品类型	收费项目	计费标准
普通型	Lindorm（包年包月）	• 集群基础费 • 引擎节点规格费 • 集群存储空间费	价格请参见产品价格。 说明 具体已实际售卖为准。
	Lindorm（按量付费）		
Serverless型	LindormServerless（包年包月）	• 请求CU费 • 数据存储费	
	LindormServerless（按量付费）		

计费方式

计费方式	到期/欠费说明	续费说明
包年包月：预付费方式，在新建实例时就需要支付费用。	到期后，实例将被锁定，数据保留15天。实例锁定期间不发生收费行为。实例到期15天后会被释放，数据也会被删除且无法恢复。	在合同期内或合同到期后的15天内，可以进行续费。续费时，按照您新选择的实例配置和购买时长计费。
按量付费：后付费方式，每小时根据实例配置生成1个收费订单，并扣除相应的账户余额。	欠费的账户下所有按量付费实例都会变成欠费状态。实例欠费超过24小时会被锁定，持续欠费15天会被释放，数据也被删除且无法恢复。	按量付费实例根据实际使用时间计费，无需续费，请保障账户资金充足。

Serverless计费项

- **请求计费**

Serverless读写计量单位以CU（Capacity Unit，能力单元）来计算，读写请求都会消耗一定的CU，具体请求消耗的CU数，请参见宽表Serverless计费方式。对于包年包月模式，预购CU代表用户的请求峰值，当超过时会遇到服务端拒绝的错误。对于按量付费模式，CU按实际请求统计。

- **存储计费**

对于包年包月模式，需要提前预购存储空间，当超过后，实例将被锁定。对于按量付费模式，按实际的用户数据量统计，具体请参见宽表Serverless计费方式。

> 说明
> 目前Serverless模式只支持宽表引擎，宽表Serverless的完整计费说明，请参见Serverless计费方式。

4.2. 产品价格

创建云原生多模数据库Lindorm实例，您需要根据实例的规格支付相应的费用，在实例运行过程中，如果升级了规格或扩容了存储空间，还会产生相应的费用。

支持地域

地区	地域	Region ID（API使用）
	华东1（杭州）	cn-hangzhou
	华东2（上海）	cn-shanghai

地区	地域	Region ID（API使用）
中国	华北1（青岛）	cn-qingdao
	华北2（北京）	cn-beijing
	华北3（张家口）	cn-zhangjiakou
	华北5（呼和浩特）	cn-huhehaote
	华南1（深圳）	cn-shenzhen
	中国（香港）	cn-hongkong
亚太	日本（东京）	ap-northeast-1
	新加坡	ap-southeast-1
	澳大利亚（悉尼）	ap-southeast-2
	马来西亚（吉隆坡）	ap-southeast-3
	印度尼西亚（雅加达）	ap-southeast-5
欧洲与美洲	德国（法兰克福）	eu-central-1
	英国（伦敦）	eu-west-1
	美国（硅谷）	us-west-1
	美国（弗吉尼亚）	us-east-1

计费模式

- Lindorm经过大量的性能成本优化，相比自建开源数据库，TCO可低至50%，在性能和成本方面有很大的优势。
- 支持按量计费与包年包月两种付费模式，长时间使用选择包年包月模式更加划算。
- 测试场景可以先选择按量付费，支持通过Lindorm控制台直接操作从按量付费转包年包月模式。
- 如果有大量需求，
  请提交工单。

存储价格

> **注意** 价格在不同地域站点可能存在差别，并且可能存在调价，下表价格仅供参考，具体请以实际售卖为准。

资源模式	存储类型	包月价（元/GB/月）	小时价 （元/GB/小时）
普通	冷存储	0.12	-
普通	容量型云存储	0.12	-
普通	标准型云存储	0.7	-
普通	性能型云存储	2.0	-
Serverless	基础规格	0.36	-

计算节点价格

> **注意** 价格在不同地域站点可能存在差别，并且可能存在调价，下表价格仅供参考，具体请以实际售卖为准。

资源模式	计算规格	包月价（元/月）	小时价（元/小时）
普通	4核16GB	600	1.875
普通	8核32GB	1248	3.900
普通	16核64GB	2592	8.100
普通	32核128GB	5184	16.200
普通	本地SSD 4核32GB 894GB	1,039.60	-
普通	本地SSD 8核64GB 1788GB	2,079.20	-
普通	本地SSD 16核128GB 3576GB	4,158.40	-
普通	本地SSD 32核236GB 7152GB	8,316.80	-
Serverless	基础规格	0.15 元/CU/月	-

> **说明** 关于Serverless的计费说明，详情请参见Serverless计费项。

公网流量

> **注意** 优惠活动中，流出流量全部免费，收费时间待定。

类型	价格（元）
流入	免费。
流出	优惠活动中，流量全免（节省0.8 元/GB）。

4.3. 到期和欠费说明

云原生多模数据库Lindorm实例到期或欠费后会影响Lindorm实例正常运行，如果您需要继续使用该实例，请及时完成续费或开通自动续费。

实例类型	实例状态说明	处理方法
包年包月	实例到期后，实例处于锁定状态并无法被访问，数据保留15天。	手动为实例续费。续费方法，请参见手动续费包年包月实例。 **说明** 为避免错过续费时间而造成数据丢失，建议您开通自动续费，开通方法请参见开通自动续费。
	实例到期15天后，实例的计算资源和存储资源被释放，数据会被删除且无法找回。	无法找回。

实例类型	实例状态说明	处理方法
按量付费	• 阿里云账号欠费后，该账号下所有按量付费实例将会进入欠费状态。 • 欠费后如果在延停权益额度内，您的实例将不会停服。在此期间，您每天会收到一次短信提醒。欠费期间无法释放实例。 **说明** 阿里云提供延停权益，即当按量付费的资源发生欠费后，提供一定额度或时长继续使用云服务的权益。延停期间正常计费。延停的权益额度不是欠费总额的上限。延停的额度或时长根据您在阿里云的历史消费等因素，每个月自动计算并更新。更多信息，请参见延期免停权益。 • 超出了延停额度，阿里云会暂停该实例的服务，并停止计费。	给阿里云账号充值，实例立即恢复正常。 **说明** 如果您需要长时间使用按量付费实例，您可以将其付费类型转换为包年包月，并根据业务使用时间选择包年包月时长，为您节省开支。具体操作，请参见按量付费转包年包月。
	停服15天后，实例的计算资源和存储资源被释放，数据会被删除且无法找回。	无法找回。

4.4. 续费说明

续费是指延长包年包月实例的使用时间。按量付费实例没有到期时间，无需续费，您需要确保阿里云账号有足够的余额。

费用说明

续费的费用与购买实例的费用相同。购买实例的费用与所选地域、规格大小、存储类型等配置有关，请前往购买页面了解详情。

扣款时间

包年包月实例续费的方式包括以下两种：

- 自动续费：首次扣款时间为实例到期前第9天。如果扣款失败，系统将在接下来的每天尝试扣款。请确保阿里云账号的余额充足，避免续费失败。
- 手动续费：续费时手动支付。

相关文档

- 自动续费
- 手动续费

4.5. 变更配置计费说明

本文对云原生多模数据库Lindorm实例在变更配置时产生的费用进行详细说明。

按量付费实例变配

按量付费实例的计费周期为小时，变更实例配置后按照变配后新规格的售价进行小时计费。
示例：您在14点20分变更Lindorm实例的配置，那么实例在14点~15点的账单按照变配后新规格的售价进行计费。

包年包月实例变配

变配类型	费用说明

变配类型	费用说明
升级配置	支付费用 = 新配置剩余时长总价费用 - 老配置剩余时长总价费用 其中，新配置剩余时长总价费用 =（新配置的月单价÷30天÷24小时）× 包年包月剩余时长 × 24小时。 老配置剩余时长总价费用 =（老配置的月单价÷30天÷24小时）× 包年包月剩余时长 × 24小时。 示例：新配置月单价14400元，老配置月单价7200元，包年包月剩余天数50天，则需要支付费用 =（14400元÷30天÷24小时×50天×24小时）-（7200元÷30天÷24小时×50天×24小时）=12000元。
降级配置	关于立即降级配置的退款费用，请参见降配退款规则。

4.6. 退费说明

本文介绍云原生多模数据库Lindorm实例的退费说明和退费步骤。

退费概述

根据实例的计费类型，Lindorm实例支持的退费情况如下表所示。

计费类型	退费说明
包年包月	如果您的业务不再需要使用包年包月实例，可申请退款。退款时，系统会根据您包年包月资源的选购情况进行结算。您可以通过用户中心控制台发起退订申请，具体操作请参见退费步骤。
按量付费	按量付费的Lindorm实例根据实际使用情况实时产生账单，因此不涉及退款。如果您不再需要使用按量付费的Lindorm实例，可以释放该实例，实例释放后将不再进行计费。释放实例的方法请参见释放实例。

退费步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页面，复制需要退费的包年包月实例ID。
4. 在页面右上角，单击**费用 > 用户中心**。
5. 在左侧导航栏，选择**订单管理 > 退订管理**。
6. 在**退订管理**页面，为Lindorm包年包月实例退费。
   i. 在**退订使用中资源**页签，**实例ID**文本框中输入需要退费的包年包月实例ID，并单击**搜索**。
   ii. 单击目标实例所在行**操作**列的**退订资源**。
   iii. 在**退订资源**页面，确认目标实例ID并选择退订原因。
   iv. 阅读并勾选退订规则，单击**立即退订**。

> **说明** 如果您在退费过程中遇到问题，请提交工单。

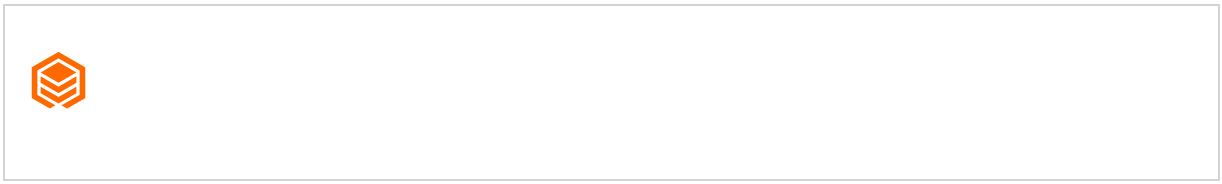
5.快速入门

如果您初次使用阿里云原生多模型数据库Lindorm，请参见快速入门系列文档。

5.1. 快速入门概览

通过本文您可以了解如何选择云原生多模数据库Lindorm实例的引擎类型和规格，并帮助您快速掌握Lindorm实例从创建、连接到使用的流程。

如何选择Lindorm实例的引擎类型与规格



选择Lindorm引擎类型与规格

在创建Lindorm实例时，您需要完全根据业务场景、访问量、访问模式、容量大小等因素进行选型。

- Lindorm引擎类型
- Lindorm，即规格选型

如何使用Lindorm实例

通常从购买Lindorm实例到可以开始使用，您需要完成如下操作：



步骤	说明
步骤1：创建实例	根据业务需求，确认实例的规格、存储等信息。
步骤2：设置白名单	提升数据库访问的安全性，将允许访问云数据库的IP地址设置为白名单，防止其他外部地址的访问。
步骤3：连接并使用实例	支持通过API、CQL、Shell等方式连接并使用Lindorm实例。

5.2. 创建实例

您可以通过云原生多模数据库Lindorm控制台创建Lindorm实例。本文介绍创建Lindorm实例的步骤，以及创建过程中各个参数的含义。

前提条件

已注册阿里云账号。具体操作请参见注册阿里云账号。

注意事项

如果应用部署在ECS实例，您需要确保云原生多模数据库Lindorm实例和ECS实例满足以下条件，以保证网络的连通性。ECS实例信息的查看方法请参见查看实例信息。

- 所在地域相同，并建议所在可用区相同（以减少网络延时）。
- 网络类型相同。

> **说明** 专有网络具有更高的安全性，建议使用专有网络。如果网络类型为专有网络，需确保使用相同的专有网络ID。

操作步骤

1. 登录Lindorm管理控制台。
2. 在**实例列表**页面，单击**创建**。

> **说明** 您也可以前往产品详情页面，单击**立即购买**创建实例。

3. 在**商品类型**区域，请根据业务需求，选择**Lindorm（包年包月）**或者**Lindorm（按量付费）**。

> **说明** 按量付费实例可以转换为包年包月实例，包年包月实例无法转换为按量付费实例。
>
> - **Lindorm（包年包月）**：预付费，在创建实例时就需要支付费用。适合长期需求，价格比按量付费更实惠，且购买时长越长，折扣越多。
> - **Lindorm（按量付费）**：后付费，每小时根据实例配置生成1个收费订单，并扣除相应的账户余额。适合短期需求，用完可立即释放实例，节省费用。

4. 设置以下参数，完成实例的参数配置。

参数	参数说明
部署方案	选择**单可用区**，如果您选择**多可用区**，更多操作请参见创建多可用区实例。 - **单可用区**：主节点和备节点位于同一可用区，适合可用区灾备级别的要求为单机房的场景。 - **多可用区**：主节点和备节点位于不同可用区，具备跨可用区容灾。
地域	选择实例所属地域。 **说明** - 实例创建成功后将无法更换地域，请您谨慎选择。 - 如果应用部署在ECS实例，请您选择与ECS实例相同的地域，否则无法直接通过内网通信。
可用区	选择实例所属可用区。 **说明** 如果应用部署在ECS实例，建议您选择与ECS实例相同的可用区，以减少网络延时。
多可用区组合	仅当**部署方案**选择为**多可用区**时会显示此参数。 **说明** 您无法选择**多可用区组合**，Lindorm支持多可用区的每个地域会默认匹配可用区组合。
网络类型	固定为**专有网络**。
专有网络（VPC）	选择实例所属专有网络。如果您没有可以使用的专有网络，您可以在专有网络控制台创建专有网络，创建方法请参见创建和管理专有网络。
专有网络交换机	选择实例所属交换机。如果您没有可以使用的交换机，您可以在专有网络控制台创建交换机，创建方法请参见创建交换机。
集群名称	设置实例的名称，便于后续业务识别。

参数	参数说明
存储类型	可选标准型云存储、性能型云存储、容量型云存储、本地SSD盘、本地HDD盘。您可以根据实际业务需求选购存储类型，具体请参见存储类型。 ○ **标准型云存储**：适用于Feed流数据、聊天、实时报表、在线计算等实时访问数据的场景。 ○ **性能型云存储**：适用于广告竞价投放、用户画像、人群圈选、实时搜索、风控大脑等低延迟访问数据的场景。 ○ **容量型云存储**：适用于监控日志、历史订单、音视频归档、数据湖存储、离线计算等低频访问数据的场景。 ○ **本地SSD盘**：本地盘来自单台物理机，具有低时延、高随机IOPS、高吞吐量、高性价比等优势。适用于网络游戏、电商、视频直播、媒体等在线业务，满足I/O密集型应用对块存储的低时延和高I/O性能需求。 ○ **本地HDD盘**：本地盘来自单台物理机，具有低时延、高随机IOPS、高吞吐量、高性价比等优势。适用于互联网行业、金融行业等有大数据计算与存储分析需求的行业，进行海量数据存储和离线计算的业务场景。
选择数据引擎	包括宽表引擎、时序引擎、搜索引擎、文件引擎和流引擎。您可以根据实际业务需求选购数据引擎的节点规格和数量，具体请参见引擎类型。 ○ **宽表引擎**：可按需选择宽表引擎节点规格和数量，以core为单位。 **说明** ■ 若存储类型选择**本地SSD盘**和**本地HDD盘**，则不支持此功能。 ■ 目前宽表引擎仅支持1：4的CPU内存配比。 ○ **时序引擎**：可按需选择时序引擎节点规格和数量，以core为单位。 **说明** ■ 若存储类型选择**本地SSD盘**和**本地HDD盘**，则不支持此功能。 ■ 目前时序引擎仅支持1：4的CPU内存配比。 ○ **搜索引擎**：可按需选择搜索引擎节点规格和数量，以core为单位。 **说明** ■ 若存储类型选择**本地SSD盘**和**本地HDD盘**，则不支持此功能。 ■ 目前搜索引擎仅支持1：4的CPU内存配比。 ○ **文件引擎**：可按需选择文件引擎节点数量，节点规格默认为**标准规格**。 **说明** 若存储类型选择**本地SSD盘**和**本地HDD盘**，则不支持此功能。
存储空间	选择实例的存储空间的大小。
是否选购冷存储	实例是否需要开通冷存储功能，更多请参见冷存储介绍。 **说明** 云存储系列实际可用存储容量为总容量。本地盘系列集群采用三副本架构，实际可用存储容量为总容量的三分之一。
冷存储存储容量	选择实例的冷存储空间大小。初次购买容量上限为100000 GB，后续可无限扩容。 **说明** 仅当**是否选购冷存储**参数选择**是**时会配置此参数。

参数	参数说明
购买时长	如果您创建的是包年包月实例，请您选择购买时长。包月可选择1~9个月，包年可选择1~3年。 **说明** 如果您希望实例到期后自动续费，勾选**到期自动续费**，并确保实例在即将到期前您的账户中有足够的余额。

5. 设置完以上参数后，单击**立即购买**。
6. 在**确认订单**页面，阅读并勾选服务协议，并单击**立即开通**。
7. 支付完成后，单击**登录控制台**，进入云原生多模数据库Lindorm控制台。

5.3. 设置白名单

为保障云原生多模数据库Lindorm的安全稳定，系统默认任何设备均无法访问Lindorm实例，所以在使用Lindorm实例前，您需要为实例设置白名单，以允许外部设备访问该实例。正确使用白名单可以使Lindorm实例得到更高级别的安全保护，建议您定期维护白名单。

准备工作

在为Lindorm实例设置白名单前，您需要先根据客户端的所属位置，获取应添加的客户端IP地址。

客户端位置	连接使用的网络类型	获取客户端IP地址的方法
（推荐）ECS实例	专有网络	查询ECS实例的IP地址 **说明** 需确保ECS实例与Lindorm实例属于同一专有网络。
本地	公网	根据本地设备的操作系统，选择下述方法： • Linux操作系统：在本地设备上执行`curl ipinfo.io |grep ip`命令，获取其公网IP地址。 • Windows操作系统：在本地设备上访问查询网获取其公网IP地址。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**访问控制**。
5. 单击**修改分组白名单**。
6. 在**修改分组白名单**对话框中，添加允许访问该实例的IP地址或者IP地址段。

说明

- 支持如下格式的IP地址或者IP地址段。
  - 单个IP地址，例如：192.0.XX.XX。
  - IP地址段支持CIDR模式，即无类域间路由。例如：192.0.XX.XX/24，其中/24表示地址中前缀的长度，范围[1，32]。
- 多个IP地址或IP地址段之间以英文逗号（,）分隔。
- IP地址段设置为0.0.0.0/0表示允许所有IP地址访问Lindorm实例，此时云原生多模数据库Lindorm将面临高安全风险，请谨慎使用。
- IP地址设置为127.0.0.1表示禁止所有IP地址访问Lindorm实例。

7. 单击**确认**。

5.4. 连接并使用实例

本文汇总了云原生多模数据库Lindorm实例的宽表引擎、时序引擎、搜索引擎、文件引擎的连接方式和参考文档。

引擎类型	连接方式	参考文档
宽表引擎	通过Lindorm宽表SQL连接宽表引擎	• 通过Lindorm宽表SQL Java API连接并使用宽表引擎 • 通过Lindorm宽表SQL 非Java API连接并使用宽表引擎 • 通过Lindorm-cli连接并使用宽表引擎
	通过HBase API连接宽表引擎	• 通过HBase Java API连接并使用宽表引擎 • 通过HBase非Java API连接并使用宽表引擎 • 通过HBaseue Shell连接并使用宽表引擎
	通过Cassandra CQL连接宽表引擎	• 通过Lindorm-cqlsh连接并使用宽表引擎 • 通过Cassandra CQL Java Driver连接并使用宽表引擎 • 通过Cassandra CQL多语言Driver连接并使用宽表引擎
	通过Phoenix SQL连接宽表引擎	• 通过Phoenix SQL Java API连接并使用宽表引擎 • 通过Phoenix SQL非Java API连接并使用宽表引擎 • 通过Phoenix Sqlline连接并使用宽表引擎
	通过S3协议连接宽表引擎	• 通过S3 Java API连接并使用宽表引擎 • 通过S3 非Java API连接并使用宽表引擎
时序引擎	通过Lindorm-cli连接时序引擎	通过Lindorm-cli连接并使用时序引擎
	通过JDBC Driver连接时序引擎	通过JDBC Driver连接并使用时序引擎（推荐）
	通过Lindorm JDBC Driver连接时序引擎	通过Lindorm JDBC Driver连接并使用时序引擎
搜索引擎	通过solr Shell连接搜索引擎	通过Search Shell连接并使用搜索引擎
	通过SQL连接搜索引擎	通过SQL连接并使用搜索引擎
文件引擎	通过HDFS Shell连接文件引擎	通过HDFS Shell连接并使用文件引擎
	通过开源HDFS客户端连接文件引擎	通过开源HDFS客户端连接并使用文件引擎
	通过HDFS FUSE连接文件引擎	通过HDFS FUSE连接并使用文件引擎
计算引擎	通过JDBC连接计算引擎	使用SQL访问Lindorm计算引擎

6.用户指南
6.1. 实例管理
6.1.1. 创建实例

您可以通过云原生多模数据库Lindorm控制台创建Lindorm实例。本文介绍创建Lindorm实例的步骤，以及创建过程中各个参数的含义。

前提条件
已注册阿里云账号。具体操作请参见注册阿里云账号。

注意事项
如果应用部署在ECS实例，您需要确保云原生多模数据库Lindorm实例和ECS实例满足以下条件，以保证网络的连通性。ECS实例信息的查看方法请参见查看实例信息。

- 所在地域相同，并建议所在可用区相同（以减少网络延时）。
- 网络类型相同。

> **说明** 专有网络具有更高的安全性，建议使用专有网络。如果网络类型为专有网络，需确保使用相同的专有网络ID。

操作步骤
1. 登录Lindorm管理控制台。
2. 在**实例列表**页面，单击**创建**。

   > **说明** 您也可以前往产品详情页面，单击**立即购买**创建实例。

3. 在**商品类型**区域，请根据业务需求，选择**Lindorm（包年包月）**或者**Lindorm（按量付费）**。

   > **说明** 按量付费实例可以转换为包年包月实例，包年包月实例无法转换为按量付费实例。
   > - **Lindorm（包年包月）**：预付费，在创建实例时就需要支付费用。适合长期需求，价格比按量付费更实惠，且购买时长越长，折扣越多。
   > - **Lindorm（按量付费）**：后付费，每小时根据实例配置生成1个收费订单，并扣除相应的账户余额。适合短期需求，用完可立即释放实例，节省费用。

4. 设置以下参数，完成实例的参数配置。

参数	参数说明
部署方案	选择**单可用区**，如果您选择**多可用区**，更多操作请参见创建多可用区实例。 - **单可用区**：主节点和备节点位于同一可用区，适合可用区灾备级别的要求为单机房的场景。 - **多可用区**：主节点和备节点位于不同可用区，具备跨可用区容灾。
地域	选择实例所属地域。 **说明** - 实例创建成功后将无法更换地域，请您谨慎选择。 - 如果应用部署在ECS实例，请您选择与ECS实例相同的地域，否则无法直接通过内网通信。

参数	参数说明
可用区	选择实例所属可用区。 **说明** 如果应用部署在ECS实例，建议您选择与ECS实例相同的可用区，以减少网络延时。
多可用区组合	仅当**部署方案**选择为**多可用区**时会显示此参数。 **说明** 您无法选择**多可用区组合**，Lindorm支持多可用区的每个地域会默认匹配可用区组合。
网络类型	固定为**专有网络**。
专有网络（VPC）	选择实例所属专有网络。如果您没有可以使用的专有网络，您可以在专有网络控制台创建专有网络，创建方法请参见创建和管理专有网络。
专有网络交换机	选择实例所属交换机。如果您没有可以使用的交换机，您可以在专有网络控制台创建交换机，创建方法请参见创建交换机。
集群名称	设置实例的名称，便于后续业务识别。
存储类型	可选标准型云存储、性能型云存储、容量型云存储、本地SSD盘、本地HDD盘。您可以根据实际业务需求选购存储类型，具体请参见存储类型。 ○ **标准型云存储**：适用于Feed流数据、聊天、实时报表、在线计算等实时访问数据的场景。 ○ **性能型云存储**：适用于广告竞价投放、用户画像、人群圈选、实时搜索、风控大脑等低延迟访问数据的场景。 ○ **容量型云存储**：适用于监控日志、历史订单、音视频归档、数据湖存储、离线计算等低频访问数据的场景。 ○ **本地SSD盘**：本地盘来自单台物理机，具有低时延、高随机IOPS、高吞吐量、高性价比等优势。适用于网络游戏、电商、视频直播、媒体等在线业务，满足I/O密集型应用对块存储的低时延和高I/O性能需求。 ○ **本地HDD盘**：本地盘来自单台物理机，具有低时延、高随机IOPS、高吞吐量、高性价比等优势。适用于互联网行业、金融行业等有大数据计算与存储分析需求的行业，进行海量数据存储和离线计算的业务场景。

参数	参数说明
选择数据引擎	包括宽表引擎、时序引擎、搜索引擎、文件引擎和流引擎。您可以根据实际业务需求选购数据引擎的节点规格和数量，具体请参见引擎类型。 ○ **宽表引擎**：可按需选择宽表引擎节点规格和数量，以core为单位。 **说明** ■ 若存储类型选择**本地SSD盘**和**本地HDD盘**，则不支持此功能。 ■ 目前宽表引擎仅支持1：4的CPU内存配比。 ○ **时序引擎**：可按需选择时序引擎节点规格和数量，以core为单位。 **说明** ■ 若存储类型选择**本地SSD盘**和**本地HDD盘**，则不支持此功能。 ■ 目前时序引擎仅支持1：4的CPU内存配比。 ○ **搜索引擎**：可按需选择搜索引擎节点规格和数量，以core为单位。 **说明** ■ 若存储类型选择**本地SSD盘**和**本地HDD盘**，则不支持此功能。 ■ 目前搜索引擎仅支持1：4的CPU内存配比。 ○ **文件引擎**：可按需选择文件引擎节点数量，节点规格默认为**标准规格**。 **说明** 若存储类型选择**本地SSD盘**和**本地HDD盘**，则不支持此功能。
存储空间	选择实例的存储空间的大小。
是否选购冷存储	实例是否需要开通冷存储功能，更多请参见冷存储介绍。 **说明** 云存储系列实际可用存储容量为总容量。本地盘系列集群采用三副本架构，实际可用存储容量为总容量的三分之一。
冷存储存储容量	选择实例的冷存储空间大小。初次购买容量上限为100000 GB，后续可无限扩容。 **说明** 仅当**是否选购冷存储**参数选择**是**时会配置此参数。
购买时长	如果您创建的是包年包月实例，请您选择购买时长。包月可选择1~9个月，包年可选择1~3年。 **说明** 如果您希望实例到期后自动续费，勾选**到期自动续费**，并确保实例在即将到期前您的账户中有足够的余额。

5. 设置完以上参数后，单击**立即购买**。
6. 在**确认订单**页面，阅读并勾选服务协议，并单击**立即开通**。
7. 支付完成后，单击**登录控制台**，进入云原生多模数据库Lindorm控制台。

6.1.2. 变更实例配置

6.1.2.1. 变更实例规格

云原生多模数据库Lindorm实例支持变更各引擎的规格配置，当Lindorm实例的引擎规格不符合（高于或低于）当前业务需求时，您可以对当前实例的引擎进行变更规格操作，使当前实例的引擎规格与您的业务需求更加匹配。

前提条件

已创建云原生多模数据库Lindorm实例并且实例状态为**运行中**，具体操作，请参见创建实例。

注意事项

- 对Lindorm实例的引擎进行降配规格操作时，请确保降配后的规格能满足业务的访问需求。
- 变更实例规格的过程中可能会发生抖动。
- 对当前业务评估实例规格时，不仅需要考虑业务的读写请求量，而且还要综合考虑多方面因素。如果评估规格时需要帮助请提交工单。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在**配置信息**区域中，选择需要变更配置的引擎对应的**管理**。
5. 单击**升级规格**或者**降配规格**。

   > **说明** 云原生多模数据库Lindorm宽表引擎、时序引擎和搜索引擎支持**升级规格**和**降配规格**，其他引擎类型支持部分变更配置操作，具体请以控制台显示为准。

6. 在**降配**或者**变配**页面，设置引擎的节点规格。

   > **说明** 有关引擎的节点规格选择方法请参见选择节点规格和数量。

7. 阅读并勾选服务协议，并单击**立即购买**。

   执行变更规格操作时，实例状态为**规格变更中**，当实例状态变更为**运行中**，说明变更规格操作成功。

   > **说明** 变更规格的等待时间和引擎节点数量有关。通常一个节点需要等待2～10分钟，节点越多，变更等待时间越长。

6.1.2.2. 变更实例节点

云原生多模数据库Lindorm支持变更各引擎的节点数量，当Lindorm实例的引擎节点数不符合（高于或低于）当前业务需求时，您可以对当前实例的引擎进行添加节点或者减少节点操作。

前提条件

已创建云原生多模数据库Lindorm实例并且实例状态为**运行中**，具体操作，请参见创建实例。

注意事项

- 对Lindorm实例的引擎进行减少节点操作时，请确保降配后的节点数量能满足业务的访问需求。变更实例节点时，如果您遇到困难请提交工单。
- 变更实例节点数量的过程中可能会发生抖动。
- 减少引擎节点数量时，至少需要保留两个节点，确保实例在运维变更时至少有一个节点可以提供服务。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在**配置信息**区域中，选择需要变更节点的引擎对应的**管理**。
5. 单击**添加节点**或者**减少节点**。

> **说明**
> - 云原生多模数据库Lindorm宽表引擎和搜索引擎支持**添加节点**和**减少节点**，其他引擎类型支持部分变更节点操作，具体请以控制台显示为准。
> - 减少引擎节点数量时，至少需要保留两个，确保实例在运维变更时至少有一个节点可以提供服务。

6. 在**降配**或者**变配**页面，设置引擎的节点数量和存储空间。

> **说明** 有关引擎的节点数量建议请参见选择节点规格和数量。

7. 阅读并勾选服务协议，并单击**立即购买**。

   执行变更节点操作时，实例状态为**节点扩容中**或**节点缩容中**，当实例状态变更为**运行中**，说明变更节点操作成功。

> **说明** 变更节点的等待时间和节点数量有关。通常一个节点需要等待2～10分钟，节点越多，变更等待时间越长。

6.1.3. 升级小版本

云原生多模数据库Lindorm发布新的引擎小版本时，您可以通过控制台一键将实例引擎升级至最新的小版本。升级小版本可以完善老版本的功能，同时也为您提供更多的新功能。

前提条件

已创建云原生多模数据库Lindorm实例，具体操作，请参见创建实例。

注意事项

- 升级小版本不支持回滚操作，建议优先考虑升级到稳定版本。
- 执行升级小版本的操作，将导致实例被重启，请在业务低峰期操作或确保应用具备重连机制。
- 升级时间与所在地域的实例个数有关，个数越多升级时间越长。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在**配置信息**区域中，单击需要升级的引擎对应的**管理** > **小版本升级**。



> **说明**
> - 系统会自动检测实例的引擎版本，如果**小版本升级**按钮不存在，表示该实例的引擎版本已经是最新的。
> - 单击不同引擎对应的**版本说明**，可以查看实例的引擎版本说明。

5. 在**升级小版本**对话框中，选择目标版本为**最新版本**或者**稳定版本**。

> **警告** 小版本升级不支持回滚操作，建议优先考虑升级至稳定版本。

6. 单击**确定**。

6.1.4. 开启或关闭删除保护

开启实例删除保护可以防止您在操作过程中将承载有重要业务的实例误释放。开启实例删除保护后，将无法释放该实例，如果需要释放，需要先关闭实例删除保护，然后释放实例。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在**基本信息**区域，单击**删除保护**右侧的**启用**或者**禁用**。
5. 在**启用删除保护**或者**禁用删除保护**对话框，单击**确定**。

> **说明** 开启实例删除保护后，将无法释放该实例，如果需要释放，需要先关闭实例删除保护。您可以选择以下任意一种方法释放实例。
>
> - 通过云原生多模数据库Lindorm控制台释放，具体操作请参见释放实例。
> - 通过调用ReleaseLindormInstance接口释放，具体请参见ReleaseLindormInstance。

6.1.5. 释放实例

云原生多模数据库Lindorm支持手动释放按量付费的实例。

前提条件

目标实例的付费类型为按量付费，包年包月实例无法手动释放。

注意事项

目标实例释放后，实例的所有数据都会被同时删除无法找回，请谨慎操作。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页面，单击目标实例所在行**操作**列的⋮，并选择**释放**。

> **说明** 如果目标实例的服务类型为Lindorm Serverless版，单击目标实例所在行**操作**列的**释放**。

4. 在弹出的对话框中，单击**确定**。

> **注意** 目标实例释放后，实例的所有数据都会被同时删除无法找回，请谨慎操作。

6.1.6. 设置可维护时间段

为保障云原生多模数据库Lindorm实例的稳定性，阿里云会不定期对实例进行维护操作。可维护时间段是指您允许阿里云进行维护操作的时间段。建议您将维护时间段设置在业务低峰期，以避免维护过程中可能对业务造成的影响。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在**基本信息**区域，单击**可维护时间段**右侧的**编辑**。
5. 在**修改可维护时间段**对话框，单击**可维护时间段**列表并选择实例的可维护时间段。
6. 单击**确定**。

6.1.7. 查看连接地址

本文介绍通过云原生多模数据库Lindorm控制台查看各引擎类型的连接地址。

专有网络和公网

网络类型	说明
专有网络	专有网络VPC（Virtual Private Cloud）是您自己独有的云上私有网络，不同的专有网络之间通过二层逻辑隔离，拥有较高的安全性和性能。Lindorm-cli部署在ECS实例上时，通过专有网络连接至Lindorm实例，可获得更高的安全性和更低的网络延迟。
公网	公网即互联网，当本地设备需要连接至Lindorm实例时，您可以为Lindorm实例申请公网连接地址。通过公网连接不会产生流量费用，但存在一定的安全风险，推荐通过专有网络连接以获取更高的安全性。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**数据库连接**。
5. 切换至目标引擎页签并查看各引擎的连接地址。

   > **说明** 如果您需要使用公网地址连接目标引擎，请单击目标引擎页签右上角的**开通外网地址**。

   - 查看宽表引擎连接地址
   - 查看时序引擎连接地址
   - 查看搜索引擎连接地址
   - 查看文件引擎连接地址
   - 查看计算引擎连接地址
   - 查看流引擎连接地址

查看宽表引擎连接地址

查看宽表引擎的连接地址前，请确保已开通Lindorm宽表引擎。如果您需要使用公网地址连接宽表引擎，请单击**数据库连接**的**宽表引擎**页签右上角的**开通外网地址**。不同类型的宽表引擎连接地址如下表：



连接地址类型	连接方式	服务类型	网络类型	参考文档
宽表SQL地址	通过Lindorm宽表SQL连接宽表引擎（连接地址为图示中①）	• Lindorm • Lindorm单机版	• 专有网络 • 公网	• 通过Lindorm宽表SQL Java API连接并使用宽表引擎 • 通过Lindorm宽表SQL 非Java API连接并使用宽表引擎 • 通过Lindorm-cli连接并使用宽表引擎
HBase兼容地址	通过HBase Java API连接宽表引擎（连接地址为图示中②）	• Lindorm • Lindorm单机版 • Lindorm Serverless版	• 专有网络 • 公网	• 通过HBase Java API连接并使用宽表引擎 • 通过HBaseue Shell连接并使用宽表引擎
	通过HBase 非Java API连接宽表引擎（连接地址为图示中③）	• Lindorm • Lindorm单机版 • Lindorm Serverless版	• 专有网络 • 公网	通过HBase非Java API连接并使用宽表引擎
Cassandra兼容地址	通过CQL连接宽表引擎（连接地址为图示中④）	• Lindorm • Lindorm单机版 • Lindorm Serverless版	• 专有网络 • 公网	• 通过Lindorm-cqlsh连接并使用宽表引擎 • 通过Cassandra CQL Java Driver连接并使用宽表引擎 • 通过Cassandra CQL多语言Driver连接并使用宽表引擎

连接地址类型	连接方式	服务类型	网络类型	参考文档
S3兼容地址	通过S3协议连接宽表引擎（连接地址为图示中⑤）	Lindorm	专有网络	说明 查看S3兼容地址需要开通S3协议兼容功能，请参见开通S3协议兼容功能。 • 通过S3 Java API连接并使用宽表引擎 • 通过S3非Java API连接并使用宽表引擎

查看时序引擎连接地址

查看时序引擎的连接地址前，请确保已开通Lindorm时序引擎。如果您需要使用公网地址连接时序引擎，请单击**数据库连接**的**时序引擎**页签右上角的**开通外网地址**。不同类型的时序引擎连接地址如下表：



连接地址类型	连接方式	服务类型	网络类型	参考文档
时序SQL地址	通过JDBC Driven连接时序引擎（连接地址为图示中①）	• Lindorm • Lindorm单机版	• 专有网络 • 公网	通过JDBC Driver连接并访问Lindorm时序引擎
时序HTTP地址	通过Lindorm-cli连接时序引擎（连接地址为图示中②）	• Lindorm • Lindorm单机版	• 专有网络 • 公网	• 通过Java Native SDK连接并使用Lindorm时序引擎 • 通过Lindorm-cli连接访问Lindorm时序引擎
PromQL兼容地址	通过PromQL插件来访问时序引擎（连接地址为图示中③）	Lindorm	• 专有网络 • 公网	对接Prometheus

查看搜索引擎连接地址

查看搜索引擎的连接地址前，请确保已开通Lindorm搜索引擎。如果您需要使用公网地址连接搜索引擎，请单击**数据库连接**的**时序引擎**页签右上角的**开通外网地址**。不同类型的搜索引擎连接地址如下表：



连接地址类型	连接方式	服务类型	网络类型	参考文档
搜索SQL地址	通过SQL连接搜索引擎（连接地址为图示中①）	Lindorm	• 专有网络 • 公网	通过SQL连接并使用搜索引擎
Solr兼容地址	通过Search Shell连接搜索引擎（连接地址为图示中②）	• Lindorm • Lindorm单机版	专有网络	通过Search Shell连接并使用搜索引擎

查看文件引擎连接地址

查看文件引擎的连接地址前，请确保已开通Lindorm文件引擎。

文件引擎提供了客户端连接配置，请单击**数据库连接**的**文件引擎**页签中**一键生成配置项**就可以查看对应的配置内容。

查看计算引擎连接地址

查看计算引擎的连接地址前，请确保已开通Lindorm计算引擎。如果您需要使用公网地址连接计算引擎，请单击**数据库连接**的**计算引擎**页签右上角的**开通外网地址**。不同类型的计算引擎连接地址如下表：



连接地址类型	连接方式	服务类型	网络类型	参考文档
JDBC地址	通过SQL连接计算引擎（连接地址为图示中①）	• Lindorm • Lindorm Serverless版	• 专有网络 • 公网	使用SQL访问Lindorm计算引擎
JAR地址	通过计算引擎提交JAR作业（连接地址为图示中②）	• Lindorm • Lindorm Serverless版	• 专有网络 • 公网	JAR作业开发实践

查看流引擎连接地址

查看流引擎的连接地址前，请确保已开通Lindorm流引擎。如果您需要使用公网地址连接流引擎，请单击**数据库连接**的**流引擎**页签右上角的**开通外网地址**。不同类型的流引擎连接地址如下表：



连接地址类型	连接方式	服务类型	网络类型	参考文档
Stream SQL地址	通过Lindorm-cli连接Lindorm流引擎（连接地址为图示中①）	Lindorm	• 专有网络 • 公网	通过Lindorm-cli连接并使用Lindorm流引擎
Stream Kafka地址	通过开源Kafka客户端连接Lindorm流引擎（连接地址为图示中②）	Lindorm	专有网络	通过开源Kafka客户端写入Lindorm流引擎数据

6.1.8. 创建多可用区实例

云原生多模数据库Lindorm支持创建多可用区的实例。该方案将一个Lindorm实例部署在多个可用区，多可用区实例具备更高的容灾能力。当多个可用区中的任一可用区因电力、网络等不可抗因素失去通信时，容灾系统将自动触发切换操作，确保实例的持续可用。

功能特性

云原生多模数据库Lindorm多可用区部署支持以下功能特性：

- 具备机房级或者城市级容灾能力。
- 满足Lindorm实例的数据强一致或者最终一致需求。
- 故障识别和切换操作都是由Lindorm自动触发，易用性高。

操作步骤

1. 登录Lindorm管理控制台。
2. 在**实例列表**页面，单击**创建**。
3. 在**商品类型**区域，请根据业务需求，选择**Lindorm（包年包月）**或**Lindorm（按量付费）**。

   > **说明** 按量付费可转为包年包月，包年包月无法转为按量付费。
   > - **Lindorm（包年包月）**：适合长期需求，价格比按量付费更实惠，且购买时长越长，折扣越多。
   > - **Lindorm（按量付费）**：后付费，按小时扣费。适合短期需求，实例在使用后可立即释放，节省费用。

4. 设置以下参数，完成多可用区实例的参数配置。

参数	参数说明
部署方案	选择**多可用区**。

参数	参数说明
地域	选择实例所属地域。 **说明** ○ 实例创建成功后将无法更换地域，请您谨慎选择。 ○ 如果应用部署在ECS实例，请您选择与ECS实例相同的地域，否则无法直接通过内网通信。
多可用区组合	选择实例所属的多可用区组合。 **说明** 您无法选择可用区组合，Lindorm支持多可用区的每个地域会默认匹配三个可用区组合。
网络类型	固定为**专有网络**。
专有网络（VPC）	选择实例所属专有网络。如果您没有可以使用的专有网络，您可以在专有网络控制台创建专有网络，创建方法请参见创建和管理专有网络。
主可用区	选择实例的主可用区。
主可用区VSwitch	选择实例主可用区所属的交换机。如果您没有可以使用的交换机，您可以在专有网络控制台创建交换机，创建方法请参见创建和管理交换机。
备可用区	选择实例的备可用区。
备可用区VSwitch	选择实例备可用区所属的交换机。如果您没有可以使用的交换机，您可以在专有网络控制台创建交换机，创建方法请参见创建和管理交换机。
协调节点可用区	选择实例的协调节点可用区。
协调节点可用区VSwitch	选择实例协调节点可用区所属的交换机。如果您没有可以使用的交换机，您可以在专有网络控制台创建交换机，创建方法请参见创建和管理交换机。
集群名称	设置实例的名称，便于后续业务识别。
存储类型	可选标准型云存储、性能型云存储、本地SSD盘、本地HDD盘。您可以根据实际业务需求选购存储类型，具体请参见存储类型。
宽表节点规格(HBase/Cassandra API)	您可以根据业务需要选择宽表节点规格，相关信息请参见集群规格选型。
宽表节点数量	宽表节点是跨两个可用区部署的，所以宽表节点数量要以2的倍数增加，每个可用区至少2个宽表节点。
Log规格	建议您选择默认的Log规格4核8 GB。 **说明** Log仅承载日志WAL。所有强一致表的日志WAL会备份至Log节点，由于不直接承载用户流量，建议您选择默认的规格，如果实例数据比较多，配置的咨询方法请参见技术支持。
Log节点数量	建议您选择默认的Log节点数量。
core单节点容量	根据业务需求，选择Core单节点容量。

参数	参数说明
Log磁盘类型	请根据业务场景选择Log的磁盘类型。 ○ SSD云盘：应用在数据经常被访问或者高I/O负载的场景。适用于MongoDB、HBase、Cassandra等非关系型数据库。 ○ 高效云盘：应用在I/O密集型场景。适用于中小型关系数据库。
Log单节点容量	根据业务需求，选择Log单节点容量。
是否选购冷存储	固定为否。
购买时长	如果您创建的是包年包月实例，请您选择购买时长。包月可选择1~9个月，包年可选择1~3年。 **说明** 如果您希望实例到期后自动续费，勾选**到期自动续费**，并确保实例在即将到期前您的账户中有足够的余额。

5. 单击**立即购买**。
6. 在**确认订单**页面，阅读并勾选服务协议，并单击**立即开通**。
7. 支付完成后，单击**登录控制台**，进入云原生多模数据库Lindorm控制台。

6.2. 监控与报警

6.2.1. 查看监控信息

通过云原生多模数据库Lindorm控制台和云监控控制台查看Lindorm实例各引擎的监控信息，您可以了解实例的常用资源（例如CPU使用率和内存使用率）使用情况。本文介绍查看Lindorm实例各引擎监控信息的方法。

通过云原生多模数据库Lindorm控制台监控

通过云原生多模数据库Lindorm控制台查看监控信息仅支持Lindorm单可用区实例。

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，单击**实例监控**，可以查看各项监控指标。

通过云监控控制台监控

Lindorm使用云监控的云产品监控功能，以实例为粒度组织和展示监控数据。

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，单击**实例监控**。
5. 单击**云监控**，跳转至Lindorm实例的云产品监控页面。

> **说明** Lindorm实例的监控项说明请参见云原生多模数据库Lindorm监控项。

6.2.2. 报警规则

6.2.2.1. 创建报警规则

云原生多模数据库Lindorm支持为实例的重要监控项创建报警规则。当监控项的值不在设置的阈值范围内时，系统会自动向报警联系组中联系人发出报警通知，从而快速定位问题。本文介绍通过云监控控制台创建报警规则的方法。

前提条件

如果报警规则需要作用于指定应用分组的实例上，请您确保已创建应用分组，且已将实例添加至该应用分组。具体操作，请参见创建应用分组和添加资源至应用分组。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏单击**报警配置**。
5. 单击**云监控**，进入云监控控制台。
6. 在云监控控制台，为Lindorm实例创建报警规则。
   i. 在**报警规则列表**页面单击**创建报警规则**。
   ii. 在**创建报警规则**面板，设置以下参数。

参数	说明
产品	固定填写为**云原生多模数据库Lindorm**。
资源范围	报警规则作用的资源范围。取值： ■ **全部资源**：表示该规则作用于当前阿里云账号下云原生多模数据库Lindorm的全部实例上。 ■ **应用分组**：表示该规则作用于云原生多模数据库Lindorm的指定应用分组内的全部实例上。 ■ **实例**：表示该规则作用于某个指定Lindorm实例上。
规则描述	报警规则的主体。当监控数据满足报警条件时，触发报警规则。规则描述的设置方法如下： a. 单击**添加规则**。 b. 在**添加规则描述**面板，设置规则名称、指标类型、监控指标、维度、阈值、报警级别和报警方式等。 c. 单击**确定**。
关联资源	选择云原生多模数据库Lindorm的应用分组名称或者实例ID。 **说明** 当资源范围为**应用分组**或者**实例**时配置该参数。
通道沉默周期	监控指标出现异常时发送报警通知的间隔时长。取值：5分钟、15分钟、30分钟、60分钟、3小时、6小时、12小时和24小时。 某监控指标达到报警阈值时发送报警，如果监控指标在通道沉默周期内持续超过报警阈值，在通道沉默周期内不会重复发送报警通知。如果监控指标在通道沉默周期后仍未恢复正常，则云监控再次发送报警通知。 **说明** 单击**高级设置**，可设置该参数。
生效时间	报警规则的生效时间，报警规则只在生效时间内才会检查监控数据是否需要报警。 **说明** 单击**高级设置**，可设置该参数。
报警联系人组	发送报警通知的联系人组。 应用分组的报警通知会发送给该报警联系人组中的报警联系人，报警联系人组中可以包含一个或多个报警联系人。关于如何创建报警联系人和报警联系人组，请参见创建报警联系人或报警联系组。

参数	说明
报警回调	公网可访问的URL，用于接收云监控通过POST请求推送的报警信息。目前仅支持HTTP协议。关于如何设置报警回调，请参见使用阈值报警回调。
弹性伸缩	开启**弹性伸缩**开关，当报警发生时，会触发相应的伸缩规则。您需要设置弹性伸缩的地域、弹性伸缩组和弹性伸缩规则。 ■ 关于如何创建弹性伸缩组，请参见创建伸缩组。 ■ 关于如何创建弹性伸缩规则，请参见配置伸缩规则。
日志服务	开启**日志服务**开关，当报警发生时，会将报警信息写入日志服务的日志库。您需要设置日志服务的地域、ProjectName和Logstore。关于如何创建Project和Logstore，请参见快速入门。
消息服务MNS-Topic	开启**消息服务MNS-Topic**开关，当报警发生时，会将报警信息写入消息服务的主题。您需要设置消息服务的地域和主题。关于如何创建主题，请参见创建主题。
无数据报警处理方法	无监控数据时报警的处理方式。取值： ■ **不做任何处理**（默认值） ■ **发送无数据报警** ■ **视为正常**
标签	报警规则的标签。您可以设置标签键和标签值。

7. 设置完成后，单击**确定**。

6.2.2.2. 管理报警

云原生多模数据库Lindorm实例提供实例监控功能，当监控指标达到报警条件时报警联系组中联系人会及时收到报警通知。本文介绍通过云原生多模数据库Lindorm控制台来管理报警规则。

背景信息

监控报警是通过阿里云监控产品实现的。通过阿里云监控产品，您可以设置监控项，并在触发监控项的报警规则时，通过邮件和短信通知报警联系组中的所有联系人。您也可以维护报警监控项对应的报警联系组，以便发生报警时，能及时通知到相关联系人。

创建报警规则

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏单击**报警配置**可以查看实例的报警规则。

> **说明** 通过云原生多模数据库Lindorm控制台创建的报警规则无法在云监控控制台的报警规则列表中查看，通过云监控控制台创建报警规则的方法请参见创建报警规则。



5. 在页面右上角单击**创建报警**。
6. 在**创建报警**面板，设置以下参数。

参数	说明

参数	说明
告警模板	选择报警模板，如果您需要创建新的报警模板，具体操作请参见创建报警模板。 说明 报警模板仅支持时序引擎的指标模板。
规则名称	报警规则的名称。
告警表达式	报警规则的主体。当监控指标满足报警条件时，触发报警规则。
持续时间	设置报警持续的时间长度，单位为分钟。
告警消息	设置收到的报警信息。
创建标签	报警规则的标签。您可以设置标签键和标签值。
创建注释	报警规则的注释。您可以设置注释键和注释值。
通知策略	选择**不指定通知规则**或者指定通知规则发送报警。您可以单击**新建通知**创建通知策略，具体参数说明请参见创建通知策略。

7. 设置完成后，单击**确定**。

创建通知策略

在**创建报警**面板中，您可以单击页面右下方的**新建通知**创建通知策略，进入**新建通知策略**面板，参数说明如下：

参数	说明
通知策略名称	自定义通知策略名称。
通知人	通知人支持设置联系人、联系人组和IM机器人。 说明 联系人的创建方法，请参见联系人概述。
通知方式	通知方式支持钉钉、邮件、WebHook、短信和电话，可以同时选择多种方式。 说明 未验证手机号的联系人无法使用电话通知方式。验证手机号的操作，请参见验证手机号。
通知时段	告警会在设置的通知时段内重复发送告警通知。

管理报警规则

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏单击**报警配置**。
5. 在报警规则列表中找到目标报警规则，在**操作**列中选择需要进行的操作，包括以下操作。
   - 查看报警规则的历史告警：单击**操作**列的**告警历史**，跳转至应用实时监控服务ARMS控制台查看该报警规则的历史告警。
   - 修改报警规则：单击**操作**列的**编辑**，在**编辑报警**面板修改参数，详细参数说明请参见创建报警规则。
   - 关闭报警规则：单击**操作**列的**关闭**，关闭后当资源的监控指标达到报警条件时，不会触发报警。
   - 删除报警规则：单击**操作**列的**删除**，删除后无法恢复，只能重新创建。

6.3. 标签管理

6.3.1. 新建标签

当云原生多模数据库Lindorm实例数量较多时，您可以通过标签对Lindorm实例进行分类和筛选。本文介绍新建标签的方法。

前提条件

已创建云原生多模数据库Lindorm实例，具体操作请参见创建实例。

使用限制

- 标签由一对键值组成，键在同一个Lindorm实例中是唯一的，值没有限制。
- 每个实例最多可以绑定20个标签。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页面，单击目标实例所在行**操作**列的⋮，并选择**编辑标签**。
4. 在**编辑标签**对话框中，单击**新建标签**。

   > **说明** 如果您已经新建了标签，请单击**已有标签**，将已新建的标签绑定至目标实例，具体操作请参见绑定已有标签。

5. 设置标签的键和值，并单击右侧的**确定**。
6. 单击**编辑标签**对话框右下角的**确定**。

   > **说明** 新建标签完成后，您可以在其它实例中绑定已新建的标签。

6.3.2. 绑定已有标签

完成新建标签操作后，您可以为云原生多模数据库Lindorm实例绑定不同的标签对其进行分类和筛选，本文介绍为Lindorm实例绑定已有标签的方法。

前提条件

Lindorm实例已完成新建标签，具体操作请参见新建标签。

使用限制

每个Lindorm实例最多可以绑定20个标签。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页面，单击目标实例所在行**操作**列的⋮，并选择**编辑标签**。
4. 在**编辑标签**对话框中，单击**已有标签**并选中所需的标签信息。

   > **说明** 如果单击**已有标签**后没有弹出标签信息，您需要新建标签，具体操作请参见新建标签。

5. 单击**确定**。

6.3.3. 根据标签筛选实例

为Lindorm实例绑定标签后，您可以在实例列表中根据绑定的标签来筛选实例。

前提条件

Lindorm实例已完成新建标签，具体操作请参见新建标签。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页面单击**标签**，选中需要筛选的标签键或者标签值。

> **说明** 如果需要清除筛选的标签，单击已选中标签右侧的。

6.3.4. 解绑或删除标签

如果Lindorm实例不需要某个标签时，您可以解除Lindorm实例和该标签的绑定关系。如果该标签没有绑定到其他Lindorm实例，那么该标签会被删除。

注意事项

- 当标签与所有绑定的实例解绑后，该标签会被自动删除。
- 每次解绑的标签数量不能超过20个。
- 解绑标签的操作对实例运行无影响。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页面，单击目标实例所在行**操作**列的，并选择**编辑标签**。
4. 在**编辑标签**对话框中，单击目标标签右侧的。
5. 单击**确定**。

6.4. 添加白名单和安全组

6.4.1. 设置白名单

为保障云原生多模数据库Lindorm的安全稳定，系统默认任何设备均无法访问Lindorm实例，所以在使用Lindorm实例前，您需要为实例设置白名单，以允许外部设备访问该实例。正确使用白名单可以使Lindorm实例得到更高级别的安全保护，建议您定期维护白名单。

准备工作

在为Lindorm实例设置白名单前，您需要先根据客户端的所属位置，获取应添加的客户端IP地址。

客户端位置	连接使用的网络类型	获取客户端IP地址的方法
（推荐）ECS实例	专有网络	查询ECS实例的IP地址 **说明** 需确保ECS实例与Lindorm实例属于同一专有网络。
本地	公网	根据本地设备的操作系统，选择下述方法： • Linux操作系统：在本地设备上执行 `curl ipinfo.io |grep ip` 命令，获取其公网IP地址。 • Windows操作系统：在本地设备上访问查询网获取其公网IP地址。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**访问控制**。
5. 单击**修改分组白名单**。
6. 在**修改分组白名单**对话框中，添加允许访问该实例的IP地址或者IP地址段。

   > **说明**
   > - 支持如下格式的IP地址或者IP地址段。
   >   - 单个IP地址，例如：192.0.XX.XX。
   >   - IP地址段支持CIDR模式，即无类域间路由。例如：192.0.XX.XX/24，其中/24表示地址中前缀的长度，范围[1，32]。
   > - 多个IP地址或IP地址段之间以英文逗号（,）分隔。
   > - IP地址段设置为0.0.0.0/0表示允许所有IP地址访问Lindorm实例，此时云原生多模数据库Lindorm将面临高安全风险，请谨慎使用。
   > - IP地址设置为127.0.0.1表示禁止所有IP地址访问Lindorm实例。

7. 单击**确认**。

6.4.2. 添加安全组

安全组是一种虚拟防火墙，用于控制安全组中的ECS实例的出入流量。在Lindorm实例的白名单中添加安全组后，该安全组中的ECS实例就可以访问Lindorm实例。本文介绍添加安全组的方法。

前提条件

通过ECS安全组设置Lindorm白名单时，需要满足以下条件：

- 已创建ECS安全组，具体操作请参见创建安全组。
- ECS实例和Lindorm实例的网络类型相同。如果是专有网络，需要属于同一个专有网络。

注意事项

- 您可以同时设置实例的白名单和安全组。白名单中的IP地址和安全组中的ECS实例都可以访问该Lindorm实例。
- 每个云原生多模数据库Lindorm实例最多添加三个安全组。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**访问控制**。
5. 单击**安全组**页签。
6. 单击**添加安全组**。
7. 在**添加安全组**对话框中选中需要添加的安全组。
8. 单击**确定**。

6.5. Lindorm服务关联角色

在使用云原生多模数据库Lindorm的数据加密TDE功能时，您需要将名称为AliyunServiceRoleForTSDBLindormEncryption的关联角色授权给云原生多模数据库Lindorm使用。经过授权后，Lindorm可以使用当前云账号下数据加密功能。

背景信息

服务关联角色是与某个云服务关联的角色，在您使用特定功能时，关联的云服务会自动创建或删除服务关联角色，不需要您主动创建或删除。例如服务关联角色（AliyunServiceRoleForTSDBLindormEncryption）在某些场景下可以帮助Lindorm访问密钥管理服务（KMS）获取访问权限。开启数据加密功能时系统会自动创建服务关联角色，关于服务关联角色，请参见服务关联角色。

> **说明** 服务关联角色的权限策略由关联的云服务定义和使用，您不能修改或删除权限策略，也不能为服务关联角色添加或移除权限。

AliyunServiceRoleForTSDBLindormEncryption关联角色介绍

角色名称：**AliyunServiceRoleForTSDBLindormEncryption**

角色权限策略：**AliyunServiceRoleForTSDBLindormEncryption**

权限说明：允许云原生多模数据库Lindorm访问KMS中的相关资源，在Lindorm的数据加密TDE功能中使用该权限可以查询和管理密钥，详细策略内容如下所示：

```
{
  "Version": "1",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "kms:ListKeys",
        "kms:ListResourceTags",
        "kms:DescribeKey"
      ],
      "Resource": [
        "acs:kms:*:*:*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:GenerateDataKey",
        "kms:CreateAlias"
      ],
      "Resource": [
        "acs:kms:*:*:*"
      ],
      "Condition": {
        "StringEqualsIgnoreCase": {
          "kms:tag/acs:lindorm:instance-encryption": "true"
        }
      }
    },
    {
      "Action": "ram:DeleteServiceLinkedRole",
      "Resource": "*",
      "Effect": "Allow",
      "Condition": {
        "StringEquals": {
          "ram:ServiceName": "encryption.hitsdb.aliyuncs.com"
        }
      }
    }
  ]
}
```

删除服务关联角色

如果您需要删除服务关联角色：AliyunServiceRoleForTSDBLindormEncryption，请先确定您账号下没有正在使用该服务关联角色的实例，具体操作请参见删除服务关联角色。

RAM用户创建服务关联角色所需要的权限

创建服务关联角色的权限通常包含在其对应云服务的管理员权限策略，因此只要具有该云服务的管理员权限，就可以为该云服务创建服务关联角色。

如果您的RAM用户权限不足，您可以添加下述权限后再执行关联角色的授权操作，添加权限的方法请参见创建自定义权限策略和为RAM用户授权。您也可以直接使用主账号执行关联角色的授权操作。

```
{
  "Action": "ram:CreateServiceLinkedRole",
  "Resource": "*",
  "Effect": "Allow",
  "Condition": {
    "StringEquals": {
      "ram:ServiceName": "encryption.hitsdb.aliyuncs.com"
    }
  }
}
```

6.6. 计费管理

6.6.1. 按量付费转包年包月

如果您需要长期使用按量付费实例，为了节约成本，可以将实例的付费类型转换为包年包月，并根据业务的使用时间选择包年包月时长。

前提条件

实例的计费类型为按量付费。

注意事项

- 包年包月实例无法转换为按量付费实例。
- 包年包月实例无法释放。
- 服务类型为Lindorm Serverless的实例不支持按量付费转包年包月。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页面，单击目标实例所在行**操作**列的**转预付费**。
4. 在**后付费转预付费**页面，选择实例的**购买时长**。
5. 阅读并勾选服务协议，单击**立即购买**。

   > **说明** 购买后，系统会生成类型为**转换**的订单。如果该订单没有支付或者作废，您将无法购买新实例或转包年包月的操作。您可以登录订单管理控制台完成订单支付或者作废该订单。

6. 在**支付**页面，根据提示完成支付流程。
7. 支付完成后，单击**登录控制台**，进入云原生多模数据库Lindorm控制台，实例的付费类型更新为**包年包月**。

6.6.2. 开通自动续费

开通自动续费包年包月实例可以减少定期手动续费的管理成本，避免因忘记续费而导致Lindorm实例服务中断。如果包年包月实例不需要自动续费，您可以关闭自动续费功能。本文介绍开通自动续费的方法。

前提条件

Lindorm实例的计费类型为包年包月。

注意事项

- Lindorm包年包月实例的自动续费按照实例到期日计算第一次自动续费日期和计费周期。自动续费首次扣款时间为实例到期前第9天。如果扣款失败，系统将在接下来的每天尝试扣款，请确保阿里云账号的余额充足，避免续费失败。
- 如果在自动续费前已进行手动续费，则同一计费周期内不会再自动续费。

- 如果自动续费扣款失败，系统会向阿里云账号绑定的手机号和邮箱发送提醒信息，请注意查收并处理，避免Lindorm实例停机造成业务中断。

开通方法

开通自动续费支持以下两种方法：

- 创建实例时开通自动续费
- 实例使用时开通自动续费

创建实例时开通自动续费

> 说明
> 在购买Lindorm实例时开通自动续费，如果按月购买，自动续费周期为1个月；如果按年购买，自动续费周期为1年。例如购买时长为3个月并勾选到期自动续费，那么该实例3个月购买时长到期前会自动续费1个月。

创建云原生多模数据库Lindorm包年包月实例时，选择**购买时长**并勾选**到期自动续费**，Lindorm实例到期前会自动续费，如下图所示。



实例使用时开通自动续费

> 说明
> 在续费管理控制台上开通实例的自动续费，您可以选择续费周期，系统将根据您选择的续费周期进行自动续费。例如续费周期选择为3个月，那么该实例3个月续费周期到期前会自动续费1个月。

1. 登录Lindorm管理控制台。
2. 在页面右上角，单击**费用>续费管理**。
3. 在**续费管理**页面，从**产品**下拉列表中选择云原生多模数据库Lindorm，单击**手动续费**或者**到期不续费**页签。
4. 根据续费需求，选择对应步骤为Lindorm实例开通自动续费。
   - 为单个实例开通自动续费
     a. 在**手动续费**或者**到期不续费**列表中，单击目标实例所在行**操作**列的**开通自动续费**。
     b. 在**开通自动续费**页面，选择自动续费周期。
     c. 单击**开通自动续费**。
   - 批量开通自动续费
     a. 在**手动续费**或者**到期不续费**列表中，勾选多个实例。
     b. 单击**手动续费**或者**到期不续费**列表左下角的**开通自动续费**。
     c. 在**开通自动续费**页面，选择自动续费周期。
     d. 单击**开通自动续费**。

相关任务

在续费管理控制台的**续费管理**页面，您还可以完成以下操作。

- 如果需要修改Lindorm实例自动续费的续费周期，请执行以下操作：
  单击**自动续费**页签，在**自动续费**列表中，单击目标实例所在行**操作**列的**修改自动续费**，重新设置自动续费周期。
- 如果需要为Lindorm实例手动续费，请执行以下操作：
  单击**手动续费**、**自动续费**或**到期不续费**页签，在对应列表中，单击目标实例所在行**操作**列的**续费**，选择购买时长为实例手动续费。
- 如果需要设置Lindorm实例为不续费，请执行以下操作：
  单击**手动续费**或**自动续费**页签，在对应列表中，单击目标实例所在行**操作**列的**不续费**，设置实例到期后不再续费。
- 如果需要关闭Lindorm实例的自动续费功能，请执行以下操作：
  单击**自动续费**或**到期不续费**页签，在对应列表中，单击目标实例所在行**操作**列的**恢复手动续费**，将自动续费修改为手动续费。

6.6.3. 手动续费包年包月实例

Lindorm实例到期没有及时续费会造成数据被删除，且删除的数据无法恢复，为防止数据丢失，建议您在实例到期后15天内完成手动续费或者开通自动续费。本文介绍为包年包月实例手动续费的方法。

前提条件

Lindorm实例的付费类型为包年包月。

手动续费方法

手动续费支持以下两种方法：

- 通过Lindorm控制台续费
- 通过续费管理控制台续费

通过Lindorm控制台续费

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页面，单击目标实例所在行**操作**列的**续费**。
4. 在**续费**页面，选择**购买时长**。
5. 阅读并勾选服务协议，单击**立即购买**。
6. 在**支付**页面，根据提示完成支付流程。
7. 支付完成后，单击**登录控制台**，进入云原生多模数据库Lindorm控制台。

通过续费管理控制台续费

1. 登录Lindorm管理控制台。
2. 在页面右上角，单击**费用 > 续费管理**。
3. 在**续费管理**页面，从**产品**下拉列表中选择云原生多模数据库Lindorm，单击**手动续费**页签。
4. 根据续费需求，选择对应步骤为Lindorm实例续费。
   - 为单个实例续费
     a. 在**手动续费**列表中，单击目标实例所在行**操作**列的**续费**。

     > **说明** 如果目标实例在**自动续费**或**到期不续费**页签中，您可以单击目标实例所在行**操作**列的**恢复手动续费**，在弹出的**恢复手动续费**对话框中单击**确认**，将实例的续费方式恢复为手动续费。

     b. 在**续费**页面，根据业务使用时间选择购买时长。
     c. 阅读并勾选服务协议，单击**立即购买**。
     d. 在**支付**页面，根据提示完成支付流程。
   - 批量续费
     a. 在**手动续费**列表中，勾选多个目标实例。
     b. 单击**手动续费**列表左下角的**批量续费**。
     c. 在**批量续费**页面，根据业务使用时间分别为每个实例选择续费时长。
     d. 单击**去支付**。
     e. 在**支付**页面，根据提示完成支付流程。

相关任务

为了避免因忘记手动续费而导致Lindorm实例服务中断，您可以为Lindorm实例开通自动续费，开通方法请参见开通自动续费。

6.7. 开通冷存储

云原生多模数据库Lindorm支持冷存储功能，可以将数据库中的冷数据存储至冷存储介质，降低存储成本。本文介绍如何开通冷存储。

使用场景

冷存储功能适用于数据归档、访问频率较低的历史数据等各种冷数据。

注意事项

- 冷存储功能仅支持云原生多模数据库Lindorm引擎版本为2.1.8及以上，如果数据库版本低于此版本，在开通冷存储过程中会自动升级至最新版本。
- 开通冷存储功能过程中业务访问可能发生抖动，建议您在业务低峰操作。

购买实例时开通冷存储

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 单击**创建**，开通冷存储需要设置以下参数，其他参数的设置请参见创建实例。
   - 选择**是否选择冷存储**参数为**是**。
   - 设置**冷存储存储容量**，冷存储的存储容量范围为800 GB~1000000 GB。
4. 单击**立即购买**。
5. 在**确认订单**页面，阅读并勾选服务协议，并单击**立即开通**。

购买实例后开通冷存储

如果您已经创建实例但是没有选择开通冷存储，可以通过Lindorm控制台开通。开通步骤如下：

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**冷存储**。
5. 单击**立即开通**。
6. 在**变配**页面设置以下参数。
   - 选择**是否选择冷存储**参数为**是**。
   - 设置**冷存储存储容量**，冷存储的存储容量范围为800 GB~1000000 GB。
7. 阅读并勾选服务协议，单击**立即购买**。

相关文档

- 配置冷存储
- 查看冷存储容量
- 扩容冷存储

7.宽表引擎（兼容HBase或Cassandra）

7.1. 引擎简介

Lindorm宽表引擎是面向海量半结构化、结构化数据设计的分布式存储，适用于元数据、订单、账单、画像、社交、feed流、日志等场景，兼容HBase、Phoenix（SQL）、Cassandra等开源标准接口，支持单表百万亿行规模、千万级并发、毫秒级响应、跨机房强一致容灾等，高效满足业务大规模数据的在线存储与查询需求。

核心能力

- **低成本**
  通过高密度低成本介质、智能冷热分离、高压缩比算法、自适应编码等技术，达到海量数据存储处理成本比自建低80%。
- **云原生弹性**
  通过存计分离架构和Serverless技术，支持计算和存储资源的按需即时弹性。
- **企业级稳定性**
  基于高可用架构设计，历经十余年阿里巴巴集团生产环境严苛考验，提供稳如磐石的企业级稳定性。
- **丰富生态**
  兼容HBase、Cassandra（CQL）等多种开源标准接口，与Spark、Flink等多种计算引擎互联互通，无缝对接主流数据生态。

关于Lindorm宽表引擎的更多内容，请参见功能特性和Lindorm VS HBase VS Cassandra。

典型场景

- 大数据场景：海量数据存储与分析
- 广告场景：海量广告营销数据的实时存储
- 金融&零售：海量订单记录与风控数据的实时存储
- 车联网：车辆轨迹与状况数据的高效存储处理
- 互联网社交：高效、稳定的社交Feed流信息存储

访问方式

- 通过Lindorm宽表SQL访问（推荐），开发体验与关系型数据库相似。
- 通过HBase API访问，适合已基于HBase开发的应用。
- 通过CQL访问，适合已基于Cassandra开发的应用
- Lindorm宽表支持多端同时访问，即对于同一张数据表，可以任意选择上述的其中一种方式进行访问。

7.2. 通过Lindorm宽表SQL连接宽表引擎

7.2.1. Lindorm宽表SQL Driver说明

Lindorm宽表SQL支持多种语言Driver，包括Java、Python、Go等，通过Lindorm宽表SQL多语言Driver可以访问Lindorm所有的功能。本节介绍Java和非Java的Driver使用参考文档。
关于Lindorm宽表SQL Java Driver安装和使用，请参见通过Lindorm宽表SQL Java API连接并使用宽表引擎。

关于Lindorm宽表SQL 非Java Driver安装和使用，请参见通过Lindorm宽表SQL 非Java API连接并使用宽表引擎。

关于Lindorm宽表SQL的语法使用，请参见Lindorm宽表SQL语法手册。

7.2.2. 访问实例

通过Lindorm宽表SQL客户端访问Lindorm实例需要的信息包括连接地址、用户名和密码。本节介绍获取连接地址和添加白名单操作。

获取连接地址

1. 登录Lindorm控制台。
2. 单击目标实例ID，进入**实例列表**页面。
3. 单击左侧导航栏**数据库连接**，默认**宽表引擎**页面，查看**Lindorm宽表SQL链接地址**，如下图所示。



说明 Lindorm实例支持专网访问（VPC）和公网访问。专网访问地址可直接获取，公网访问地址需要先开通外网地址。
Lindorm宽表SQL多语言客户端和Lindorm-cli客户端都使用相同的地址和端口。

添加白名单

为了连接安全，无论是公网访问还是私网访问，均需要将访问数据库的地址加入到白名单中。具体操作，请参见设置白名单。

7.2.3. 通过Lindorm宽表SQL Java API连接并使用宽表引擎

本文介绍通过Lindorm宽表SQL Java API连接并使用宽表引擎的具体操作和使用示例。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已获取Lindorm宽表SQL的连接地址并配置白名单，具体操作，请参见访问实例。

操作步骤

1. 下载Lindorm客户端。打开Eclipse客户端，创建一个Project并在*pom.xml*中配置Maven依赖，具体内容如下：

```
<dependency>
  <groupId>com.aliyun.lindorm</groupId>
  <artifactId>lindorm-all-client</artifactId>
  <version>2.0.14</version>
</dependency>
```

2. 初始化Lindorm客户端。在Project中输入下述代码，然后根据下表描述修改代码。

```
String LINDORM_JDBC_URL =  "Lindorm宽表SQL的连接地址";
String LINDORM_USER = "user";
String LINDORM_PASSWORD = "password";
String USER_NAME = "Lindorm控制台获取的数据库用户名";
String PASSWORD = "Lindorm控制台获取的数据库密码";
Properties properties = new Properties();
properties.put(LINDORM_USER, USER_NAME);
properties.put(LINDORM_PASSWORD, PASSWORD);
//如果需要建连的时候指定对应的database（namespace），默认链接default database
properties.put("database", 业务想指定的database);
```

参数	示例值	获取方法
LINDORM_JDBC_URL	jdbc:lindorm:table:url=http://ld-bp17j28j2y7pm****-proxy-lindorm-pub.lindorm.rds.aliyuncs.com:30060	Lindorm宽表SQL的连接地址，具体获取方法请参见访问实例。

参数	示例值	获取方法
USER_NAME	root	如果您忘记用户名密码，可以通过Lindorm宽表引擎的集群管理系统修改密码，具体操作请参见修改用户密码。
PASSWORD	root	
database	default	表示用户第一次连接Lindorm宽表时指定的database（namespace）。

3. 建立Lindorm客户端和数据之间的连接。

```
pconn = DriverManager.getConnection(LINDORM_JDBC_URL, properties);
```

> **说明**
> 服务端为了提高资源利用率，在连接Lindorm客户端空闲90 s后，服务端会主动断开连接。如果此时复用该连接，会出现com.aliyun.lindorm.client.shaded.org.apache.calcite.avatica.http.ConnectionDisconnectedException报错，重新建立连接即可解决。

4. 建立连接后，使用Lindorm宽表SQL Java API访问Lindorm宽表，以下提供访问示例代码。

> **说明**
> 默认使用的Schema都是default，如果想要访问指定的Schema，在访问前执行 `USE SCHEMA <Schema名>` 进行切换。

DDL操作：

```
//创建表
String tableName = "sql_table_" + new Random().nextInt(1000);
String sqlCreate = "create table if not exists " + tableName
        + "(id VARCHAR, name VARCHAR, primary key(id))";
 Statement statement = pconn.createStatement();
//如果不指定schema默认是"default",可以使用statement.execute("use schema名字"); 进行切换schema。
 int retCreate = statement.executeUpdate(sqlCreate);
 //删除表
 String sqlDelete = "drop table " + tableName;
 Statement deletestatement = pconn.createStatement();
 int retDel = deletestatement.executeUpdate(sqlDelete);
```

DML操作：

```
//使用PrepareStatement进行数据的插入，需要指定好对应的Primary Key
StringBuilder sqlBuilder = new StringBuilder();
sqlBuilder.append("upsert into " + tableName + "(id,name) values(?,?)");
String sqlUp = sqlBuilder.toString();
PreparedStatement psUp = pconn.prepareStatement(sqlUp);
psUp.setString(1, "aa");
psUp.setString(2, "bb");
int retupdate = psUp.executeUpdate();
//使用Sql进行数据的读取Select操作
String sqlSelect = "select * from " + tableName + " where id=?";
PreparedStatement psSelect = null;
psSelect = pconn.prepareStatement(sqlSelect);
psSelect.setString(1, "aa");
ResultSet rs = psSelect.executeQuery();
while (rs.next()) {
    String id = rs.getString(1);
    String name = rs.getString(2);
}
//删除数据
String sqlDeleteRow = "delete from " + tableName + " where id=?";
PreparedStatement psDeleteRow = pconn.prepareStatement(sqlDeleteRow);
psDeleteRow.setString(1, "aa");
psDeleteRow.executeUpdate();
//执行批量写入，lindorm的批量写入支持2种方式
//方式一：lindorm特有批量方式
String upsertSQL = "upsert into dt (id, name) values(?,?), (?,?), (?,?)";
PreparedStatement psBatchSql1 = pconn.prepareStatement(upsertSQL);
psBatchSql1.setString(1, "1");
psBatchSql1.setString(2, "2");
psBatchSql1.setString(3, "3");
psBatchSql1.setString(4, "4");
psBatchSql1.setString(5, "5");
psBatchSql1.setString(6, "6");
psBatchSql1.executeUpdate();
//方式二：标准JDBC方式
Connection connection = createConnection;
connection.setAutoCommit(false);
PreparedStatement ps = conn.prepareStatement("upsert into " + tableName1 + "(id, name) values(?,?)");
for (int i = 0; i < 20; i++) {
  ps.setString(i, "id" + i);
  ps.addBatch(i, "name");
}
ps.executeBatch();
conn.commit();
conn.setAutoCommit(true);
//注意，一批数据在executeBatch后需要主动调用链接地commit方法,这样数据才会真实的被写入数据库,否则只会缓存在内存里面
//如果链接地autocommit被设置为true，那么每次即使不用commit也会执行批量update。
```

相关文档

关于Lindorm宽表SQL的语法使用请参见Lindorm宽表SQL语法手册。

7.2.4. 通过Lindorm宽表SQL 非Java API连接并使用宽表引擎

本文介绍如何通过Lindorm宽表SQL 非Java API连接并使用宽表引擎的代码示例，以基于Python语言为例。

前提条件

- 已获取云原生多模数据库Lindorm宽表引擎的连接地址，具体操作请参见访问实例。
- 在使用Python连接Lindorm宽表之前，需要安装Python，Python的下载链接为Python。
- Python环境下安装依赖的phoenixdb：`pip install phoenixdb` 。

使用Python连接Lindorm宽表（phoenixdb版本以0.7为例）

在Python环境中复制以下代码，并修改其中的配置信息。

- 将 `localhost` 修改为Lindorm宽表引擎的连接地址，例如：`ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com` 。
- 将用户名和密码修改为Lindorm实例的用户名和用户名对应的密码，默认为root。如果您忘记了密码，可以在Lindorm宽表引擎的集群管理系统中重置密码，具体操作请参见修改用户密码。

```
import phoenixdb
database_url = "http://localhost:30060"
class PhoenixConnectionTest:
    def __init__(self):
        return
    def _connect(self, connect_kw_args):
        try:
            r = phoenixdb.connect(database_url, autocommit=True, **connect_kw_args)
            return r
        except AttributeError:
            print ("Failed to connect")
connect_kw_args = {'user': '用户名', 'password': '密码'}
db = PhoenixConnectionTest()
connection = db._connect(connect_kw_args)
with connection.cursor() as statement:
    sql_drop_table = "drop table if exists test_python"
    print (sql_drop_table)
    statement.execute(sql_drop_table)
    sql_create_table ="create table if not exists test_python(c1 int, c2 int, primary key(c1))"
    print (sql_create_table)
    statement.execute(sql_create_table)
    sql_upsert = "upsert into test_python(c1, c2) values(1,1)"
    print (sql_upsert)
    statement.execute(sql_upsert)
    sql_select = "SELECT * FROM test_python"
    print (sql_select)
    statement.execute(sql_select)
    rows = statement.fetchall()
    print (rows)
    statement.close()
connection.close
```

相关代码示例

- 有关Python语言访问Lindorm宽表的代码示例请参见Python演示程序。
- 有关Go语言访问Lindorm宽表的代码示例请参见Go演示程序。
- 有关Java语言访问Lindorm宽表的代码示例请参见Java演示程序。

语法参考

关于Lindorm宽表SQL的语法使用请参见Lindorm宽表SQL语法手册。

7.2.5. 通过Lindorm-cli连接并使用宽表引擎

本文介绍通过Lindorm-cli连接Lindorm宽表的具体操作和使用示例。

前提条件

已获取Lindorm宽表SQL的连接地址并配置白名单，具体操作，请参见访问实例。

连接Lindorm宽表使用的网络类型

网络类型	说明

网络类型	说明
专有网络（推荐）	专有网络VPC（Virtual Private Cloud）是您自己独有的云上私有网络，不同的专有网络之间二层逻辑隔离，拥有较高的安全性和性能。Lindorm-cli部署在ECS实例上时，通过专有网络连接至Lindorm宽表，可获得更高的安全性和更低的网络延迟。
公网	公网即互联网，当本地设备需要测试或管理Lindorm宽表时，可在本地设备上部署Lindorm-cli，然后通过公网连接至Lindorm宽表。 **说明** 通过公网连接不会产生流量费用，但存在一定的安全风险，推荐通过专有网络连接以获取更高的安全性。

常用命令

关于Lindorm宽表SQL的语法使用请参见Lindorm宽表SQL语法手册。

- **!help**：查看帮助命令。
- **!connect**：连接服务器命令。
- **exit**、**quit**和**ctrl+d**：退出当前连接。

操作步骤

1. 下载Lindorm-cli。您可以根据不同的客户端下载Lindorm-cli，下载链接如下表所示。

客户端	下载链接
Linux	lindorm-cli for linux
Mac	lindorm-cli for mac
Windows	lindorm-cli for windows-x64

2. 解压Lindorm-cli压缩包。
3. 获取连接信息并执行下述命令连接Lindorm宽表。

```
./lindorm-cli -url <jdbc url> -username <用户名> -password <密码>
```

参数	示例值	获取方法
jdbc url	jdbc:lindorm:table:url=http://ld-bp17j28j2y7pm****-proxy-lindorm-pub.lindorm.rds.aliyuncs.com:30060	Lindorm宽表SQL的连接地址，具体获取方法请参见访问实例。
用户名	root	如果您忘记用户名密码，可以通过Lindorm宽表引擎的集群管理系统修改密码，具体操作请参见修改用户密码。
密码	root	

返回如下结果：

```
Connected to jdbc:lindorm:table:url=http://ld-bp17j28j2y7pm****-proxy-lindorm-pub.lindorm.rds.aliyuncs.com:30060
lindorm-cli version: 1.0.15
```

4. 访问Lindorm宽表的语法示例如下。
   i. 执行以下语句创建名为test的DataBase。

```
create database test;
```

ii. 执行以下语句使用test数据库。

```
use test;
```

iii. 在test数据库中创建tb表格，并插入一条数据。

```
create table tb (id varchar, name varchar, address varchar,  primary key(id, name)) ;
upsert into tb (id, name, address) values ('001', 'jack',  'hz');
```

iv. 查询tb表格中的数据。

```
select * from tb;
```

返回如下结果：

```
+-----+-------+---------+
| id  | name  | address |
+-----+-------+---------+
| 001 | jack  | hz      |
+-----+-------+---------+
1 rows in set (24 ms)
```

7.2.6. 通过Java框架访问Lindorm宽表引擎示例

为了满足更多用户的需求，本文汇总了不同Java框架访问Lindorm宽表引擎的代码示例。

- 通过Druid框架访问Lindorm宽表SQL，代码示例请参见Druid访问示例。
- 通过Spring框架访问Lindorm宽表SQL，代码示例请参见Spring访问示例。
- 通过Mybatis框架访问Lindorm宽表SQL，代码示例请参见Mybatis访问示例。
- 通过Hibernate框架访问Lindorm宽表SQL，代码示例请参见Hibernate访问示例。

7.3. 使用开源客户端访问Lindorm宽表引擎

7.3.1. 通过HBase API连接并使用宽表引擎

7.3.1.1. HBase SDK 安装

Java SDK 安装请参见升级HBase Java SDK

Java API的使用和连接参数请参见通过HBase Java API连接并使用宽表引擎

非Java（C++/Python/Go等） SDK安装及访问请参见通过HBase非Java SDK访问Lindorm宽表引擎

7.3.1.2. 通过HBase Java API连接并使用宽表引擎

本文介绍通过HBase Java API访问Lindorm宽表引擎的具体操作和使用示例。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已获取Lindorm宽表引擎中**使用HBase Java API访问**的连接地址，具体操作请参见查看连接地址。
- 将客户端IP地址添加至云原生多模数据库Lindorm实例的白名单中，添加方法请参见设置白名单。
- 已完成HBase Java SDK的安装。具体操作请参见升级HBase Java SDK。

操作步骤

1. 配置客户端连接Lindorm实例。在Project中输入下述代码，然后根据下表描述配置参数。

```
// 新建一个Configuration
Configuration conf = HBaseConfiguration.create();
// 集群的连接地址（公网地址）在控制台页面的数据库连接界面获得
conf.set("hbase.zookeeper.quorum", "host:port");
// xml_template.comment.hbaseue.username_password.default
conf.set("hbase.client.username", "用户名");
conf.set("hbase.client.password", "密码");
// 如果您直接依赖了阿里云hbase客户端，则无需配置connection.impl参数，如果您依赖了alihbase-connector，则需要配置此参数
//conf.set("hbase.client.connection.impl", AliHBaseUEClusterConnection.class.getName());
```

参数	示例值	获取方法
host:port	ld-bp17j28j2y7pm****-proxy-lindorm-pub.lindorm.rds.aliyuncs.com:30020	Lindorm宽表引擎中**使用HBase Java API访问**的连接地址，具体操作请参见查看连接地址。
用户名	root	如果您忘记用户名密码，可以通过Lindorm宽表引擎的集群管理系统修改密码，具体操作请参见修改用户密码。
密码	root	

> **注意** 如果您依赖了老版本的alihbase-connector（版本号小于1.0.9或者2.0.9），请参考老版本的参数配置。

2. 建立Lindorm客户端和数据之间的连接。

```
Connection connection = ConnectionFactory.createConnection(conf);
```

> **说明** 在程序生命周期内只需创建一次，该连接线程安全，可以共享给所有线程使用。在程序结束后，需要将Connection对象关闭，否则会造成连接泄露。也可以采用try finally方式防止泄露。

3. 连接建立后，可以使用HBase Java API访问Lindorm宽表。下面提供一些简单的Java示例。
   - DDL操作

```
try (Admin admin = connection.getAdmin()){
    // 建表
    HTableDescriptor htd = new HTableDescriptor(TableName.valueOf("tablename"));
    htd.addFamily(new HColumnDescriptor(Bytes.toBytes("family")));
    // 创建一个只有一个分区的表
    // 在生产上建表时建议根据数据特点预先分区
    admin.createTable(htd);
    // disable表
    admin.disableTable(TableName.valueOf("tablename"));
    // truncate表
    admin.truncateTable(TableName.valueOf("tablename"), true);
    // 删除表，删除表之前需要先执行DISABLE语句。
    admin.deleteTable(TableName.valueOf("tablename"));
}
```

   - DML操作

```
//Table为非线程安全对象，每个线程在对Table操作时，都必须从Connection中获取相应的Table对象
try (Table table = connection.getTable(TableName.valueOf("tablename"))) {
    // 插入数据
    Put put = new Put(Bytes.toBytes("row"));
    put.addColumn(Bytes.toBytes("family"), Bytes.toBytes("qualifier"), Bytes.toBytes("value"));
    table.put(put);
    // 单行读取
    Get get = new Get(Bytes.toBytes("row"));
    Result res = table.get(get);
    // 删除一行数据
    Delete delete = new Delete(Bytes.toBytes("row"));
    table.delete(delete);
    // scan 范围数据
    Scan scan = new Scan(Bytes.toBytes("startRow"), Bytes.toBytes("endRow"));
    ResultScanner scanner = table.getScanner(scan);
    for (Result result : scanner) {
        // 处理查询结果result
        // ...
    }
    scanner.close();
}
```

7.3.1.3. 通过HBase非Java API连接并使用宽表引擎

云原生多模数据库Lindorm宽表引擎支持通过HBase非Java（例如C++、Python和Go等）API进行访问，本文介绍具体的访问操作。

准备工作

- 已下载Thrift安装包，下载链接请单击Thrift安装包。
- 已下载HBase Thrift2定义文件，下载链接请单击HBase Thrift2定义文件。
- 已获取Lindorm宽表引擎的**使用HBase 非Java API访问**的连接地址，具体操作请参见查看连接地址。



访问Lindorm宽表引擎（以Python语言为例）

Thrift安装包的使用请参见Apache Thrift官方文档，通过Thrift访问云原生多模数据库Lindorm宽表引擎的步骤如下：

1. 使用HBase Thrift2定义文件来生成对应语言的接口文件。 命令语句如下：

```
thrift --gen <language> Hbase.thrift
```

> **说明** `language` 为目标语言，例如：python、php、cpp、py等。

语句示例：

```
thrift --gen python Hbase.thrift
```

2. 构造客户端访问云原生多模数据库Lindorm宽表引擎。
   Lindorm中Thrift服务器端的transport层使用的是HTTP，因此在构造客户端时，需要thrift中的ThttpClient（各个语言都有相应实现）。并且在ACL开启的情况下，需要在ThttpClient上加上两个header来向服务器传输用户名和密码进行认证（如果关闭ACL则不需要）。Thrift在每个语言实现的ThttpClient都有加定制header的函数。以Python语言为例，使用以下语句构造客户端并访问Lindorm宽表引擎。

```
# -*- coding: utf-8  -*-
# 以下两个模块通过执行pip install thrift 语句生成
from thrift.protocol import TBinaryProtocol
from thrift.transport import THttpClient
# 以下两个模块通过执行thrift --gen py hbase.thrift 语句生成
from hbase import THBaseService
from hbase.ttypes import TColumnValue, TColumn, TTableName, TTableDescriptor, TColumnFamilyDescriptor,
TNamespaceDescriptor, TGet, TPut, TScan
# 配置Lindorm宽表引擎的连接地址
url = "http://ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com:9190"
transport = THttpClient.THttpClient(url)
headers = {}
# 设置用户名
headers["ACCESSKEYID"]="root";
# 设置用户名对应的密码
headers["ACCESSSIGNATURE"]="root"
transport.setCustomHeaders(headers)
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)
client = THBaseService.Client(protocol)
transport.open()
# 具体操作后，执行关闭连接操作
transport.close()
```

相关代码示例

其他非Java语言的完整代码示例请直接下载Github相应语言的代码示例，下载链接如下所示。

- Python
- Go
- C++
- node.js
- PHP

7.3.1.4. 通过HBaseue Shell连接并使用宽表引擎

本文介绍通过HBaseue Shell连接并使用Lindorm宽表引擎。

前提条件

已创建云原生多模数据库Lindorm实例。

注意事项

通过HBaseue Shell连接并使用Lindorm宽表引擎，强烈建议在客户端部署在Linux或者Mac环境。如果部署在Windows环境，可能会遇到报错或缺少库的问题，您需要根据报错信息自行在系统中添加相应的库。

准备工作

- 如果应用部署在ECS，您需要确保云原生多模数据库Lindorm实例和ECS实例满足以下条件，以保证网络的连通性。ECS实例信息的查看方法请参见查看实例信息。
  - 所在地域相同，并建议所在可用区相同（以减少网络延时）。

- 网络类型相同。

  > **说明**
  > - 专有网络具有更高的安全性，建议使用专有网络。
  > - 如果网络类型为**专有网络**，需确保使用相同的专有网络ID。

- 将客户端IP地址添加至云原生多模数据库Lindorm实例的白名单中，添加方法请参见设置白名单。
- 已获取Lindorm宽表引擎的**使用HBase Java API访问**连接地址，具体操作请参见查看连接地址。

操作步骤

1. 下载HBaseue Shell工具。

   > **说明** 您可以通过HBase官网下载所需版本的tar包，然后导入JAR文件升级客户端，具体操作请参见通过导入JAR文件升级HBase客户端。

   i. 登录Lindorm管理控制台。
   ii. 在页面左上角，选择实例所属的地域。
   iii. 在**实例列表**页，单击目标实例ID。
   iv. 在左侧导航栏单击**数据库连接**，单击**宽表引擎**页签。
   v. 在**HBase兼容地址**区域单击**Lindorm Shell下载**。

2. 执行以下命令解压HBaseue Shell压缩包，将压缩包解压至目标文件（以*alihbase-2.0.18*文件为例）。

   ```
   tar zxvf hbaseue-shell.tar.gz
   ```

3. 配置连接参数。

   i. 进入*alihbase-2.0.18/conf*目录，打开*hbase-site.xml*文件。

   ```
   vi hbase-site.xml
   ```

   ii. 配置Lindorm实例的连接地址和用户名密码。

   ```
   <configuration>
       <property>
           <name>hbase.zookeeper.quorum</name>
           <value>ld-bp17j28j2y7pm****-proxy-lindorm-pub.lindorm.rds.aliyuncs.com:30020</value>
       </property>
       <property>
           <name>hbase.client.username</name>
           <value>root</value>
       </property>
       <property>
           <name>hbase.client.password</name>
           <value>root</value>
       </property>
   </configuration>
   ```

   > **说明**
   > - hbase.zookeeper.quorum：填写为Lindorm宽表引擎的**使用HBase Java API访问**连接地址。
   > - hbase.client.username和hbase.client.password：填写访问Lindorm宽表的用户名和密码，如果您忘记密码可以通过宽表引擎的**集群管理系统**修改密码，具体操作请参见修改用户密码。

4. 通过HBaseue Shell访问Lindorm宽表引擎。进入*alihbase-2.0.18/bin*目录执行以下命令。

   ```
   ./hbase shell
   ```

   返回如下信息表示成功访问。

```
Version 2.0.18, r08b8d58a9d6ce89765d5ebe2ddff425aed644c16, Mon Feb  1 12:46:39 CST 2021
Took 0.0034 seconds
```

说明

- Shell的使用方法可以参见HBase Shell使用介绍。
- 通过HBaseue Shell连接Lindorm宽表引擎仅支持简单的DDL操作和数据读写操作，详情请参见使用限制。

7.3.1.5. 使用限制

Lindorm支持通过HBase Java API进行访问，但目前存在以下几个方面的使用限制。

- 不支持HBase的系统管理API：Lindorm不支持HBase的集群管理相关操作的接口，如assign region、stopRegionServer等等，用户对集群的日常管理，可以通过控制台和我们提供的集群管理系统完成。
- 暂不支持Coprocessor：计划后续支持，如果有该需求的客户可以钉钉咨询（参考技术支持）。
- 有限开放Lindorm底层存储（HDFS协议兼容）：如果有数据导入导出需求，请参见数据导入与迁移。如果需要使用Spark分析，请参见通过Spark访问Lindorm或者直接在Spark中调用HBase API或者使用MR的TableInputFormat。**但支持开放HDFS来进行bulkload操作。**
- 不支持用户自定义的Filter：自定义Filter指的是用户将自身实现的Filter类代码封装为JAR包，上传到服务器的lib路径下或者HDFS中，并重启服务或者动态加载，使之生效。Lindorm不支持此使用方式。

如对以上的使用限制存在疑问或需求，欢迎提工单，或者加入钉群咨询（参考技术支持）。

7.3.2. 通过Cassandra CQL连接并使用宽表引擎

7.3.2.1. Cassandra CQL Driver 安装

Cassandra CQL 支持多种语言的driver，包括并不限于Java、C++、C#、Nodejs、PHP、Python、Spark Connector等，通过Cassandra CQL 多语言Driver可以完全兼容的访问Lindorm，此处介绍常见的Java、Python、C++三种语言的Driver安装以及使用，其他多语言的Driver可以借鉴Cassandra 社区关于CQL多语言Driver
Cassandra CQL Java Driver 安装请参见 通过Cassandra CQL Java Driver连接并使用宽表引擎

Cassandra CQL Java Driver 使用和连接参数请参见通过Cassandra CQL Java Driver连接并使用宽表引擎

Cassandra CQL 多语言（Python、C++）Driver安装以及访问请参见通过Cassandra CQL多语言Driver连接并使用宽表引擎

7.3.2.2. 连接集群

通过Cassandra CQL Driver访问Lindorm需要三个信息：链接地址、用户名和密码，您可以通过控制台获得。

连接地址

Lindorm支持通过内网（VPC）和公网进行访问，两者使用不同的地址域名，多语言客户端以及cqlsh都使用相同的端口（默认9042），用户可以在集群的控制台中的“数据库连接”页面中看到连接地址，如果是公网访问需要先开通外网地址。

添加白名单

实际访问之前，请确保已经将客户机的IP加入白名单即：获取访问的源地址IP通过控制台的入口添加信任白名单。

7.3.2.3. 通过Lindorm-cqlsh连接并使用宽表引擎

Lindorm-cqlsh是云原生多模数据库Lindorm团队基于Cassandra cqlsh修改后提供的Shell工具，提供Windows版本和Unix版本。您可以在本地或ECS上安装Lindorm-cqlsh工具，通过Lindorm-cqlsh连接并访问Lindorm宽表引擎。

前提条件

- 已将客户端的IP地址加入到Lindorm实例的白名单中，具体操作请参见设置白名单。
- 已获取云原生多模数据库Lindorm宽表引擎的**CQL连接**地址，具体操作请参见查看连接地址。



- 已下载并解压Lindorm-cqlsh，下载链接为下载Lindorm-cqlsh。
  如果应用部署在ECS实例，您需要确保云原生多模数据库Lindorm实例和ECS实例满足以下条件，以保证网络的连通性。ECS实例信息的查看方法请参见查看实例信息。
  - 所在地域相同，并建议所在可用区相同（以减少网络延时）。
  - 网络类型相同。

基本命令

访问Lindorm宽表支持以下Lindorm-cqlsh命令。

```
Documented shell commands:===========================
CAPTURE  COPY  DESCRIBE  LOGIN   DESC  EXIT  HELP PAGING  SHOW
CQL help topics:================
CREATE_KEYSPACE        TEXT                ALTER_KEYSPACE           TIME       CREATE_ROLE
DROP_USER              TIMESTAMP           ALTER_TABLE              CREATE_TABLE
GRANT                  ALTER_USER          INSERT                   UPDATE
CREATE_USER            INSERT_JSON         USE                      ASCII
DATE                   INT                 UUID                     BATCH
DELETE                 JSON                BEGIN                    KEYWORDS
BLOB                   DROP_COLUMNFAMILY   LIST_PERMISSIONSBOOLEAN  LIST_ROLES
COUNTER                DROP_INDEX          LIST_USERS               DROP_KEYSPACE
PERMISSIONS            CREATE_COLUMNFAMILY REVOKE                   DROP_ROLE
SELECT                 CREATE_INDEX        DROP_TABLE                SELECT_JSON
```

操作步骤

1. 执行以下命令将Lindorm-cqlsh与云原生多模数据库Lindorm连接。

   ```
   bin/cqlsh <host> <port> -u <username> -p <password>
   ```

   说明
   - <host>：在云原生多模数据库Lindorm控制台上获取的宽表引擎**CQL连接**地址，例如 `ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com` 。
   - <port>：云原生多模数据库Lindorm宽表引擎的**CQL连接**地址端口号，默认为9042。
   - <username>：Lindorm宽表引擎的用户名，默认为root。
   - <password>：Lindorm宽表引擎用户名对应的密码，如果您忘记密码，可以通过集群管理系统重置密码，具体操作请参见修改用户密码。
   - 云原生多模数据库Lindorm支持多种Lindorm-cqlsh的访问命令，更多信息您可以通过 `bin/cqlsh -help` 命令查看。

2. 通过Lindorm-cqlsh创建keyspace。Lindorm中的keyspace与关系型数据库中的database概念类似，一个keyspace可以包含

一个或多个表格或列簇。

```
CREATE KEYSPACE test_keyspace WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

创建成功后执行以下语句查看keyspace：

```
DESCRIBE KEYSPACE test_keyspace;
```

3. 使用USE命令切换至目标keyspace。

```
USE test_keyspace;
```

返回以下信息：

```
cqlsh:test_keyspace>
```

4. 通过Lindorm-cqlsh访问Lindorm宽表的代码示例如下：
   - 通过Lindorm-cqlsh创建表格test_user。表格中包含first_name和last_name两个字段，字段的数据类型为text，并且first_name是这张表的PRIMARY KEY。

     ```
     CREATE TABLE test_user(first_name text, last_name text, PRIMARY KEY (first_name));
     ```

     说明 创建表格操作前如果您没有执行 `USE test_keyspace;` 命令，可以通过以下语句创建表格。

     ```
     CREATE TABLE test_keyspace.test_user(first_name text, last_name text, PRIMARY KEY (first_name));
     ```

   - 查看表test_user的详细信息。

     ```
     DESCRIBE TABLE test_user;
     ```

     返回以下信息：

     ```
     CREATE TABLE test_keyspace.test_user (
         first_name text PRIMARY KEY,
         last_name text
     ) WITH bloom_filter_fp_chance = 0.01
         AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
         AND comment = ''
         AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max
     _threshold': '32', 'min_threshold': '4'}
         AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compr
     essor'}
         AND crc_check_chance = 1.0
         AND dclocal_read_repair_chance = 0.1
         AND default_time_to_live = 0
         AND gc_grace_seconds = 864000
         AND max_index_interval = 2048
         AND memtable_flush_period_in_ms = 0
         AND min_index_interval = 128
         AND read_repair_chance = 0.0
         AND speculative_retry = '99PERCENTILE';
     ```

     说明 DESCRIBE TABLE命令将创建表语句以格式化的形式显示出来，除了您制定的配置项，还包含一些默认的配置项。Lindorm表级别的配置和Cassandra部分属性是不一样的，对于不一样的属性使用Cassandra CQL默认配置展示。
     - Lindorm表级别无使用意义的配置项为：`crc_check_chance`、`gc_grace_seconds`、`read_repair_chance`、`speculative_retry`、`dclocal_read_repair_chance`、`crc_check_chance`。
     - 完全兼容的配置项为：`compression`、`default_time_to_live`。

   - 在表test_user中插入数据。

```
INSERT INTO test_user (first_name, last_name) VALUES ('test', 'LINDORM');
INSERT INTO test_user (first_name, last_name) VALUES ('Zhang', 'San');
INSERT INTO test_user (first_name) VALUES ('Wu');
```

- 查看数据是否成功插入。以下命令不建议在海量数据场景中使用。

```
SELECT COUNT(*) FROM test_user;
```

返回以下信息：

```
  count
-------
    3
(1 rows)
```

也可以通过以下命令查看插入的具体数据。

```
SELECT * FROM test_user;
```

返回以下信息：

```
first_name | last_name
------------+-----------
       test |    LINDORM
         Wu |      null
      Zhang |       San
(3 rows)
```

- 删除表test_user中的指定行的last_name列的值，然后查询删除后的表数据。

```
DELETE last_name FROM test_user WHERE first_name='test';
SELECT * FROM test_user WHERE first_name='test';
```

返回以下信息：

```
first_name | last_name
------------+-----------
       test |    null
```

- 删除表test_user中指定行的数据，然后查询删除后的表数据。

```
DELETE FROM test_user WHERE first_name='test';
SELECT * FROM test_user WHERE first_name='test';
```

返回以下信息：

```
first_name | last_name
------------+-----------
(0 rows)
```

- 清空表或者删除表。只有super用户才能执行 `TRUNCATE、DROP TABLE、DROP KEYSPACE` 命令。

```
TRUNCATE test_user;
DROP TABLE test_user;
```

- 使用Lindorm SEARCH INDEX。
通过CQL为Lindorm宽表创建搜索索引，创建搜索索引前需确保表的属性满足创建需求，相关需求请提交工单咨询值班。

```
CREATE TABLE test_keyspace.tb (cn1 text PRIMARY KEY, cn2 text, cn3 text) WITH extensions = {'CONSISTEN
CY_TYPE':'strong', 'MUTABILITY':'MUTABLE_LATEST'};
CREATE SEARCH INDEX tbidx ON test_keyspace.tb WITH COLUMNS (cn2, cn3);
REBUILD SEARCH INDEX ON test_keyspace.tb ;
INSERT INTO test_keyspace.tb (cn1, cn2, cn3) VALUES ('v11', 'v12', 'v13');
SELECT * FROM test_keyspace.tb WHERE cn2 like '%v1';
```

> **说明** 上述代码表示在表 `tb` 中创建索引表 `tbidx` ，并写入数据，然后进行模糊查询。相关语法请参见管理搜索索引。

- 使用Lindorm Secondary INDEX。
  通过CQL为Lindorm宽表创建Lindorm Secondary Index，创建前需确保表的属性满足创建需求，相关信息请参考高性能原生二级索引。

```
CREATE TABLE test_keyspace.tb (cn1 text PRIMARY KEY, cn2 text, cn3 text) WITH extensions = {'CONSISTEN
CY_TYPE':'strong', 'MUTABILITY':'MUTABLE_LATEST'};
CREATE INDEX tbidx ON test_keyspace.tb (cn2);
INSERT INTO test_keyspace.tb (cn1, cn2, cn3) VALUES ('v11', 'v12', 'v13');
SELECT * FROM test_keyspace.tb WHERE cn2 = 'v12';
```

7.3.2.4. 通过Cassandra CQL Java Driver连接并使用宽表引擎

本文介绍通过Cassandra CQL Java Driver连接并使用Lindorm宽表引擎。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已完成Cassandra CQL Java Driver的安装，具体操作请参见Cassandra CQL Java Driver 安装和升级。
- 已将客户端的IP地址加入到Lindorm实例的白名单中，具体操作请参见设置白名单。
- 已获取云原生多模数据库Lindorm宽表引擎的**CQL连接**地址，具体操作请参见查看连接地址。



操作步骤

1. 配置连接参数。

```
String[] contactPoints = new String[]{
     "<host>"
};
Cluster cluster = Cluster.builder()
     .addContactPoints(contactPoints)
     .withAuthProvider(new PlainTextAuthProvider(username, password))
     .build();
cluster.init();
Session session = cluster.connect();
```

> **说明**
> - `<host>`：通过云原生多模数据库Lindorm控制台获取宽表引擎的**CQL连接**地址，例如：`ld-bp17j28j2y7pm*****-proxy-lindorm.lindorm.rds.aliyuncs.com`。
> - `username`：云原生多模数据库Lindorm实例的账号名，默认为root。
> - `password`：云原生多模数据库Lindorm实例的账号名对应的密码，如果您忘记密码可以通过宽表引擎的**集群管理系统**修改密码，具体操作请参见修改用户密码。

2. 使用Cassandra CQL Java API访问Lindorm宽表，以下提供访问示例代码。
   - DDL操作

```
// 创建keyspace，指定对应strategy， replication factor。
   session.execute(
               "CREATE KEYSPACE IF NOT EXISTS testKeyspace WITH replication "
                       + "= {'class':'SimpleStrategy', 'replication_factor':1};");
// 创建table，给table指定对应的primary key 以及cluster key 和regular key
   session.execute(
               "CREATE TABLE IF NOT EXISTS testKeyspace.testTable ("
                       + "id int PRIMARY KEY,"
                       + "name text,"
                       + "age int,"
                       + "address text"
                       + ");");
//清空表
   session.execute("TRUNCATE TABLE testKeyspace.testTable;");
//删除表
   session.execute("DROP TABLE testKeyspace.testTable ");
```

   - DML操作

```
   // 执行insert 操作
   session.execute(
               "INSERT INTO testKeyspace.testTable (id, name, age, address) "
                       + "VALUES ("
                       + "1,"
                       + "'testname',"
                       + "11,"
                       + "'hangzhou');");
   // 执行select 操作，这里select * 表示获取所有列，也可以指定需要select 的列名获取对应列数据
   ResultSet res = session.execute(
               "SELECT * FROM testKeyspace.testTable ;");
   // 如果想要获取每一列对应的数据，可以如下操作
   for (Row row : results)
   {
       int id = row.getInt("id");
       String name = row.getString("name");
       int age = row.getInt("age");
       String address = row.getString("address");
   }
   // 关闭Session
   session.close();
   // 关闭Cluster
   cluster.close();
```

7.3.2.5. 通过Cassandra CQL多语言Driver连接并使用宽表引擎

本文介绍多语言Driver（例如C++、Python、Nodejs和Go）通过Cassandra CQL访问云原生多模数据库Lindorm宽表引擎的操作方法。

前提条件

- 已获取Lindorm宽表引擎的CQL连接地址，具体操作请参见查看连接地址。



- 根据您使用的语言下载并安装驱动程序。更多信息请参见Cassandra client drivers。

Cassandra CQL Python Driver访问Lindorm宽表

1. 安装Datastax Python依赖包，具体操作请参见安装Python SDK。

```
# 指定版本安装（建议安装3.x版本）
pip install cassandra-driver==3.19.0
# 安装最新版本
pip install cassandra-driver
```

2. 访问Lindorm宽表的示例代码。

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
import logging
import sys
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
logging.basicConfig(stream=sys.stdout, level=logging.INFO)
cluster = Cluster(
    # 填写Lindorm宽表引擎的CQL连接地址，不用填端口
    contact_points=["ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com"],
    # 填写用户名和密码，默认为root
    auth_provider=PlainTextAuthProvider("cassandra", "****"))
session = cluster.connect()
# 创建keyspace
session.execute(
                "CREATE KEYSPACE IF NOT EXISTS testKeyspace WITH replication = {'class':'SimpleStrategy'
, 'replication_factor':1};");
# 创建table
session.execute(
                "CREATE TABLE IF NOT EXISTS testKeyspace.testTable (id int PRIMARY KEY, name text,age in
t,address text);");
# 插入数据
session.execute(
                "INSERT INTO testKeyspace.testTable (id, name, age, address) VALUES ( 1, 'testname', 11,
'hangzhou');");
# 查询数据
rows = session.execute(
                "SELECT * FROM testKeyspace.testTable ;");
# 打印每行数据到控制台
for row in rows:
    print("# row: {}".format(row))
# 关闭Session
session.shutdown()
# 关闭
cluster.shutdown()
```

Cassandra CQL C++ Driver访问Lindorm宽表

1. 下载C++依赖包，下载链接请参见Datastax c++。
2. 访问Lindorm宽表的示例代码。

```
CassFuture* connect_future = NULL;
CassCluster* cluster = cass_cluster_new();
CassSession* session = cass_session_new();
//填写Lindorm宽表引擎的CQL连接地址，不用填端口
char* hosts = "ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com";
//建立连接
cass_cluster_set_contact_points(cluster, hosts);
connect_future = cass_session_connect(session, cluster);
//DDL操作
if (cass_future_error_code(connect_future) == CASS_OK) {
    CassFuture* close_future = NULL;
    const char* query = "SELECT  name FROM testKeyspace.testTable ";
    CassStatement* statement = cass_statement_new(query, 0);
    CassFuture* result_future = cass_session_execute(session, statement);
    if (cass_future_error_code(result_future) == CASS_OK) {
      //获取查询结果
      const CassResult* result = cass_future_get_result(result_future);
      const CassRow* row = cass_result_first_row(result);
      if (row) {
        const CassValue* value = cass_row_get_column_by_name(row, "name");
        //打印结果
        const char* name;
        size_t name_length;
        cass_value_get_string(value, &name, &name_length);
        printf("release_version: '%.*s'\n", (int)name_length, name);
      }
      cass_result_free(result);
    } else {
      //异常处理
      const char* message;
      size_t message_length;
      cass_future_error_message(result_future, &message, &message_length);
      fprintf(stderr, "Unable to run query: '%.*s'\n", (int)message_length, message);
    }
    cass_statement_free(statement);
    cass_future_free(result_future);
    //手动释放资源信息
    close_future = cass_session_close(session);
    cass_future_wait(close_future);
    cass_future_free(close_future);
  } else {
    //异常处理
    const char* message;
    size_t message_length;
    cass_future_error_message(connect_future, &message, &message_length);
    fprintf(stderr, "Unable to connect: '%.*s'\n", (int)message_length, message);
  }
  cass_future_free(connect_future);
  cass_cluster_free(cluster);
  cass_session_free(session);
```

Cassandra CQL Nodejs Driver访问Lindorm宽表

1. 安装Node.js依赖包。

```
npm install cassandra-driver
```

2. 访问Lindorm宽表的示例代码。

```
const cassandra = require('cassandra-driver');
/**
* 填写Lindorm宽表引擎的CQL连接地址，不用填端口。dc名字填datacenter1
* 填写用户名和密码，默认为root
*/
const client = new cassandra.Client({ contactPoints: ['ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com'], localDataCenter: 'datacenter1', credentials: { username: '用户名', password: '密码' } });
client.connect()
.then(() => client.execute("CREATE KEYSPACE IF NOT EXISTS lindormtest WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '2' }"))
.then(() => client.execute("CREATE TABLE IF NOT EXISTS lindormtest.nodejs (name text PRIMARY KEY, age int)"))
.then(() => {
 return client.execute("INSERT INTO lindormtest.nodejs (name, age) VALUES ('lindorm', 10)");
})
.then(() => {
 return client.execute("SELECT name, age FROM lindormtest.nodejs WHERE name = 'lindorm' ");
})
.then(result => {
 const row = result.first();
 console.log('Obtained row: ', row);
 const p = row.age;
})
.finally(() => client.shutdown());
```

Cassandra CQL go Driver访问Lindorm宽表

1. 安装GoCQL。

```
go get github.com/gocql/gocql
```

2. 访问Lindorm宽表的示例代码。

```
package main
import (
    "fmt"
    "log"
    "github.com/gocql/gocql"
)
func main() {
    // 填写Lindorm宽表引擎的CQL连接地址，不用填端口
    cluster := gocql.NewCluster("ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com")
    cluster.Authenticator = gocql.PasswordAuthenticator{
        Username:"用户名",
        Password: "密码",
    }
    session, _ := cluster.CreateSession()
    defer session.Close()
    //create keyspace
    if err := session.Query(`CREATE KEYSPACE ks WITH replication = {'class': 'SimpleStrategy', 'replicat
ion_factor': 2}`).Exec(); err != nil {
        log.Fatal(err)
    }
    // create table
    if err := session.Query(`CREATE TABLE ks.tb (cn1 text PRIMARY KEY, cn2 text, cn3 text)`).Exec(); err
!= nil {
        log.Fatal(err)
    }
    // insert value
    if err := session.Query(`INSERT INTO ks.tb (cn1, cn2, cn3) VALUES (?, ?, ?)`,
        "v11", "v12", "v13").Exec(); err != nil {
        log.Fatal(err)
    }
    var column1 string
    var column2 string
    if err := session.Query(`SELECT cn1, cn2 FROM ks.tb WHERE cn1 = ?`,
        "v11").Consistency(gocql.One).Scan(&column1, &column2); err != nil {
        log.Fatal(err)
    }
    fmt.Println("ALL values:", column1, column2)
    var column3 string
    // list all tweets
    iter := session.Query(`SELECT * FROM ks.tb WHERE cn1 = ? `, "v11").Iter()
    for iter.Scan(&column1, &column2, &column3) {
        fmt.Println("ALL value:", column1, column2, column3)
    }
    if err := iter.Close(); err != nil {
        log.Fatal(err)
    }
}
```

7.3.2.6. 使用限制

云原生多模数据库Lindorm高度兼容Cassandra CQL，但是有一些语法存在使用限制。

- 不支持UDT/UDF/UDA。
- 不支持Trigger功能。
- 不支持SASI索引功能。
- 不支持范围delete。
- 不支持BATCH 操作。
- 部分数据类型内测中，暂不支持：集合数据类型、DURATION、COUNTER。
- 暂不支持客户端使用tokenaware policy进行访问。
- Cassandra支持的是微秒时间戳，Lindorm中支持的是毫秒时间戳，如果数据从开源Cassandra中迁移至Lindorm宽表，请勿将数据的writeTime写入Lindorm中。由于微秒时间戳的值远大于毫秒，Lindorm会把这个时间当做一个未来的时间处理，会造成

TTL、数据删除、更新等一系列问题。

- 其他限制可咨询Lindorm售后。

7.3.2.7. Lindorm CQL语法介绍

7.3.2.7.1. 名词解释

本文档主要介绍了Lindorm CQL语法中的常见术语。

名词	对应Lindorm中的含义	描述
keyspace	namespace	键值空间，keyspace逻辑包含很多表，键值空间定义的属性适用于所有表。
table	table	表，构建表的时候需要定义表的schema结构，包括表包含的列以及列的类型。
PRIMARY KEY	PRIMARY KEY	主键，用于标识一行数据，可以唯一确定一行数据的分布位置，在Lindorm CQL中由partition key和cluster column，共同组成primary key。
partition key	PRIMARY KEY的组成部分	和cluster column共同组成primary key，在Lindorm CQL中与Cassandra CQL的身份定义不同。
cluster column	PRIMARY KEY的组成部分	和partition key共同组成primary key。
regular colum	COLUMN	非主键列。
secondary index	secondary index	二级索引，能够满足基于二级索引的加速查询。
search index	search index	Lindorm CQL支持的search index，能够支持基于search index的多维查询，统计分析，模糊匹配等特性。

7.3.2.7.2. 数据类型

Lindorm CQL是一种类型化语言，支持丰富的数据类型。本文档主要介绍了Lindorm CQL支持的数据类型。

基础数据类型

数据类型	支持的常量	描述
bigint	integer	64位有符号长整数
blob	blob	任意长度的字节
boolean	boolean	取值：true或者false
counter	integer	Counter列（64位有符号值）
date	integer, string	日期时间
decimal	integer, float	可变精度十进制
double	integer, float	64位IEEE-754浮点数
float	integer, float	32位IEEE-754浮点数
inet	string	IP地址，可以为IPv4和IPv6。由于没有IP类型，所以输入为String进行存储。

数据类型	支持的常量	描述
int	integer	32位有符号整数
smallint	integer	16位有符号整数
text	string	UTF-8编码的字符串
time	integer, string	时间类型，纳秒精度
timestamp	integer, string	时间戳，精度是毫秒
timeuuid	uuid	版本1 UUID，通常用作无冲突时间戳
tinyint	integer	8位无符号整数
uuid	uuid	一个UUID
varint	integer	任意精度整数
nested	string	JSON字符串的嵌套类型

Counter

计数器类型用于定义计数器列。Lindorm的counter和Cassandra的counter是有一定区别的，Cassandra的counter表只允许包含counter列，但是Lindorm CQL的表允许counter列和非counter列共存。示例如下：

```
CREATE TABLE person_info  (name text primary key, age counter, address text);
UPDATE person_info SET age = age + 2 WHERE name = 'my';
INSERT INTO person_info  (name, age, address) VALUES ( 'my', 30, 'hz');
```

Timestamp

允许输入int数据类型的数据以及string类型的字符串，用来表示某一个时间点。下面的示例用来表示：2011年03月2号04:05:00AM GMT。

```
CREATE TABLE person_info ( name text PRIMARY KEY , birthtime timestamp , address text );
INSERT INTO person_info (name, birthtime, address) VALUES ( 'my', 1299038700000, 'hz');
INSERT INTO person_info (name, birthtime, address) VALUES ( 'mm', '2011-03-02 04:05+0000', 'hz');
INSERT INTO person_info (name, birthtime, address) VALUES ( 'lucy', '2011-03-02 04:05:00+0000', 'bj');
```

Time

支持int以及string表示times的数据类型，如果是string表示时间的话，格式为 `hh:mm:ss[.fffffffff]` ，其中 `[.fffffffff]` 是可选的。示例如下：

```
CREATE TABLE person_info ( name text PRIMARY KEY , birthtime time , address text );
INSERT INTO person_info (name,birthtime,address) VALUES ( 'lili', '08:12:54', 'hz');
```

date

支持使用int以及string表示date数据类型。如果使用string表示的date类型，格式为 `yyyy-mm-dd` 。示例如下：

```
CREATE TABLE person_info ( name text PRIMARY KEY , birthday date , address text );
INSERT INTO person_info (name,birthday,address) VALUES ( 'lucy', '2021-02-01', 'beijing');
```

nested

嵌套类型。在建表的时候可以指定对应的列为nested，该类型的数值支持按照String写入JSON数据，支持执行select搭配 `search_query` 语句进行嵌套类型的子查询，需要结合Lindorm的 `search index` 功能一起使用，需要注意的是 `describe` 出来该列暂时还是text类型。此外，nested类型在开源侧driver展示的还是text数据类型。示例如下：

```
CREATE TABLE person_info ( name text PRIMARY KEY , info nested , address text );
CREATE SEARCH INDEX  sidx ON ctable3 WITH COLUMNS (info ,address );
INSERT INTO ctable3 (name,info,address) VALUES ( 'lilei', '{"school":"alibaba"}', 'hz');
SELECT * FROM ctable3 WHERE search_query = '{+address:hz}';
```

暂时不支持的数据类型

- 集合数据类型：set，list，map
- 用户自定义类型：User-Defined Types
- 基础数据类型：durations

7.3.2.7.3. DESCRIBE

本节介绍DESCRIBE语法的说明和示例。

DESCRIBE KEYSPACES

输出所有Keyspace的名称。

语法

```
DESCRIBE KEYSPACES
```

参数

参数	类型	描述
keyspace_name	text	Keyspace名字。
name	text	Schema元素名字。

示例

```
DESCRIBE KEYSPACES;
```

DESCRIBE KEYSPACE

输出对应Keyspace组织结构以及该Keyspace下所有相关的表、UDT、UDF等Schema状态。

语法

```
DESCRIBE KEYSPACE [keyspace_name]
```

> **说明**
>
> - 如果省略 `keyspace_name` ，将输出当前Keyspace下的所有Schema状态。
> - 如果指定了 `ONLY` ，则仅创建用于重新创建键空间的DDL。所有键空间元素（如表）都将被省略。

参数

参数	类型	描述
keyspace_name	text	Keyspace名字。
name	text	Schema元素名字。
create_statement	text	用于重新创建Schema的Lindorm CQL语句。

示例

```
DESCRIBE KEYSPACE testks;
```

DESCRIBE TABLES

输出当前Keyspace中所有表的名称，如果Keyspace不存在，则输出所有Keyspace的名称。

语法

```
DESCRIBE TABLES
```

参数

参数	类型	描述
keyspace_name	text	Keyspace名字。
name	text	Schema元素名字。

示例

```
DESCRIBE TABLES
```

DESCRIBE TABLE

输出TABLE的Schema状态，该输出状态可用于重新创建指定表的Lindorm CQL命令。

语法

```
DESCRIBE TABLE [keyspace_name.] table_name
```

参数

参数	类型	描述
keyspace_name	text	Keyspace名字。
name	text	Schema元素名字。
create_statement	text	用于重新创建Schema的Lindorm CQL语句。

示例

```
DESCRIBE TABLE persioninfo;
```

DESCRIBE INDEX

输出TABLE的索引的Schema状态，该输出状态可用于重新创建指定索引的Lindorm CQL命令。

语法

```
DESCRIBE INDEX [keyspace_name.] index_name
```

参数

参数	类型	描述
keyspace_name	text	Keyspace名字。
name	text	Schema元素名字。
create_statement	text	用于重新创建Schema的Lindorm CQL语句。

示例

```
DESCRIBE INDEX testks.myindex;
```

7.3.2.7.4. DDL

本节介绍DDL语法的说明和示例。

概述

Lindorm CQL将数据存储在表中，其模式定义了表中所述数据的布局，并且这些表被分组在键空间中。Keyspace定义了许多选项，这些选项适用于它包含的所有Table，其中最主要的是Keyspace使用的复制策略，这里Lindorm CQL并不支持Keyspace的复制策略定义，以及副本数的定义，都是默认定义的值。Lindorm CQL支持使用CQL语言操作Lindorm高性能二级索引以及Lindorm全文索引Search Index，满足用户对于索引的使用需求。

Keyspace和Table的语法定义

Lindorm CQL Keyspace语法定义：

```
keyspace_name ::=  name
name          ::=  unquoted_name | quoted_name
unquoted_name ::=  re('[a-zA-Z_0-9]{1, 48}')
quoted_name   ::=  '"' unquoted_name '"'
```

Lindorm CQL Table语法定义：

```
table_name    ::=  [ keyspace_name '.' ] name
name          ::=  unquoted_name | quoted_name
unquoted_name ::=  re('[a-zA-Z_0-9]{1, 48}')
quoted_name   ::=  '"' unquoted_name '"'
```

> **说明**
>
> - Keyspace名和Table名都由字母、数字、字符组成，不能为空。
> - Keyspace名和Table名不超过48个字符。该限制主要是为了避免文件名（可能包括Keyspace名和Table名）超过某些文件的限制系统。
> - 默认情况下，Keyspace名和Table名不区分大小写，比如myTable和mytable是一样的。但使用双引号可强制进行区分，例如“myTable”和"mytable"是不一样的。
> - 表是Keyspace的一部分，并且表名可以由Keyspace名进行强制限定，比如：键空间KS和键空间GC都具有表Table，但他们的表名分别为ks.table和gc.table。

索引

Lindorm CQL同时支持二级索引和基于搜索引擎的Search Index。

- **二级索引**
  Lindorm CQL支持在表上创建二级索引，从而允许查询表时使用这些索引。二级索引由以下名称定义：

  ```
  index_name ::=  re('[a-zA-Z_0-9]+')
  ```

  如果要使用高性能二级索引，请参见高性能原生二级索引。
- **Search Index**
  Search Index支持的功能如下：
  - 多维查询。给定多个列的随机组合，快速返回查询结果。
  - 排序。提供Order By能力，返回结果按照任意指定列进行排序。
  - 模糊匹配。

  如果您想要通过Lindorm CQL使用Search Index，需要先在Lindorm上开通全文检索以及LTS服务，请参见特性介绍。

CREATE KEYSPACE

创建Keyspace。

语法

```
CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name WITH options
```

> **说明** Keyspace在Lindorm中类似于Namespace的概念，由于底层设计限制，所以Lindorm CQL暂时不支持设置副本复制策略以及副本数，都是使用默认参数。

参数

参数名称	示例值	描述
keyspace_name	testks	键值空间名称。
options	replication	目前取值固定为replication和durable_writes。 • replication：Map类型。表示副本的默认数据，底层默认2副本存储。 • durable_writes：Boolean类型。表示数据是否持久化写入。默认值为True。

示例

```
CREATE KEYSPACE testks WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
```

> **说明** Lindorm CQL Keyspace兼容层暂时不支持设置replication和durable_writes属性，您可以看到默认的2副本或者默认值为True的durable_writes属性，后续会开放对应功能设置。

ALTER KEYSPACE

修改Keyspace的选项。

语法

```
ALTER KEYSPACE keyspace_name WITH options
```

> **说明** ALTER KEYSPACE语句中的options和CREATE KEYSPACE语句中的options是一样的，但是由于Keyspace级别的options都是默认的，所以无需修改。

参数

ALTER KEYSPACE语句中的options和CREATE KEYSPACE语句中的options是一样的，请参见参数说明。

示例

```
ALTER KEYSPACE testks WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
```

USE

使用某个Keyspace。Lindorm CQL中的许多对象都绑定到一个键空间（表、用户定义的类型、函数等）。

语法

```
USE keyspace_name
```

参数

参数名称	示例值	描述
keyspace_name	testks	键值空间名称。

示例

```
USE testks;
```

DROP KEYSPACE

删除Keyspace。

语法

```
DROP KEYSPACE [ IF EXISTS ] keyspace_name
```

> 说明
> - 使用DROP KEYSPACE删除Keyspace会立即生效并且不可恢复，包括Keyspace中的所有表、UDT和函数以及这些表中包含的所有数据。
> - 如果Keyspace不存在，使用该语句系统会返回错误。除非您使用 `IF EXISTS` ，系统不返回错误，但该操作为无效操作。
> - 只有root用户具有DROP KEYSPACE的权限。

参数

参数名称	示例值	描述
keyspace_name	testks	键值空间名称。

示例

```
DROP KEYSPACE testks;
```

CREATE TABLE

创建TABLE。

语法

```
CREATE TABLE [ IF NOT EXISTS ] table_name
                  '('
                       column_definition
                       ( ',' column_definition )*
                       [ ',' PRIMARY KEY '(' primary_key ')' ]
                 ')' [ WITH table_options ]
column_definition      ::=  column_name cql_type [ STATIC ] [ PRIMARY KEY]
primary_key            ::=  partition_key [ ',' clustering_columns ]
partition_key          ::=  column_name
                            | '(' column_name ( ',' column_name )* ')'
clustering_columns     ::=  column_name ( ',' column_name )*
table_options          ::=  CLUSTERING ORDER BY '(' clustering_order ')' [ AND options ]
                            | options
clustering_order       ::=  column_name (ASC | DESC) ( ',' column_name (ASC | DESC) )*
```

说明

Lindorm CQL表具有名称，由一组行组成。创建TABLE表示将定义一行数据将由哪些列组成，哪些列将组成主键（Primary Key）以及该表的可选选项（Table Options）。除非使用 `IF NOT EXISTS` 语句，否则创建一个已经存在的表将返回错误。

Lindorm CQL表中的每一行都有一组预定义列，这些预定义列在创建表时定义（或后面使用ALTER语句添加）。

`column_definition` 主要由定义的列的名称和类型组成，它限制了该列接受哪些值。此外，列定义可以具有以下修饰符：

- `STATIC` ：表示将列声明为静态列（暂不支持）。
- `PRIMARY KEY` ：表示该列是表主键的唯一组成部分。

在一个表中，一行数据由其 `PRIMARY KEY` 唯一标识，因此所有表都必须定义一个 `PRIMARY KEY` （只有一个）。 `PRIMARY KEY` 定义由表中定义的一列或多列组成。在语法上，主关键字定义为关键字 `PRIMARY KEY` ，后跟用括号括起来的列名称的列表，该列名称用括号括起来，但是如果主键只有一列，则可以用 `PRIMARY KEY` 关键字替换该列定义。主键中定义的列的顺序很重要，将会影响数据的分布和存储顺序。

Lindorm CQL主键由两部分组成：

- `partition key` ：它是主键（Primary Key）的第一部分。它可以是单个列，或者使用附加的括号可以是多个列。一个表总是至少有一个分区键。
- `clustering columns` ：这些是主键定义的第一部分之后的列，这些列的顺序定义了聚类顺序，Lindorm CQL中允许不定义 `clustering columns` 。

`PRIMARY KEY` 定义的一些示例：

- PRIMARY KEY（a）：a是分区键，没有聚类列。
- PRIMARY KEY（a，b，c）：a是分区键，b和c是聚簇列。
- PRIMARY KEY（（a，b），c）：a和b组成分区键（通常称为复合分区键），c是聚簇列。

在Lindorm CQL中，`PRIMARY KEY`具有较强的意义，`partition key`和`clustering columns`组成`PRIMARY KEY`，单独的`partition key`或`clustering columns`不具有独立使用的意义。

在表中，Lindorm CQL定义分区的概念。分区就是一组共享相同分区键值的行。请注意，如果分区键由多个列组成，则行仅属于同一分区，并且所有分区键列的值都相同。给定以下表定义和内容：

```
CREATE TABLE persioninfo (
    a int,
    b int,
    c int,
    d int,
    PRIMARY KEY ((a, b), c, d)
);
SELECT * FROM persioninfo;
   a | b | c | d
  ---+---+---+---
   0 | 0 | 0 | 0    // row 1
   0 | 0 | 1 | 1    // row 2
   0 | 1 | 2 | 2    // row 3
   0 | 1 | 3 | 3    // row 4
   1 | 1 | 4 | 4    // row 5
```

Lindorm CQL中，`partition key`以及`clustering columns`共同组成`PRIMARY KEY`，他们的地位是等价的，Lindorm CQL中`partition key`和`cluster column`组合在一起共同决定一条数据属于哪个节点，而不是CQL中由`partition key`单独决定一条数据属于哪个节点，例如上述例子中row1和row2在Lindorm CQL中可以确定是在一个节点中，但是在传统CQL中并不能保证，这是Lindorm CQL与传统CQL的较大区别。

示例

```
CREATE TABLE tb (name text PRIMARY KEY , age int);   //默认创建以name为主键的表。
CREATE TABLE ttltb (name text PRIMARY KEY, age int) WITH default_time_to_live = 1000;  //设置表的ttl时间是1000s。
CREATE TABLE cptb (name text PRIMARY KEY, age int) WITH compression = {'sstable_compression': 'LZ4Compressor'};   //设置压缩算法为LZ4压缩；默认是SNAPPY。
CREATE TABLE hct ( name text PRIMARY KEY , age int  ) WITH extensions = {'COLD_BOUNDARY':'10'};  //设置lindorm的冷热分离的时间线是10s。
```

Lindorm CQL Table暂不支持的定义功能

partition key：确定数据分布式从属的对应节点，Lindorm CQL不能由单一的`partition key`确定该行数据从属的物理节点。

支持的table_Option列表

Lindorm CQL的`table_options`和Cassandra的`table_options`有一定差异，Lindorm CQL目前支持如下的选项设置。

选项	类型	描述
default_time_to_live	int	表的默认到期时间（TTL），以秒为单位。默认值为0。
compression	map	可以设置对应文件（sstable）的compression算法，支持LZ4、ZSTD、SNAPPY算法。具体请参考下文Compression介绍。
extensions	map	支持扩展设置，包括设置是否使用冷存储、使用冷热分离、表的一致性级别等属性。具体请参考下文Extensions介绍。

Compression

Lindorm CQL支持如下几种可配置的压缩算法，各个压缩算法的系数暂时使用默认参数进行设置。

- LZ4（LZ4Compressor）。创建LZ4压缩的表如下：

```
CREATE TABLE persioninfo (
   id int,
   name text,
   address text,
   PRIMARY KEY (id, name)
) with compression = {'class': 'LZ4Compressor'};
```

- ZSTD（ZstdCompressor）。创建ZSTD压缩的表如下：

```
CREATE TABLE persioninfo (
   id int,
   name text,
   address text,
   PRIMARY KEY (id, name)
) with compression = {'class': 'ZstdCompressor'};
```

- SNAPPY（SnappyCompressor）。创建SNAPPY压缩的表如下：

```
CREATE TABLE persioninfo (
   id int,
   name text,
   address text,
   PRIMARY KEY (id, name)
) with compression = {'class': 'SnappyCompressor'};
```

> **说明** compression属性也可以通过 `ALTER TABLE` 语法进行修改设置。

Extensions

对于Lindorm的特有属性我们通过extensions扩展属性进行设置，扩展属性如下：

- 冷存储。关键字为 `STORAGE_POLICY` ，配置值为 `COLD` 表示冷存储， `DEFAULT` 表示使用默认热存储。

```
CREATE TABLE persioninfo (name text PRIMARY KEY, age int) WITH extensions = {'STORAGE_POLICY' : 'COLD'};
//建表指定表为冷存储。
ALTER TABLE persioninfo WITH extensions = {'STORAGE_POLICY' : 'DEFAULT'};  //修改表属性为热存储。
```

- 冷热分离。关键字为 `COLD_BOUNDARY` ，在冷热分离使用过程中，无需把表或列簇的属性设置为COLD，如果已经把列簇的属性设置为了COLD，请您将冷存储的属性去除，具体请参考冷存储介绍。

```
CREATE TABLE persioninfo (name text PRIMARY KEY, age int) with extensions = {'COLD_BOUNDARY':'86400'};  //
建表指定冷热分离时间线为1天（86400秒），超过时间线数据被写到冷介质。
ALTER TABLE persioninfo with extensions = {'COLD_BOUNDARY':''};    //取消冷热分离。
ALTER TABLE persioninfo with extensions = {'COLD_BOUNDARY':'1000'};   //修改时间线为1000s。
```

- CONSISTENCY_TYPE：对于lindorm多zone场景下，设置表的一致性级别，有eventual、timestamp、basic、strong四种级别。

```
CREATE TABLE persioninfo (name text PRIMARY KEY, age int) with extensions = {'CONSISTENCY_TYPE':'strong'};
//建表一致性级别为strong。
ALTER TABLE persioninfo with extensions = {'CONSISTENCY_TYPE':'eventual'};  //修改一致性级别为eventual。
```

- Mutability：用户在使用二级索引、search index之前需要设置Mutability，取值：IMMUTABLE、IMMUTABLE_ROWS（CONSISTENCY_TYPE必须为strong），MUTABLE_LATEST（CONSISTENCY_TYPE必须为strong），MUTABLE_ALL（建表默认的级别，CONSISTENCY_TYPE必须为strong）。

```
CREATE TABLE persioninfo (name text PRIMARY KEY, age int) with extensions = {'MUTABILITY':'IMMUTABLE'};  /
/建表指定mutability为IMMUTABLE。
ALTER TABLE persioninfo with extensions = {'MUTABILITY':'MUTABLE_LATEST'};   //修改表mutability为MUTABLE_LA
TEST。
```

暂不支持的table_Option列表

选项	类型	描述
comment	string	表示对表的描述。默认值为f。
speculative_retry	simple simple	默认值为99PERCENTILE。
cdc	boolean	在表上创建一个更改数据捕获（CDC）日志。默认值为false。
gc_grace_seconds	int	等待垃圾收集墓碑（删除标记）之前的时间。默认值为86400。
bloom_filter_fp_chance	float	稳定布隆过滤器误报的目标概率。所述布隆过滤器的大小将确定所提供的概率，因此降低此值会影响内存中和磁盘上布隆过滤器的大小。默认值为0.00075。
compaction	map	默认值为STCS策略。
caching	map	无
memtable_flush_period_in_ms	map	默认值为0。
read_repair	int	默认值为BLOCKING。

ALTER TABLE

修改TABLE。

语法

```
ALTER TABLE table_name alter_table_instruction
alter_table_instruction ::=  ADD column_name cql_type ( ',' column_name cql_type )*
                           | DROP column_name ( column_name )*
                           | WITH options
```

说明

`ALTER TABLE` 语句可以进行以下操作：

- 可以通过ADD指令，将新列添加到表中。由于表的主键无法更改，因此新添加的列将永远不会成为主键的一部分。需要注意，紧凑表对列添加有一定的限制。
- Lindorm CQL暂时不支持DROP column操作。
- 可以通过WITH指令，更改表的选项。支持修改的表的选项与创建表时的选项相同，但是无法更改 `CLUSTERING ORDER` 。Lindorm CQL支持修改 `default_time_to_live` 、 `compression` 、 `extensions` 选项的设置。

示例

```
CREATE TABLE persioninfo (name text PRIMARY KEY, age int);
ALTER TABLE persioninfo ADD address text;
```

DROP TABLE

删除TABLE。

语法

```
DROP TABLE [ IF EXISTS ] table_name
```

> 说明
> - 使用DROP TABLE删除TABLE会立即生效并且不可恢复，包括Table中的所有数据。
> - 如果TABLE不存在，使用该语句系统会返回错误。除非您使用 `IF EXISTS` ，系统不返回错误，但该操作为无效操作。
> - 只有root用户具有DROP TABLE的权限。

参数

参数名称	示例值	描述
table_name	persioninfo	表名称。

示例

```
DROP TABLE persioninfo;
```

TRUNCATE

清空表数据。

语法

```
TRUNCATE [ TABLE ] table_name
```

> 说明
> - 使用 `TRUNCATE` 会永久删除表中所有的数据，但不会删除表本身的表结构状态。
> - 只有root用户具有TRUNCATE TABLE的权限。

参数

参数名称	示例值	描述
table_name	persioninfo	表名称。

示例

```
TRUNCATE TABLE persioninfo;
```

CREATE INDEX

创建二级索引。

语法

```
CREATE [ CUSTOM ] INDEX [ IF NOT EXISTS ] [ index_name ]
                                ON table_name '(' index_identifier ')'
                                [ USING string [ WITH OPTIONS = map_literal ] ]
index_identifier       ::=  column_name
                          | '(' column_name ')'
```

说明

`CREATE INDEX` 语句用于为指定的表中的列自动创建二级索引。您可以在 `ON` 关键字之前指定索引名。如果该列已经存在数据，则将对其进行异步索引。创建索引后，您在插入数据时，系统将自动为该列的新数据建立索引。

如果创建一个已经存在的索引系统将返回错误。如果使用 `IF NOT EXISTS` 选项创建一个已经存在的索引，则操作属于无效操作。

`CREATE INDEX` 语句只支持对单个列进行索引构建。如果想要对表下面的多个列构建索引的话，您可以使用 `CREATE CUSTOM INDEX` 语句中的 `USING 'com.alibaba.lindorm.cserver.schema.index.LindormSecondaryIndex'` 进行构建。

示例

```
CREATE INDEX myindex ON persioninfo (c2);
CREATE INDEX ON persioninfo (c2);
CREATE CUSTOM INDEX myindex ON persioninfo (c1,c2) USING 'com.alibaba.lindorm.cserver.schema.index.LindormSe
condaryIndex';
```

DROP INDEX

删除二级索引。

语法

```
DROP INDEX [ IF EXISTS ] index_name
```

> **说明**
>
> - 使用 `DROP INDEX` 语句用于删除已存在的二级索引。语句的变量是索引名称 `index_name` ，可以选择指定索引的键空间。
> - 如果索引不存在，使用该语句系统会返回错误。除非您使用 `IF EXISTS` ，系统不返回错误，但该操作为无效操作。

示例

```
DROP INDEX myindex;
```

CREAT SEARCH INDEX

创建全文索引。

语法

```
CREATE SEARCH INDEX [ IF NOT EXISTS ] index_name ON [keyspace_name.]table_name
| [ WITH [ COLUMNS  (column1,...,columnn)  ]
| [ WITH [ COLUMNS  (*)  ]
```

> **说明**
>
> - `CREATE SEARCH INDEX` 语句支持在源数据表的某些列上构建全文索引。
> - `WITH COLUMNS (column)` ：指定某个或某几个列构建SEARCH INDEX，各column间以英文逗号（,）分隔。
> - `WITH COLUMNS (*)` ：使用（*）符号表示对所有列构建全文索引。
> - 创建SEARCH INDEX的表需要对源数据表的属性通过Extension进行扩展。如果是Lindorm多Zone场景下，默认需要指定一致性以及MUTABLE属性，非多Zone场景下不需要进行设置。

示例

```
CREATE SEARCH INDEX schidx ON persioninfo WITH COLUMNS (c2, c3);
```

DROP SEARCH INDEX

删除全文索引。

语法

```
DROP SEARCH INDEX [IF EXISTS] ON [keyspace_name.]table_name;
```

示例

```
DROP SEARCH INDEX ON testks.persioninfo;
```

REBUILD SEARCH INDEX

将索引状态设置为有效。

语法

```
REBUILD SEARCH INDEX [ASYNC] [IF EXISTS] ON [keyspace_name.]table_name;
```

> **说明** 全文索引构建完成以后，索引的状态是 `INACTIVE` 的，需要手动执行 `REBUILD` 操作，才能将索引状态置为有效。执行 `REBUILD` 操作还会为存量数据构建SEARCH INDEX，整个过程会比较漫长，可以通过 `ASYNC` 参数指定此次构建是异步操作。

示例

```
REBUILD SEARCH INDEX ON persioninfo;
REBUILD SEARCH INDEX ASYNC ON persioninfo;
```

ALTER SEARCH INDEX

修改全文索引的状态、手动添加索引列或者删除索引列。

语法

```
ALTER SEARCH INDEX SCHEMA [IF EXISTS] ON [keyspace_name.]table_name
  ( ADD FIELD column_name
  | DROP FIELD column_name) ;
```

> **说明**
> - 使用 `ALTER SEARCH INDEX` 修改全文索引的状态后，需要使用 `REBUILD` 重建索引的状态。
> - 添加 `ADD SEARCH INDEX` 或删除 `DROP SEARCH INDEX` 的索引列需要在源数据表中存在。

示例

```
ALTER SEARCH INDEX SCHEMA ON persioninfo ADD  (c3);
ALTER SEARCH INDEX SCHEMA ON persioninfo DROP  (c2);
```

7.3.2.7.5. DML

本节介绍DML语法的说明和示例。

SELECT

查询数据。

语法

```
select_statement ::=  SELECT [ JSON | HOTDATA ] ( select_clause | '*' )
                      FROM table_name
                      [ WHERE where_clause ]
                      [ GROUP BY group_by_clause ]
                      [ ORDER BY ordering_clause ]
                      [ PER PARTITION LIMIT (integer | bind_marker) ]
                      [ LIMIT (integer | bind_marker) ]
                      [ ALLOW FILTERING ]
select_clause    ::=  selector [ AS identifier ] ( ',' selector [ AS identifier ] )
selector         ::=  column_name
                      | term
                      | CAST '(' selector AS cql_type ')'
                      | function_name '(' [ selector ( ',' selector )* ] ')'
                      | COUNT '(' '*' ')'
where_clause     ::=  relation ( AND relation )*
relation         ::=  column_name operator term
                      '(' column_name ( ',' column_name )* ')' operator tuple_literal
                      TOKEN '(' column_name ( ',' column_name )* ')' operator term
operator         ::=  '=' | '<' | '>' | '<=' | '>=' | '!=' | IN | CONTAINS | CONTAINS KEY
group_by_clause  ::=  column_name ( ',' column_name )*
ordering_clause  ::=  column_name [ ASC | DESC ] ( ',' column_name [ ASC | DESC ] )*
```

> 说明
> - SELECT语句读取表中一个或多个行的一个或多个列。它返回与请求匹配的行的结果集，其中每一行都包含与查询相对应的选择值。另外，可以将包括聚合函数在内的函数应用于结果。
> - SELECT语句至少包含一个选择子句和该选择所在的表的名称。请注意，Lindorm CQL暂不支持Join或子查询，因此SELECT语句仅适用于单个表。在大多数情况下，SELECT还有WHERE子句，并且可以选择其他子句来对结果进行排序或限制。如果提供了 `ALLOW FILTERING` 语句，则可以允许需要过滤的查询。

示例

```
CREATE TABLE persioninfo  (c1 int primary key, c2 text, c3 text);
SELECT c1, c2 FROM persioninfo WHERE c1 IN (199, 200, 207);
SELECT JSON c1, c2 FROM persioninfo WHERE c1 = 199;
SELECT c1 AS name, c2 AS address FROM persioninfo;
CREATE TABLE persioninfo  (c1 int , c2 text, c3 text, PRIMARY KEY(c1,c2));
SELECT *
FROM persioninfo
WHERE c1 = 12
  AND c2 > '2011-02-03'
  AND c2 <= '2012-01-01'
SELECT COUNT (*) AS user_count FROM persioninfo;
```

SELECT的选择子句

选择子句确定需要查询哪些列并在结果集中返回，以及在返回之前对该结果进行对应的转换。它由逗号分隔的选择器（Selector）列表组成，或者由通配符（*）组成，以选择表中定义的所有列。包括以下内容：

- **Selectors**
  包括以下类型：
  - `column_name`：查询的表的列名，以检索该列的值。
  - `term`：一个term，通常嵌套在其他选择器（如函数）中，如果直接选择一个术语，则结果集的对应列将仅具有返回的每一行的该术语的值。
  - `CAST '(' selector AS cql_type ')'`：强制转换，允许将嵌套选择器转换为（兼容）类型。
  - `function_name '(' [ selector ( ',' selector )* ] ')'`：函数调用，其中参数是选择器本身。
  - `COUNT '(' '*' ')'`：对COUNT函数的特殊调用COUNT（*），对所有非空结果进行计数。
- **WRITETIME function**
  Selection仅支持特殊函数（其他任何地方都不允许）`WRITETIME`，这个函数仅接受一个参数，并且该参数必须为列名。这个函数允许检索存储在每个列内部的元信息，即WRITETIME列的值的时间戳。
- **where clause**
  `where` 子句指定必须查询的行。它由作为主键的一部分（partition key 或者clustering columns），或在其上定义了二级索引的列的关系组成。并非所有关系查询都是允许的，例如，不支持 `partition key` 的非相等关系（其中IN被视为相等关系）。对于给定的 `partition key`，`clustering column` 引起行的排序，并且它们的关系被限制为允许选择连续的（对于排序）行集的关系。例如，创建如下表 `posts`：

```
CREATE TABLE posts (
    userid text,
    blog_title text,
    posted_at timestamp,
    entry_title text,
    content text,
    category int,
    PRIMARY KEY (userid, blog_title, posted_at)
)
```

可以进行如下的查询：

```
 SELECT entry_title, content FROM posts
 WHERE userid = 'john doe'
   AND blog_title='John''s Blog'
   AND posted_at >= '2012-01-01' AND posted_at < '2012-01-31'
```

但是不能进行如下的查询，因为它不会选择一组连续的行（并且我们假设未设置二级索引）。

```
SELECT entry_title, content FROM posts
 WHERE userid = 'john doe'
   AND posted_at >= '2012-01-01' AND posted_at < '2012-01-31'
```

此外，仅在 `partition key` 的最后一列和 `primary key` 的最后一列上允许IN关系，也可以使用元组符号将 `Clustering column` 组合在一起。如下：

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND (blog_title, posted_at) > ('John''s Blog', '2012-01-01')
```

将按照聚类顺序要求所有排在“John''s Blog”为blog_tile和“ 2012-01-01”为post_at之后的行。特别是，只要post_at <='2012-01-01'的行的blog_title>'John's Blog'，则将返回该行，但以下情况除外：

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND blog_title > 'John''s Blog'
   AND posted_at > '2012-01-01'
```

元组符号也可用于 `clustering column` 的IN子句：

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND (blog_title, posted_at) IN (('John''s Blog', '2012-01-01'), ('Extreme Chess', '2014-06-01'))
```

- **Grouping results**
  GROUP BY选项允许将所有共享一组列的相同值的选定行浓缩为单个行。使用GROUP BY选项，只能在 `partition key` 级别或 `clustering column` 级别对行进行分组。因此，GROUP BY选项仅按主键顺序接受主键列名称作为参数。如果主键列受到相等性限制的约束，则它不需要出现在GROUP BY子句中。

  > **说明** 因为cql的group by是把数据拉到一个server节点进行处理，数据量大的情况下，整个事件比较长。

- **Ordering results**
  ORDER BY子句允许选择返回结果的顺序。它以列名列表和列顺序作为参数（升序为ASC，降序为DESC，没有指定顺序与ASC等效）。当前，可能的顺序受表上定义的 `clustering order` 限制：
  - 如果表是在没有任何特定 `CLUSTERING ORDER` 的情况下定义的，则允许的排序是由 `clustering columns` 和与之相反的顺序引起的。
  - 否则，允许的顺序是 `CLUSTERING ORDER` 选项的顺序和相反的顺序。

- **Limiting results**
  SELECT语句的LIMIT选项限制查询返回的行数。

- **Allowing filtering**
  默认情况下，Lindorm CQL仅允许不涉及“filtering”服务器端的数据的选择查询，即我们知道所有（实时的）记录读取将在结果集中返回（或部分返回）。原因是这些“non filtering”查询在某种意义上可以具有可预测的性能，因为它们可以在与查询返回的数据量成一定比例的时间内执行完成（可以通过LIMIT进行控制）。
  ALLOW FILTERING选项允许显式允许（某些）需要过滤的查询。请注意，使用ALLOW FILTERING的查询可能造成服务端出现不可预测的性能问题，即使选择少数记录的查询也可能表现，这取决于集群中存储的数据总量。
  例如，创建的如下表格，其中包含用户个人资料及其出生年份（带有次级索引）和居住国家/地区：

```
CREATE TABLE users (
    username text PRIMARY KEY,
    firstname text,
    lastname text,
    birth_year int,
    country text
)
CREATE INDEX ON users(birth_year);
```

可以进行以下有效查询：

```
SELECT * FROM users;
SELECT * FROM users WHERE birth_year = 1981;
```

因为在这两种情况下，Lindorm CQL都会保证这些查询的性能与返回的数据量成正比。特别是，如果没有用户出生于1981年，那么第二次查询的性能将不取决于存储在数据库中的用户配置文件的数量（至少不是直接地：由于二级索引实现的考虑，该查询可能仍取决于集群中节点的数量，以及间接取决于集群存储的数据量。当然，在实践中，两个查询都可能返回非常大的结果集，但是始终可以通过添加LIMIT来控制返回的数据量。以下的查询将被拒绝：

```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR';
```

因为Lindorm CQL无法保证即使在查询的结果很小的情况下，也不去进行大量数据的扫描。通常，它将扫描所有1981年出生的用户的索引条目，即使实际上只有少数来自法国。但是，如果您“知道自己在做什么”，则可以使用ALLOW FILTERING强制执行此查询，因此以下查询有效：

```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR' ALLOW FILTERING;
```

- **HOTDATA**
  在用户建表指定冷热分离的场景下，如果用户想要查询热数据，可以在SELECT语句关键字下指定HOTDATA：

  ```
  SELECT HOTDATA * FROM persioninfo;
  ```

  默认不指定的话，请求会在冷热数据都进行查询，然后合并出所需的数据。
- **SEARCH INDEX SELECT操作**
  对构建SEARCH INDEX的列支持模糊匹配查询，以及多维组合查询以及排序功能。
  - **常见模糊匹配**
    构建SEARCH INDEX的列，在进行select操作的时候，支持对构建索引的列进行模糊匹配查询，模糊匹配使用%表示0到任意个字符匹配，常见的模糊匹配支持前缀匹配、后缀匹配、前后缀匹配、完全匹配。示例如下：

    ```
    SELECT * FROM persioninfo WHERE c2 LIKE 'v2';//完全匹配，c2列数字为v2。
    SELECT * FROM kss.tb WHERE c2 LIKE '%v2' //前缀匹配，表示v2关键字符前支持0到任意个字符匹配。
    SELECT * FROM kss.tb WHERE c2 LIKE 'v2%' //后缀匹配，表示v2关键字符后支持0到任意个字符匹配。
    SELECT * FROM kss.tb WHERE c2 LIKE '%v2%'//contains匹配，表示v2关键字符前后支持任意个字符匹配。
    ```

- **指定search_query查询**
  使用search_query关键字查询，支持在select 语句中支持任意类solr原生query的语句，满足更丰富的查询检索需求。示例如下：

```
SELECT selectors
  FROM table
  WHERE (indexed_column_expression | search_query = 'search_expression')
  [ LIMIT n ]
  [ ORDER BY column_name ] ;
```

Select语句中search_query关键字后面会跟随search_expression，具体的search_query的跟随字符限制请参考跟随符限制。常见的search_query跟随的选项支持情况如下：

```
{
  "q": query_expression (string),
  "fq": filter_query_expression(s) (string_or_array_of_strings, ...),
  "facet": facet_query_expression (object)
  "sort": sort_expression (string),
  "start": start_index(number),
  timeAllowed: search_time_limit_ms,
  "TZ": zoneID),   //  Any valid zone ID in java TimeZone class
  "paging": "driver" (string),
  "distrib.singlePass": true|false (boolean),
  "shards.failover": true|false (boolean),    // Default: true
  "shards.tolerant": true|false (boolean),    // Default: false
  "commit": true|false (boolean),
  "route.partition": partition_routing_expression (array_of_strings),
  "route.range": range_routing_expression (array_of_strings),
  "query.name": query_name (string),
}
```

示例如下：

```
CREATE TABLE persioninfo (name text PRIMARY KEY, city text,id int, friend nested) WITH extensions = {'CO
NSISTENCY_TYPE':'strong', 'MUTABILITY':'IMMUTABLE'};
CREATE SEARCH INDEX sidx on persioninf WITH COLUMNS (id,city);
INSERT INTO persioninfo (name,city, id, friend) VALUES ('MY','hangzhou', 18, '[{"lili":18}, {"gg":20}]')
;
SELECT name,id FROM persioninfo  WHERE search_query = '+city:hangzhou +age:[1 TO 18]';
```

> **说明** 因为 `search_query` 支持的语义比较丰富，所以暂时不支持 `searc_query` 语句与其他condition的组合使用，比如 `select * from table where search_query = 'c1:name' And column = 'a'` ，这种用法是不支持的。

UPDATE

更新数据。

语法

```
update_statement ::=  UPDATE table_name
                      [ USING update_parameter ( AND update_parameter )* ]
                      SET assignment ( ',' assignment )*
                      WHERE where_clause
update_parameter ::=  ( integer | bind_marker )
assignment       ::=  simple_selection '=' term
                     | column_name '=' column_name ( '+' | '-' ) term
                     | column_name '=' list_literal '+' column_name
simple_selection ::=  column_name
                     | column_name '[' term ']'
                     | column_name '.' `field_name
condition        ::=  simple_selection operator term
```

> 说明
- UPDATE语句为表指定的行更新一个或多个列。where_clause用于选择要更新的行，并且必须包括组成 `PRIMARY KEY` 的所有列。然后使用SET关键字设置非主键列。
- 与SQL不同，UPDATE默认情况下不检查该行之前是否存在：如果该行之前不存在，则创建该行，否则进行更新。所以，没有办法知道是否发生了创建或更新。Lindorm CQL中的counter列允许与非counter列共存，这点是lindorm CQL与Apache Cassandra CQL的一个区别。
- 在UPDATE语句中，同一分区键（partition key）中的所有更新都是 `atomically` 以及 `isolation` 。
- 关于 `assignment` ，c=c + 3用于递增/递减计数器。 “=”符号后的列名称必须与“=”符号前的列名称相同。请注意，递增/递减仅允许在计数器（counter）上进行，并且是计数器唯一允许的更新操作。

示例

```
CREATE TABLE product (productname text PRIMARY KEY, company text, subdepartment text)
UPDATE product
   SET company    = 'aliyun cloud',
       subdepartment = 'database nosql'
 WHERE productname = 'lindorm';
```

DELETE

删除行或行的一部分。

语法

```
delete_statement ::=  DELETE [ simple_selection ( ',' simple_selection ) ]
                      FROM table_name
                      [ USING update_parameter ( AND update_parameter )* ]
                      WHERE where_clause
```

> 说明
- `DELETE` 语句删除列或者行。如果在 `DELETE` 关键字之后直接提供了列名，则仅从WHERE子句指示的行中删除那些列。否则将删除整个行。
- WHERE子句指定要删除的行。使用IN运算符可以用一条语句删除多行。暂不支持删除一定范围的行。
- 在 `DELETE` 语句中，同一分区键（Partition Key）中的所有删除操作都是以原子方式独立应用的。

示例

```
 INSERT INTO t1 (c1,c2,c3) VALUES ( '11', '22', '33');
 DELETE c2 FROM t1 WHERE c1 IN ( '11');
```

7.3.2.7.6. DCL

DCL（Data Control Language）主要包含用户角色管理（Database Role）语法和权限管理（Permission）语法。本节介绍DCL语法的说明和示例。

用户角色管理

Lindorm CQL语法使用Database Roles来表示用户角色。Role的定义如下：

```
role_name ::=  identifier | string
```

Lindorm CQL与Apache Cassandra CQL对Database Role的操作相似。默认情况下，用户不具有登录权限或超级用户权限。

权限管理

Lindorm CQL语法支持对语言结构中的各层级授予权限，权限支持向下兼容，所以对上层结构中的资源授予权限会自动赋予该结构之下的所有资源相同的权限。例如：当对1个Keyspace授予SELECT权限后，该权限将自动授予Keyspace中的所有TABLE。

> **说明** 权限进行更改后，客户端不需要重新建立连接。

Lindorm CQL支持的权限有：`CREATE`、`ALTER`、`DROP`、`SELECT`、`MODIFY`。

各权限对资源类型的支持情况如下表：

> **说明** 各权限对支持的资源类型有区分。若对无法应用权限的资源授予权限将导致错误。

权限类型	资源支持情况	可操作的语句
`CREATE`	ALL KEYSPACES	在任意keyspace执行`CREATE KEYSPACES`和`CREATE TABLE`。
`CREATE`	KEYSPACE	在某个特定keyspace执行`CREATE TABLE`。
`ALTER`	ALL KEYSPACES	在任意keyspace执行`ALTER KEYSPACES`和`ALTER TABLE`。
`ALTER`	KEYSPACE	在某个特定keyspace执行`ALTER TABLE`。
`DROP`	ALL KEYSPACES	在任意keyspace执行`DROP KEYSPACES`和`DROP TABLE`。
`DROP`	KEYSPACE	在某个特定keyspace执行`DROP TABLE`。
`SELECT`	ALL KEYSPACES	在任意keyspace执行`SELECT KEYSPACES`和`SELECT TABLE`。
`SELECT`	KEYSPACE	在某个特定keyspace执行`SELECT TABLE`。
`SELECT`	TABLE	在特定表下执行`SELECT`操作。
`MODIFY`	ALL KEYSPACES	在任意keyspace执行`MODIFY KEYSPACES`和`MODIFY TABLE`。
`MODIFY`	KEYSPACE	在某个特定keyspace执行`MODIFY TABLE`。
`MODIFY`	TABLE	在特定表下执行`MODIFY`操作。

CREATE ROLE

创建用户角色。

语法

```
create_role_statement ::=  CREATE ROLE [ IF NOT EXISTS ] role_name
                               [ WITH role_options ]
role_options          ::=  role_option ( AND role_option )*
role_option           ::=  PASSWORD '=' string
                         | LOGIN '=' boolean
                         | SUPERUSER '=' boolean
                         | OPTIONS '=' map_literal
```

> **说明**
> - 通过`CREATE ROLE`语句，可创建用户角色，必须指定登录密码。通过该语句，您也可指定登录权限或超级用户权限。
> - 默认情况下，Lindorm CQL的Database Role操作中，角色不具有超级用户权限。

参数

参数	说明
role_name	指定用户角色名称。

参数	说明
role_option	• `PASSWORD` 标示用户登录的密码信息。 **说明** `PASSWORD` 参数必须设置。 • `LOGIN` 标示用户是否具有登录权限。默认为False。 **说明** 若用户具备登录权限，则客户端在连接时，会被标识为该用户的角色，即：在连接期间，客户端可获得授予该用户的权限。 • `SUPERUSER` 标示用户是否为超级用户。 **说明** 超级用户具备以下权限： ○ 支持创建超级用户。 ○ 具备Drop Keyspace、Drop Table、Truncate权限。 ○ 授权数据库资源角色权限。资源类型包括：Keyspace、Table。 ○ 支持创建层次权限结构。

示例

```
CREATE ROLE role1 WITH PASSWORD = 'password_a' AND LOGIN = true;
CREATE ROLE role2 WITH PASSWORD = 'password_b' AND LOGIN = true AND SUPERUSER = true;
```

ALTER ROLE

更改用户角色选项。

语法

```
alter_role_statement ::=  ALTER ROLE role_name WITH role_options
```

说明 通过 `ALTER ROLE` 语句，可修改用户的登录密码，登录权限和超级用户身份。

参数

`ALTER ROLE` 语句和 `CREATE ROLE` 语句的参数说明相同，请参见参数说明。

示例

```
ALTER ROLE bob WITH PASSWORD = 'PASSWORD_B' AND SUPERUSER = false;
```

DROP ROLE

删除用户。

语法

```
drop_role_statement ::=  DROP ROLE [ IF EXISTS ] role_name
```

说明

- 通过 `DROP ROLE` 语句，可删除用户。
  - 客户端对待删除的用户需具有删除权限。
  - 客户端不能删除状态为正在登录的用户。
  - 只有超级用户才可删除超级用户。
- `[ IF EXISTS ]` 语句说明：
  - 若未使用该选项，尝试删除不存在的用户时会出现无效的查询条件。
  - 若使用了该选项，尝试删除不存在的用户时，则该语句为 `no-op` 。

参数

参数	说明
role_name	指定用户角色名称。

示例

```
DROP ROLE bob;
DROP ROLE IF EXISTS bob;
```

LIST ROLES

显示已有的用户角色。

语法

```
list_roles_statement ::=  LIST ROLES [ OF role_name ]
```

参数

参数	说明
role_name	指定用户角色名称。

示例

```
LIST ROLES;//超级用户才可执行
LIST ROLES  OF role1;//展示某个用户的信息
```

GRANT PERMISSION

授予用户的权限。

语法

```
grant_permission_statement ::=  GRANT permissions ON resource TO role_name
permissions                ::=  ALL [ PERMISSIONS ] | permission [ PERMISSION ]
permission                 ::=  CREATE | ALTER | DROP | SELECT | MODIFY
resource                   ::=  ALL KEYSPACES
                                | KEYSPACE keyspace_name
                                | [ TABLE ] table_name
```

参数

参数	说明
role_name	指定用户角色名称。

参数	说明
permission	• `CREATE` 指定具有“创建“的权限。 • `ALTER` 指定具有更换的权限。 • `DROP` 指定具有删除的权限。 • `SELECT` 指定具有选择的权限。 • `MODIFY` 指定具有修改的权限。
resource	指定被授予权限的资源，如 `ALL KEYSPACES` 、 `KEYSPACE` 、 `TABLE` 。
keyspace_name	指定Keyspace名称。
table_name	指定表名称。

示例

```
GRANT SELECT ON ALL KEYSPACES TO role;
GRANT SELECT ON KEYSPACE  ks TO role;
```

REVOKE PERMISSION

撤销用户的权限。

语法

```
revoke_permission_statement ::=  REVOKE permissions ON resource FROM role_name
```

参数

参数	说明
role_name	指定用户角色名称。
permission	• `CREATE` 指定具有创建的权限。 • `ALTER` 指定具有更换的权限。 • `DROP` 指定具有删除的权限。 • `SELECT` 指定具有选择的权限。 • `MODIFY` 指定具有修改的权限。
resource	指定被授予权限的资源，如 `ALL KEYSPACES` 、 `KEYSPACE` 、 `TABLE` 。

示例

```
REVOKE SELECT ON ALL KEYSPACES FROM role;
```

LIST PERMISSIONS

显示用户的各资源下所授权的权限。

语法

```
list_permissions_statement ::=  LIST permissions [ ON resource ] [ OF role_name  ]
```

> **说明**
> - `ON resource`：限定资源。若未设置，则会显示用户下所有资源的权限信息。
> - `OF role_name`：限定用户。默认必须指定用户。仅超级用户可在不指定用户的情况下列举权限。

参数

参数	说明
role_name	指定用户角色名称。
permissions	• `CREATE` 指定具有创建的权限。 • `ALTER` 指定具有更换的权限。 • `DROP` 指定具有删除的权限。 • `SELECT` 指定具有选择的权限。 • `MODIFY` 指定具有修改的权限。
resource	指定被授予权限的资源，如 `ALL KEYSPACES` 、 `KEYSPACE` 、 `TABLE` 。

示例

```
LIST ALL PERMISSIONS OF role;
LIST ALL PERMISSIONS ON KEYSPACE kss OF role;
```

7.3.2.7.7. 函数说明

本节介绍Lindorm CQL语法支持的函数及其使用。
Lindorm CQL支持两类函数：scalar function和native aggregate function。

- scalar function：用于获取一些值并生成一个输出。
- native aggregate function：用于聚合SELECT语句的多行结果。

> **说明** Lindorm CQL提供了许多本机的硬编码函数。

常见的scalar function函数和native aggregate function函数如下。

类别	函数
scalar function	`Cast` 、 `Now` 、 `maxTimeuuid` 、 `minTimeuuid` 、Time conversion函数、Blob conversion函数等。
native aggregate function	`Count` 、 `Sum` 、 `Avg` 、 `Min` 、 `Max` 等。

Cast

Cast函数用于数据类型的转换，支持将一种本机数据类型转换为另一种。

数据类型
Cast函数支持转换的数据类型如下：

转换前	转换后
ascii	text
bigint	tinyint , smallint , int , float , double , decimal , varint , text
boolean	text
counter	tinyint , smallint , int , bigint , float , double , decimal , varint , text
date	timestamp
decimal	tinyint , smallint , int , bigint , float , double , varint , text
double	tinyint , smallint , int , bigint , float , decimal , varint , text
float	tinyint , smallint , int , bigint , double , decimal , varint , text
inet	text
int	tinyint , smallint , bigint , float , double , decimal , varint , text
time	text
timestamp	date , text
timeuuid	timestamp , date , text
tinyint	tinyint , smallint , int , bigint , float , double , decimal , varint , text
uuid	text
varint	tinyint , smallint , int , bigint , float , double , decimal , text

转换说明
通过Cast函数转换数据类型，需严格遵守Java的语义规范。

转换示例

```
CREATE TABLE persioninfo (c1 int PRIMARY KEY, c2 timeuuid);
SELECT avg(cast(c1 as double)) FROM persioninfo;
```

Now

Now函数用于返回当前的时间。

Now函数不接受任何参数，并在调用该函数时在协调器节点上会生成一个新的唯一timeuuid。因此，在使用Now函数时，在 `WHERE` 语句中基本没有意义，例如对以下表单的查询，不会按设计返回任何结果，因此 `now()` 返回的值保证是唯一的。

```
SELECT * FROM persioninfo WHERE c2 = now()
```

MaxTimeuuid和MinTimeuuid

MinTimeuuid和MaxTimeuuid函数用于获取时间戳值（可以是时间戳或日期字符串<Timestamps>），并返回与最小值或最大值时间戳T对应的Timeuuid。

示例

```
SELECT * FROM persioninfo WHERE c2 > maxTimeuuid('2013-01-01 00:05+0000')    AND c2 < minTimeuuid('2013-02-0
2 10:00+0000') ALLOW FILTERING ;
```

Datetime

Datetime相关函数用于检索调用函数时的日期和时间。

函数名	输出形式
currentTimestamp	Timestamp
currentDate	date
currentTime	time
currentTimeUUID	timeUUID

Time Conversion

Time Conversion函数用于将Timeuuid、时间戳或日期转换成Lindorm CQL自带的支持格式。

函数名	输入类型	转换描述
toDate	timeuuid	将 timeuuid 类型转换为 date 类型。
toDate	timestamp	将 timestamp 类型转换为 date 类型。
toTimestamp	timeuuid	将 timeuuid 类型转换为 timestamp 类型。
toTimestamp	date	将 date 类型转换为 timestamp 类型。
toUnixTimestamp	timeuuid	将 timeuuid 类型转换为 bigint 类型。
toUnixTimestamp	timestamp	将 timestamp 类型转换为 bigint 类型（raw）。
toUnixTimestamp	date	将 date 类型转换为 bigint 类型（raw）。

Blob Conversion

Blob Conversion系列函数支持将本机类型转换为二进制数据（Blob）。

对于Lindorm CQL支持的每个<native type>类型，函数TypeAsBlob接受一个Type类型的参数并将其作为Blob返回。相反，函数BlobAsType接受64位Blob参数并将其转换为Bigint值。例如，bigintAsBlob（3）是0x0000000000000003，blobAsBigint（0x0000000000000003）是3。

```
CREATE TABLE persioninfo (c1 text PRIMARY KEY, c2 bigint);
INSERT INTO persioninfo (c1, c2) VALUES ( '11', blobAsBigint(0x0000000000000003));
```

Count

Count函数用于计算给定列的非空值，还可以对查询返回的行进行计数。

示例

对查询返回的行进行计算：

```
SELECT COUNT (*) FROM persioninfo;
SELECT COUNT (1) FROM persioninfo;
```

计算给定列的非空值：

```
SELECT COUNT (c2) FROM persioninfo;
```

Max和Min

Max和Min函数用于计算给定列的查询返回的最大值和最小值。

示例

```
CREATE TABLE persioninfo (c1 text PRIMARY KEY, c2 int);
INSERT INTO persioninfo (c1, c2) VALUES ('k1', 3);
INSERT INTO persioninfo (c1, c2) VALUES ('k2', 4);
SELECT min(c2), max(c2) FROM persioninfo;
```

Sum

Sum函数用于求和，计算指定列的查询返回值之和。

示例

```
SELECT sum(c2) FROM persioninfo ;
```

Avg

Avg函数用于计算给定列查询返回值的平均值。

示例

```
SELECT avg(c2) FROM persioninfo ;
```

7.3.2.7.8. JSON支持

JSON支持SELECT和INSERT语句。但是Json format的支持并没有从根本上改变cql api的使用方式，CQL 需要的数据定义模型还是必须定义的。JSON format的使用只是提供了一种使用JSON方式的方法。Lindorm CQL支持将多个列的数据按照JSON的格式进行输入和输出。

SELECT JSON

对于SELECT语句，可以使用JSON关键字将每一行作为单个JSON编码的映射返回。SELECT语句行为的其余部分是相同的。

结果映射键与普通结果集中的列名相同。例如，`SELECT JSON a, b FROM ...` 将产生一个带有键 `a` 和 `b` 的映射。

INSERT JSON

对于INSERT语句，可以使用JSON关键字将JSON编码的映射作为一行插入。JSON映射的格式通常应该与同一个表上SELECT JSON语句返回的格式匹配。特别是，区分大小写的列名应该用双引号括起来。例如，要插入到一个有两列名为 `myKey` 和 `value` 的表中，可以执行以下操作：

```
INSERT INTO persioninfo JSON '{ "c1": "key", "c2": "value"}'
```

默认情况下或者显式使用default NULL，JSON映射中省略的列将被设置为NULL，这意味着该列的任何预先存在的值都将被删除。

JSON Encoding of CQL Data Types

在可能的情况下，Lindorm CQL将在原生JSON表示中表示和接受数据类型。Lindorm CQL还将接受与所有单个字段类型的Lindorm CQL文本格式匹配的字符串表示。例如，float、int、uuid和dates可以由Lindorm CQL文本字符串表示。

下表描述了Lindorm CQL在INSERT JSON值（和fromJson（）参数）中接受的编码，以及Lindorm CQL在返回SELECT JSON语句（和fromJson（））数据时使用的格式：

数据类型	接受的格式	返回的格式	描述
ascii	string	string	使用JSON的 `\u` 字符转义。
bigint	integer, string	integer	字符串必须是有效的64位整数。
blob	string	string	字符串应为0x，后跟偶数个十六进制数字。

数据类型	接受的格式	返回的格式	描述
boolean	boolean, string	boolean	取值为 `true` 或者 `false` 。
date	string	string	Date格式 `YYYY-MM-DD` ，时区为UTC。
decimal	integer, float, string	float	可能超过32或64位IEEE-754浮点精度。
double	integer, float, string	float	字符串必须是有效的整数或浮点。
float	integer, float, string	float	字符串必须是有效的整数或浮点。
inet	string	string	IPv4 or IPv6 address。
int	integer, string	integer	字符串必须是有效的32位整数。
smallint	integer, string	integer	字符串必须是有效的16位整数。
text	string	string	使用JSON的 `\u` 字符转义。
time	string	string	时间格式 `HH-MM-SS[.fffffffff]` 。
timestamp	integer, string	string	时间戳timestamp. 字符串可以是表示datas的格式 `YYYY-MM-DD HH:MM:SS.SSS` 。
timeuuid	string	string	Type 1 UUID。
tinyint	integer, string	integer	8位有效的整数。
uuid	string	string	uuid的格式字符串。
varint	integer, string	integer	可变长度；可能在客户端解码器中溢出32位或64位整数。

7.3.3. 通过Phoenix SQL连接并使用宽表引擎（已下线）

7.3.3.1. PhoenixSQL SDK安装

Java API的使用和连接参数请参见通过Phoenix SQL Java API连接并使用宽表引擎。

对于非Java API，PhoenixSQL采用Avatica协议，原生支持.NET/Go/Java/Python/JavaScript等非Java API访问。目前阿里云暂未提供非Java API下载，您可从Avatica官网下载开源版本进行使用。

7.3.3.2. 开通宽表SQL

宽表SQL基于Phoenix为Lindorm提供SQL能力，支持JDBC、Python、Go等多语言访问。本文介绍如何开通宽表SQL服务。

前提条件

- 请确保您已经购买宽表引擎服务。购买宽表引擎请参见创建实例。
- 已登录Lindorm控制台。

> **注意**
> 开通宽表SQL过程中会重启集群，可能造成业务访问抖动，建议在业务低峰操作。

开通宽表SQL

1. 在lindorm控制台单击目标实例**操作**列的**管理**按钮，进入**实例详情**页面，单击左侧导航栏的**宽表引擎>宽表SQL**。

2. 单击**立即开通**按钮，开通宽表SQL服务。



3. 为宽表SQL服务购买资源，选择Phoenix Core节点规格和Phoenix节点数量，勾选**云原生多模数据库Lindorm服务协议**。



4. 单击**立即购买**。
5. 购买完成后，在**实例列表**页面可以看到实例的状态变为**节点扩容中**，当实例状态变为**运行中**时，则可以开始使用该实例。

查看访问地址

开通宽表SQL后，在**Lindorm控制台>实例详情>宽表引擎>宽表SQL**页面查看客户端访问地址（私网），如需公网访问请单击**开通外网地址**，如需停止公网访问，请单击**释放外网地址**。



管理Phoenix节点

添加节点

如需添加Phoenix节点，操作步骤如下：

1. 进入**Lindorm控制台>实例详情>宽表引擎>宽表SQL**页面，在**Phoenix节点信息**区域单击**管理>添加节点**，跳转到**变配**页面。
2. 配置所需**Phoenix节点数量**，勾选**云原生多模数据库Lindorm服务协议**，单击**立即购买**。



3. 购买完成后，在**实例列表**页面可以看到实例的状态变为**节点扩容中**，当实例状态变为**运行中**时，则可以开始使用该实例。

升级规格

如需升级Phoenix Core节点规格，操作步骤如下：

1. 进入**Lindorm控制台>实例详情>宽表引擎>宽表SQL**页面，在**Phoenix节点信息**区域单击**管理>升级规格**，跳转到**变配**页面。

2. 配置所需**宽表引擎节点规格**，勾选**云原生多模数据库Lindorm服务协议**，单击**立即购买**。



3. 购买完成后，在**实例列表**页面可以看到实例的状态变为**规格变更中**，当实例状态变为**运行中**时，则可以开始使用该实例。

7.3.3.3. 连接集群

连接集群需要三个信息，连接地址，用户名，密码，您可以通过控制台获得。

前提条件

请保证已开通宽表SQL服务，具体操作请参见开通宽表SQL。

连接地址

Lindorm支持通过私网（VPC）和公网进行访问，私网地址直接从页面获取， 公网访问需要先开通外网地址。具体操作请参见查看访问地址。

白名单

为了连接安全，无论是公网访问还是私网访问，均需要将访问数据库的地址加入到白名单中。具体操作请参见设置白名单。

用户名和密码

Lindorm内置用户认证和ACL的安全体系，具体操作请参见管理用户。

Lindorm在实例创建后会默认新增用户名为root，密码为root的超级账号，拥有集群的所有权限，您可以使用该root账号访问该集群。也可以在集群管理系统中新增、删除或者修改root账号的密码，具体操作请参见用户和ACL管理。

如果不需要访问时提供用户名密码，可以在集群管理系统中关闭此功能，具体操作请参见用户和ACL管理。

7.3.3.4. 通过Phoenix SQL Java API连接并使用宽表引擎

本文介绍通过Phoenix SQL Java API访问Lindorm宽表引擎的具体操作和使用示例。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已下载PhoenixSQL SDK，下载链接请参见Phoenix。
- 已获取Lindorm宽表引擎的链接地址，具体操作请参见查看连接地址。
- 已将访问数据库的IP地址加入至云原生多模数据库Lindorm白名单中。具体操作请参见设置白名单。

操作步骤

1. 创建连接。在Project中输入下述代码，填写Lindorm宽表引擎的链接地址和用户名密码。

```
String url = "jdbc:phoenix:thin:url=http://ld-xxxxx-m1-ps-proxy-phoenix.hbaseue.rds.aliyuncs.com:8765;se
rialization=PROTOBUF";
Properties props = new Properties();
props.put("user", "root");
props.put("password", "root");
Connection pconn = DriverManager.getConnection(url, props);
pconn.setAutoCommit(true);
```

2. 通过PhoenixSQL访问Lindorm宽表引擎的示例代码。

```
Statement stmt = null;
try {
    stmt = pconn.createStatement();
    stmt.execute("create table test(c1 integer, c2 integer, constraint pk primary key(c1))");
    stmt.execute("upsert into test(c1,c2) values(1,1)");
    ResultSet rs = stmt.executeQuery("select * from test");
    while (rs.next()) {
        int c1 = rs.getInt("c1");
        int c2 = rs.getInt("c2");
        System.out.println("c1=" + c1 + ",c2=" + c2);
    }
} catch (Throwable e) {
    e.printStackTrace();
} finally {
    if (stmt != null) {
        stmt.close();
    }
}
```

7.3.3.5. 通过Phoenix SQL非Java API连接并使用宽表引擎

PhoenixSQL原生支持非Java API（例如.NET、Go、Python、JavaScript等）连接并访问Lindorm宽表引擎。
PhoenixSQL基于Avatica协议支持非Java API访问Lindorm宽表，您可以通过Avatica官网下载开源版进行使用，如果有疑问请通过技术支持解决，请参见技术支持。

7.3.3.6. 通过Phoenix Sqlline连接并使用宽表引擎

Sqlline是PhoenixSQL的命令行工具，本文介绍通过Sqlline连接并使用Lindorm宽表引擎的具体操作。

前提条件

- 已获取Lindorm宽表引擎的PhoenixSQL连接地址，具体操作请参见查看连接地址。
- 已将访问数据库的IP地址加入至云原生多模数据库Lindorm白名单中。具体操作请参见设置白名单。

Sqlline的常用命令

命令	描述
`!help`	查看帮助。
`!q`	退出Sqlline命令环境。
`!tables`	查看所有表。
`!desc table`	查看表的详细信息。
`!indexes table`	查看表的索引列。
`select`	查询语句操作。
`upsert`	插入数据。
`delete`	删除操作。

操作步骤

1. 下载Phoenix Sqlline。

```
wget https://hbase-opt.oss-cn-hangzhou.aliyuncs.com/ali-phoenix-5.2.4.1-HBase-2.x.tar.gz
```

2. 解压压缩包。

```
tar zxvf ali-phoenix-5.2.4.1-HBase-2.x.tar.gz
```

3. 解压后的文件名为*ali-phoenix-5.2.4.1-HBase-2.x*，进入bin目录。

```
cd ali-phoenix-5.2.4.1-HBase-2.x/bin
```

4. 访问Lindorm宽表引擎，填写Lindorm宽表引擎的PhoenixSQL连接地址和用户名密码。

```
./sqlline-thin.py http://ld-xxxxx-m1-ps-proxy-phoenix.hbaseue.rds.aliyuncs.com:8765 -u root -p root
```

7.3.3.7. 使用限制

Lindorm高度兼容PhoenixSQL语法，但目前有少量语法不支持。
关于PhoenixSQL语法介绍请参见PhoenixSQL。

- CREATE TABLE时不支持指定DEFAULT VALUE，不支持ARRAY数据类型，不支持设置MULTI_TENANT，SALT_BUCKETS，DISABLE_WAL，DEFAULT_COLUMN_FAMILY，TRANSACTIONAL属性。
- 不支持CREATE/DROP VIEW。
- ALTER TABLE 时不支持DROP COLUMN，不支持修改MULTI_TENANT，SALT_BUCKETS，DISABLE_WAL，DEFAULT_COLUMN_FAMILY， TRANSACTIONAL属性。
- 不支持CREATE/DROP LOCAL INDEX。
- 不支持CREATE/DROP FUNCTION，即用户自定义函数UDF。
- DELETE时必须指定WHERE条件。

如对以上的使用限制存在疑问或需求，欢迎提工单，或者加入钉群咨询，具体请参见技术支持。

7.3.3.8. PhoenixSQL与Spark的选择

Lindorm上的PhoenixSQL适用于轻量级分析。
PhoenixSQL与Spark的对比参考：HBase SQL（Phoenix）与Spark的选择。

另外， Spark也可集成Phoenix SQL， 具体方案可加入钉群咨询（参考：技术支持）。

7.3.4. 通过S3协议连接并使用宽表引擎

7.3.4.1. S3兼容协议使用文档

Lindorm宽表引擎兼容了S3协议的常用接口，便于各类基于AWS S3协议开发的应用及服务的接入。只要相关接口实现了兼容，您就能在应用和服务中使用对应的功能，对接Lindorm宽表引擎。
使用限制以及接口的具体兼容情况，请参考使用限制以及兼容API。
S3兼容协议默认为关闭状态。如果您需要使用该功能，请参考开通S3协议兼容功能，并参考获取连接地址来获取通过S3协议连接Lindorm宽表引擎所需要的地址。
S3协议支持多种语言，包括但不限于Java、C++、Go、Python、Nodejs、.NET、PHP、JavaScript、Ruby等。通过S3多语言SDK可以兼容的访问Lindorm，本章节介绍常见的Java、Python、Go三种语言的访问方式。

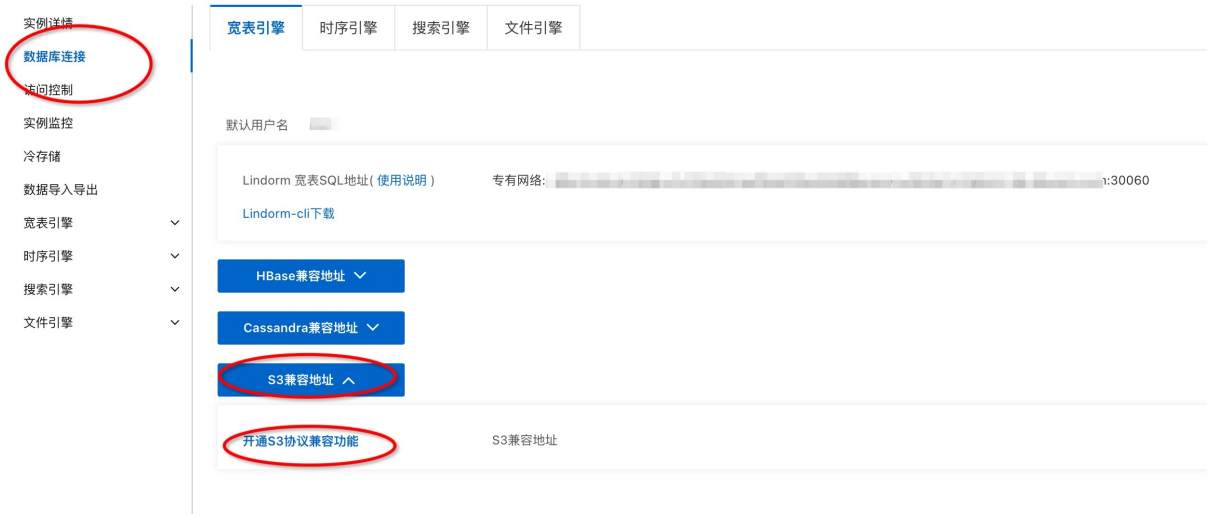
- S3 Java使用和连接参数请参见通过S3 Java API连接并使用宽表引擎。
- S3 多语言（Python、Go）使用和连接参数请参见通过S3 非Java API连接并使用宽表引擎。

7.3.4.2. 开通S3协议兼容功能

默认情况下S3协议兼容功能是关闭状态。本文介绍了开通S3协议兼容功能的步骤。

开通S3协议兼容功能

1. 登录Lindorm控制台。
2. 单击目标实例ID，进入**实例列表**页面。
3. 单击左侧导航栏**数据库连接**，默认**宽表引擎**页面，点击**S3兼容地址**，如下图所示。
4. 单击**开通S3协议兼容功能**，稍等几分钟，会显示**S3兼容地址**用于连接Lindorm宽表引擎。



7.3.4.3. 获取连接地址

本文介绍获取S3协议连接Lindorm宽表引擎时所需的连接地址，以及如何添加白名单。

获取连接地址

您可以通过云原生多模数据库Lindorm控制台获取S3协议访问Lindorm宽表引擎时需要的连接地址，具体操作请参见查看宽表引擎连接地址。

添加白名单

为了连接安全，您需要将访问数据库的地址加入到白名单中。具体操作请参见设置白名单。

7.3.4.4. 通过S3 Java API连接并使用宽表引擎

本文介绍通过S3 Java API连接并使用云原生多模数据库Lindorm的具体操作和使用示例。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已获取云原生多模数据库Lindorm宽表引擎的S3兼容连接地址，具体操作请参见查看连接地址。



- 已将客户端地址添加至云原生多模数据库Lindorm实例的白名单中，具体操作请参见设置白名单。

操作步骤

1. 安装S3 Java SDK。打开Eclipse客户端，创建一个Project并在*pom.xml*中配置Maven依赖，具体内容如下：
   - 安装Java SDK 1.x Maven依赖

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-java-sdk-s3</artifactId>
  <version>1.11.655</version>
</dependency>
```

- 安装Java SDK 2.x Maven依赖

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>aws-sdk-java</artifactId>
  <version>2.17.32</version>
</dependency>
```

2. 在Project中输入下述代码连接并访问Lindorm宽表引擎。

Java SDK 1.x代码示例

- 创建连接

```
String s3Endpoint = "http://ld-bp17j28j2y7pm****-proxy-blob.lindorm.rds.aliyuncs.com:9053"; //Lindorm宽表引擎的S3兼容地址
String bucketName = "testbucket";

// 创建连接
AmazonS3 client = AmazonS3ClientBuilder.standard()
        .withEndpointConfiguration(new AwsClientBuilder.EndpointConfiguration(s3Endpoint, null))
        .withPathStyleAccessEnabled(true)
        .withCredentials(new AWSStaticCredentialsProvider(new BasicAWSCredentials("1", "1")))
        .build();
```

- Bucket操作

```
// 创建bucket
Bucket bucket = client.createBucket(bucketName);

// 获取bucket是否存在
HeadBucketResult result = client.headBucket(new HeadBucketRequest(bucketName));

// 列举所有bucket
List<Bucket> buckets = client.listBuckets(new ListBucketsRequest());
```

- Object操作

```
String content = "content";
// 上传object
client.putObject(bucketName, key_name, content);

// 读取object
S3Object object = client.getObject(bucketName, keyName);

// 列举bucket内所有object
// list v1
ObjectListing objects = client.listObjects(bucketName);
// list v2
ListObjectsV2Result results = client.listObjectsV2(bucketName);

// 删除object
client.deleteObject(bucketName, keyName);

// 批量删除object
client.deleteObjects(new DeleteObjectsRequest(bucketName).withKeys(keyName));
```

- Multipart Upload操作

```
// 要上传的文件
File file = new File(filePath);
long contentLength = file.length();
long partSize = 5 * 1024 * 1024; // 设置分片大小为5MB

List<PartETag> partETags = new ArrayList<PartETag>();

// 初始化分片上传
InitiateMultipartUploadRequest initRequest = new InitiateMultipartUploadRequest(bucketName, keyName);
InitiateMultipartUploadResult initResponse = client.initiateMultipartUpload(initRequest);

// 上传分片
long filePosition = 0;
for (int i = 1; filePosition < contentLength; i++) {
    partSize = Math.min(partSize, (contentLength - filePosition));

    // 分片上传请求
    UploadPartRequest uploadRequest = new UploadPartRequest()
        .withBucketName(bucketName)
        .withKey(keyName)
        .withUploadId(initResponse.getUploadId())
        .withPartNumber(i)
        .withFileOffset(filePosition)
        .withFile(file)
        .withPartSize(partSize);

    UploadPartResult uploadResult = client.uploadPart(uploadRequest);
    partETags.add(uploadResult.getPartETag());

    filePosition += partSize;
}

// 完成分片上传，object可见
CompleteMultipartUploadRequest compRequest = new CompleteMultipartUploadRequest(bucketName, keyName,
                                                                                 initResponse.getUploadId()
, partETags);
client.completeMultipartUpload(compRequest);
```

Java SDK 2.x代码示例

- 创建连接

```
String s3Endpoint = "http://ld-bp17j28j2y7pm****-proxy-blob.lindorm.rds.aliyuncs.com:9053"; //Lindorm宽表引擎的S3兼容地址
String bucketName = "testbucket";

// 创建连接
AwsBasicCredentials creds = AwsBasicCredentials.create("1", "1");

S3Client client = S3Client.builder()
    .serviceConfiguration(b -> b.checksumValidationEnabled(false))
    .region(Region.AP_EAST_1)
    .credentialsProvider(StaticCredentialsProvider.create(creds))
    .endpointOverride(new URI(s3Endpoint))
    .build();

// 关闭连接
client.close();
```

- Bucket操作

```
// 创建bucket
S3Waiter s3Waiter = client.waiter();
CreateBucketRequest bucketRequest = CreateBucketRequest.builder()
    .bucket(bucketName)
    .build();

// 获取bucket是否存在
client.createBucket(bucketRequest);
HeadBucketRequest bucketRequestWait = HeadBucketRequest.builder()
    .bucket(bucketName)
    .build();

WaiterResponse<HeadBucketResponse> waiterResponse = s3Waiter.waitUntilBucketExists(bucketRequestWait);
waiterResponse.matched().response().ifPresent(System.out::println);
```

- Object操作

```
// 写object
PutObjectRequest putOb = PutObjectRequest.builder()
        .bucket(bucketName)
        .key(keyName)
        .build();

PutObjectResponse response = client.putObject(putOb,
        RequestBody.fromString("content"));

// 读object
GetObjectRequest objectRequest = GetObjectRequest
        .builder()
        .key(keyName)
        .bucket(bucketName)
        .build();

ResponseBytes<GetObjectResponse> objectBytes = client.getObjectAsBytes(objectRequest);
    byte[] data = objectBytes.asByteArray();

// 列举object
ListObjectsRequest listObjects = ListObjectsRequest
                    .builder()
                    .bucket(bucketName)
                    .build();

ListObjectsResponse res = client.listObjects(listObjects);
List<S3Object> objects = res.contents();
```

- Multipart Upload操作

```
// 初始化分片上传
CreateMultipartUploadRequest createMultipartUploadRequest = CreateMultipartUploadRequest.builder()
    .bucket(bucketName)
    .key(keyName)
    .build();

CreateMultipartUploadResponse response = client.createMultipartUpload(createMultipartUploadRequest);
String uploadId = response.uploadId();
System.out.println(uploadId);

// 上传分片1
UploadPartRequest uploadPartRequest1 = UploadPartRequest.builder().bucket(bucketName).key(keyName)
    .uploadId(uploadId)
    .partNumber(1).build();
String etag1 = client.uploadPart(uploadPartRequest1, RequestBody.fromString("content1")).eTag();
CompletedPart part1 = CompletedPart.builder().partNumber(1).eTag(etag1).build();

// 上传分片2
UploadPartRequest uploadPartRequest2 = UploadPartRequest.builder().bucket(bucketName).key(keyName)
    .uploadId(uploadId)
    .partNumber(2).build();
String etag2 = client.uploadPart(uploadPartRequest2, RequestBody.fromString("content2")).eTag();
CompletedPart part2 = CompletedPart.builder().partNumber(2).eTag(etag2).build();

// 完成分片上传，object可见
CompletedMultipartUpload completedMultipartUpload = CompletedMultipartUpload.builder()
    .parts(part1, part2)
    .build();

CompleteMultipartUploadRequest completeMultipartUploadRequest =
    CompleteMultipartUploadRequest.builder()
    .bucket(bucketName)
    .key(keyName)
    .uploadId(uploadId)
    .multipartUpload(completedMultipartUpload)
    .build();

client.completeMultipartUpload(completeMultipartUploadRequest);
```

7.3.4.5. 通过S3非Java API连接并使用宽表引擎

本文介绍多语言（例如Go、Python）通过S3协议访问Lindorm宽表引擎的具体操作和使用示例。

前提条件

- 已获取云原生多模数据库Lindorm宽表引擎的S3兼容连接地址，具体操作请参见查看连接地址。



- 已将客户端地址添加至云原生多模数据库Lindorm实例的白名单中，具体操作请参见设置白名单。

操作步骤

Go SDK访问Lindorm宽表引擎示例

1. 安装Go SDK。

```
go get github.com/aws/aws-sdk-go
```

2. 使用Go SDK连接并访问Lindorm宽表引擎。

```
package main

import (
 "log"
 "strings"

 "github.com/aws/aws-sdk-go/aws"
 "github.com/aws/aws-sdk-go/aws/credentials"
 "github.com/aws/aws-sdk-go/aws/session"
 "github.com/aws/aws-sdk-go/service/s3"
)

const (
    s3endpoint = "http://ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com:9053" // Lindorm宽表引擎的S3兼容连接地址
)

func main() {
 conf := &aws.Config{
  Credentials:      credentials.NewStaticCredentials("1", "1", ""),
  Region:           aws.String("lindorm"),
  Endpoint:         aws.String(s3endpoint),
  S3ForcePathStyle: aws.Bool(true),
 }

 sess, err := session.NewSession(conf)

// 创建连接
 svc := s3.New(sess)

// 列举所有的bucket
 input := &s3.ListBucketsInput{}
```

```
 result, err := svc.ListBuckets(input)
 if err != nil {
  panic(err.Error())
 }
 println(result.String())

 bucket := "testbucketgo"
 key := "testObjectGo"

// 创建bucket
 out, err := svc.CreateBucket(&s3.CreateBucketInput{Bucket: &bucket})
 log.Println(out)
 if err != nil {
  log.Println("Failed to create bucket", err)
  return
 }

 if err = svc.WaitUntilBucketExists(&s3.HeadBucketInput{Bucket: &bucket}); err != nil {
  log.Printf("Failed to wait for bucket to exist %s, %s\n", bucket, err)
  return
 }

// 写入object
 _, err = svc.PutObject(&s3.PutObjectInput{
  Body:   strings.NewReader("Hello World!"),
  Bucket: &bucket,
  Key:    &key,
 })
 if err != nil {
  log.Printf("Failed to upload data to %s/%s, %s\n", bucket, key, err)
  return
 }
}
```

Python SDK访问Lindorm宽表引擎示例

1. 安装Python SDK。

```
pip install boto3
```

2. 使用Python SDK连接并访问Lindorm宽表引擎。

```
import boto3

s3endpoint = 'http://ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com:9053' # Lindorm宽表引擎的S3兼容连接地址

# 创建连接
s3client = boto3.client(
    's3',
    aws_access_key_id="1",
    aws_secret_access_key="1",
    endpoint_url = s3endpoint,
    config=Config(s3={'addressing_style': 'path'})
)

# 创建bucket
s3client.create_bucket(Bucket="testbucketpython")

# 列举bucket
buckets = s3client.list_buckets()

# 打印所有bucket
for bucket in buckets['Buckets']:
    print(bucket["Name"])

# 上传object
response = s3client.upload_file("demo.py", "testbucketpython", "demo.py")

# 下载object
s3client.download_file('testbucketpython', 'demo.py', 'demo.py.download')
```

7.3.4.6. 使用限制以及兼容API

通过S3协议使用Lindorm宽表引擎的使用限制及兼容API。

使用限制

Lindorm宽表引擎的S3兼容功能仅支持Path style的访问方式，不支持Virtual hosted style的访问方式，所以需要在客户端应用中进行相应设置。部分S3工具默认使用Virtual hosted style，也需要进行相应配置。具体的访问方式，请参考通过S3 Java API连接并使用宽表引擎和通过S3 非Java API连接并使用宽表引擎。

目前兼容的S3 API

以下为Blob对S3 Bucket、Object以及Multipart操作兼容的API列表。未列出的操作可提工单或者咨询Lindorm售后。

操作类型	API
Bucket操作	• CreateBucket • HeadBucket • ListBuckets
Object操作	• DeleteObject • DeleteObjects • GetObject • HeadObject • PutObject • ListObjects • ListObjectsV2

Multipart操作	• InitiateMultipartUpload • AbortMultipartUpload • CompleteMultipartUpload • ListParts • UploadPart

7.4. 企业特性

7.4.1. 冷存储

背景信息

1.

7.4.1.1. 冷存储介绍

云原生多模数据库Lindorm提供了冷存储介质，可以存储数据库中的冷数据，从而降低存储成本。本文介绍冷存储功能的特性和性能测试。

功能特性

- 存储成本低。
  存储成本仅为标准型存储成本的20%。
- 支持数据写入能力，并保证数据随时可读。
- 使用简单。
  购买Lindorm实例时选择冷存储并设置冷存储空间，然后通过创建表格语句指定将冷数据存储至冷存储介质上。
- 支持同一张表内的冷热分离功能。
  冷存储功能可以自动将表中的热数据放到读写速度快的热存储中，将不常访问的数据放到冷存储中，从而降低成本，如果您的业务上有冷热分离的需求，请参见冷热分离。

开通冷存储

开通冷存储功能，具体操作请参见开通冷存储。

性能测试

测试环境：Master: ecs.c5.xlarge, 4core 8G, 20G高效云盘*4 RegionServer: ecs.c5.xlarge, 4core 8G, 20G高效云盘*4 测试机器：ecs.c5.xlarge, 4core 8G。

写性能：

表类型	avg rt	p99 rt
热表	1736 us	4811 us
冷表	1748 us	5243 us

> **说明** 每条记录10列，每列100B，也就是单行1k， 16线程写。

随机Get性能：

表类型	avg rt	p99 rt
热表	1704 us	5923 us
冷表	14738 us	31519 us

> **说明** 关闭表的BlockCache，完全读盘。每条记录10列，每列100B，也就是单行1k。8线程读，每次读出1k。

范围Scan性能：

表类型	avg rt	p99 rt
热表	6222 us	20975 us
冷表	51134 us	115967 us

> **说明** 关闭表的BlockCache，每条记录10列，每列100B，也就是单行1k。8线程读，每次读出1k。Scan的Caching设为30。

注意事项

- 冷存储的读IOPS能力很低，所以冷存储只适合低频查询场景。
- 写入吞吐上，冷存储和标准型存储相当，可以放心写入数据。
- 冷存储不适合并发大量读请求，如果有这种行为可能会导致请求异常。
- 购买冷存储空间特别大的客户可以酌情调整读IOPS 能力，详情工单。
- 建议平均每个core节点管理冷数据不要超过30T。如果需要单个core节点管理更大数据量的冷数据，可以工单咨询优化建议。

7.4.1.2. 配置冷存储

开通云原生多模数据库Lindorm的冷存储功能后，需要配置数据库中的表或者列簇，本文介绍配置冷存储的方法。

背景信息

冷存储功能仅支持云原生多模数据库Lindorm引擎版本为2.1.8及以上。Lindorm支持在表级别设置存储属性，可以将整个表或某个列簇的Storage设为冷存储。如果设置为冷存储，那么这个表或某个列簇的数据都会存储在冷存储介质中，并不会占用该实例的默认存储空间。

配置冷存储

配置数据库中的表或者列簇为冷存储的方法有两种：

- 通过HBase Shell设置冷存储。在设置前，需要已下载和配置Shell，具体操作请参见通过HBaseue Shell连接并使用宽表引擎。
  - 创建表格时执行以下语句设置冷存储。

    ```
    CREATE 'coldTable', {NAME => 'f', STORAGE_POLICY => 'COLD'}
    ```

  - 如果表格已经创建，可以通过修改表中列簇的属性来设置冷存储的列簇。

    > **注意** 如果这个列簇中已经有数据，那么只有在major compaction之后，数据才会进入到冷存储。

    ```
    ALTER 'coldTable', {NAME=>'f', STORAGE_POLICY => 'COLD'}
    ```

  - 如果您需要将表重新设置为热存储请执行以下语句。

    ```
    ALTER 'coldTable', {NAME=>'f', STORAGE_POLICY => 'DEFAULT'}
    ```

- 通过HBase Java API设置冷存储。在设置前，需要已安装和配置SDK，具体操作请参见通过HBase Java API连接并使用宽表引擎。
  - 创建表格时执行以下语句设置冷存储。

    ```
    Admin admin = connection.getAdmin();
     HTableDescriptor descriptor = new HTableDescriptor(TableName.valueOf("coldTable"));
     HColumnDescriptor cf = new HColumnDescriptor("f");
     cf.setValue("STORAGE_POLICY", AliHBaseConstants.STORAGETYPE_COLD);
     descriptor.addFamily(cf);
     admin.createTable(descriptor);
    ```

- 如果表格已经创建，可以通过修改表中列簇的属性来设置冷存储的列簇。

  > **注意** 如果这个列簇中已经有数据，那么只有在major compaction之后，数据才会进入到冷存储。

```
Admin admin = connection.getAdmin();
TableName tableName = TableName.valueOf("coldTable");
HTableDescriptor descriptor = admin.getTableDescriptor(tableName);
HColumnDescriptor cf = descriptor.getFamily("f".getBytes());
// 设置表的存储类型为冷存储
cf.setValue("STORAGE_POLICY", AliHBaseConstants.STORAGETYPE_COLD);
admin.modifyTable(tableName, descriptor);
```

- 如果您需要将表重新设置为热存储请执行以下语句。

```
Admin admin = connection.getAdmin();
TableName tableName = TableName.valueOf("coldTable");
HTableDescriptor descriptor = admin.getTableDescriptor(tableName);
HColumnDescriptor cf = descriptor.getFamily("f".getBytes());
// 设置表的存储类型为默认存储，默认存储为热存储
cf.setValue("STORAGE_POLICY", AliHBaseConstants.STORAGETYPE_DEFAULT);
admin.modifyTable(tableName, descriptor);
```

7.4.1.3. 查看冷存储容量

本文介绍通过云原生多模数据库Lindorm控制台查看冷存储容量。

前提条件

- Lindorm实例已开通冷存储功能，具体操作请参见开通冷存储。
- 已将客户端IP添加至Lindorm实例的白名单，具体操作请参见设置白名单。

查看冷存储总空间

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**冷存储**，在**冷存储信息**区域查看冷存储的总空间。

查看表的冷存储使用大小

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**宽表引擎 > 集群管理**。
5. 在**UI访问**区域，单击**ClusterManager**进入集群管理系统。
6. 在左侧导航栏，选择**数据管理 > 表管理**。
7. 选择目标表的分组名、Namespace、表类型、表状态或者表名，单击**查询**，可以在**ColdStorageSiz**列查看冷存储的使用情况，如下图所示。



7.4.1.4. 扩容冷存储

本文介绍通过云原生多模数据库Lindorm控制台扩容冷存储。

前提条件

Lindorm实例已开通冷存储功能，具体操作请参见开通冷存储。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**冷存储**。
5. 单击**扩容冷存储**。
6. 在**变配**页面，根据需求设置**冷存储存储容量**。
7. 阅读并勾选服务协议，单击**立即购买**。

7.4.2. 冷热分离

Lindorm在同一张表里实现了数据的冷热分离，系统会自动根据用户设置的冷热分界线自动将表中的冷、热数据分类归档到冷、热存储中。

背景

在海量大数据场景下，一张表中的部分业务数据随着时间的推移仅作为归档数据或者访问频率很低，同时这部分历史数据体量非常大，比如订单数据或者监控数据，降低这部分数据的存储成本将会极大的节省企业的成本。如何以极简的运维配置成本就能为企业极大降低存储成本，阿里云Lindorm冷热分离功能应运而生，其支持冷热数据使用不同介质存储，冷存储价格仅为标准型存储（存储热数据）的20%。

Lindorm在同一张表里实现了数据的冷热分离，系统会自动根据用户设置的冷热分界线自动将表中的冷数据归档到冷存储中。在用户的访问方式上和普通表几乎没有任何差异，在查询的过程中，用户只需配置查询Hint或者TimeRange，系统根据条件自动地判断查询应该落在热数据区还是冷数据区。对用户而言始终是一张表，对用户几乎做到完全的透明。

原理介绍

用户在表上配置数据冷热时间分界点，Lindorm依赖用户写入数据的时间戳（毫秒）和时间分界点来判断数据的冷热。数据最初始在热存储上，随意时间的推移慢慢往冷数据迁移。同时用户可以**任意变更数据的冷热分界点**，数据可以从热到冷，也可以从冷到热。

使用准备

- **开通冷存储功能**
  请参见冷存储介绍中的指导开通集群的冷存储功能。
- **升级服务端版本**

  > **说明** 冷存储功能需要Lindorm服务端升级到2.1.8版本以上。

- **安装SDK和Shell**
  通过HBase shell使用该功能，请参见通过HBaseue Shell连接并使用宽表引擎文档中的步骤完成Shell的下载和配置。

通过HBase java api使用该功能，请参见通过HBase Java API连接并使用宽表引擎文档中的步骤完成Java SDK安装和参数配置。
通过Cassandra cqlsh使用该功能，请参见通过Lindorm-cqlsh连接并使用宽表引擎文档中的步骤完成安装和配置。
通过Lindorm-cli使用该功能，请参加通过Lindorm-cli连接并使用宽表引擎文档中的步骤完成安装和配置。

表设置冷热分界线

用户在使用过程中可以随时调整COLD_BOUNDARY来划分冷热的边界。COLD_BOUNDARY的单位为秒，例如COLD_BOUNDARY为86400，代表86400秒（一天）前写入的数据会被自动归档到冷存储介质上。

> **说明** 在冷热分离使用过程中，无需把表或列簇的属性设置为COLD，如果已经把列簇的属性设置为了COLD，请依照冷存储介绍文档中的指导将冷存储的属性去除。

具体操作方法，如下：

- 方式一：HBase Shell

```
// 创建冷热分离表
HBase(main):002:0> create 'chsTable', {NAME=>'f', COLD_BOUNDARY=>'86400'}
// 取消冷热分离
HBase(main):004:0> alter 'chsTable', {NAME=>'f', COLD_BOUNDARY=>""}
// 为已经存在的表设置冷热分离,或者修改冷热分离分界线，单位为秒
HBase(main):005:0> alter 'chsTable', {NAME=>'f', COLD_BOUNDARY=>'86400'}
```

- 方式二：HBase Java API

```
// 新建冷热分离表
Admin admin = connection.getAdmin();
TableName tableName = TableName.valueOf("chsTable");
HTableDescriptor descriptor = new HTableDescriptor(tableName);
HColumnDescriptor cf = new HColumnDescriptor("f");
// COLD_BOUNDARY 设置冷热分离时间分界点，单位为秒，示例表示1天之前的数据归档为冷数据
cf.setValue(AliHBaseConstants.COLD_BOUNDARY, "86400");
descriptor.addFamily(cf);
admin.createTable(descriptor);
// 取消冷热分离
// 注意：需要做major compaction，数据才能从冷存储上回到热存储上
HTableDescriptor descriptor = admin
    .getTableDescriptor(tableName);
HColumnDescriptor cf = descriptor.getFamily("f".getBytes());
// 取消冷热分离
cf.setValue(AliHBaseConstants.COLD_BOUNDARY, null);
admin.modifyTable(tableName, descriptor);
// 为已经存在的表设置冷热分离功能，或者修改冷热分离分界线
HTableDescriptor descriptor = admin
    .getTableDescriptor(tableName);
HColumnDescriptor cf = descriptor.getFamily("f".getBytes());
// COLD_BOUNDARY 设置冷热分离时间分界点，单位为秒，示例表示1天之前的数据归档为冷数据
cf.setValue(AliHBaseConstants.COLD_BOUNDARY, "86400");
admin.modifyTable(tableName, descriptor);
```

- 方式三：Cassandra CQL
  通过cqlsh可以使用cql设置lindorm的冷热分离属性，方式如下：

```
// COLD_BOUNDARY 设置冷热分离时间分界点，单位为秒，示例表示1天之前的数据归档为冷数据
create table tb (pk text PRIMARY KEY, c1 text) WITH extensions = {'COLD_BOUNDARY':'86400'}
//修改存储介质属性以及冷热分离时间线
alter table tb  with extensions = {'STORAGE_POLICY' : 'DEFAULT', 'COLD_BOUNDARY':'1000'}
```

- 方式四：lindorm-cli

```
use test;
// CHS 冷热分离时间点，单位为秒，示例表示1天之前的数据归档为冷数据，需要同时设置CHS_L2 为示例中的值，表明冷数据的存储属性为COLD。
create table dt (
  p1 integer, p2 integer, c1 varchar, c2 bigint,  constraint pk primary key(p1 desc))
'COMPRESSION'='ZSTD', 'CHS'='86400', 'CHS_L2' = 'storagetype=COLD';
```

数据写入

冷热分离的表与普通表的数据写入方式完全一致，时间戳默认为数据写入时的系统 `当前时间` 。数据先会存储在热存储（标准型/性能型）中。随着时间的推移，如果这行数据的写入时间超过 `COLD_BOUNDARY` 设置的值，就会在major_compact时归档到冷数据，此过程完全对用户透明。

数据查询

由于冷热数据都在同一张表中，用户全程只需要和一张表交互。在查询过程中，如果用户明确知道需要查询的数据在热数据里（写入时间少于COLD_BOUNDARY设置的值），可以在查询时设置 `HOT_ONLY` 的Hint来告诉服务器只查询热区数据。或者在查询时设置TimeRange来限定查询数据的时间，系统会根据设置TimeRange决定是查询热区，冷区还是冷热都查。查询冷区数据延迟要比热区数据延迟高的多，并且查询吞吐受到冷存储限制（详见冷存储介绍文档）。

查询示例：

- **随机查询Get**
  - 方式一：HBase Shell

```
// 不带HotOnly Hint的查询，可能会查询到冷数据
HBase(main):013:0> get 'chsTable', 'row1'
// 带HotOnly Hint的查询，只会查热数据部分，如row1是在冷存储中，该查询会没有结果
HBase(main):015:0> get 'chsTable', 'row1', {HOT_ONLY=>true}
// 带TimeRange的查询，系统会根据设置的TimeRange与COLD_BOUNDARY冷热分界线进行比较来决定查询哪个区域的数据（注意TimeRange的单位为毫秒时间戳）
HBase(main):016:0> get 'chsTable', 'row1', {TIMERANGE => [0, 1568203111265]}
```

  - 方式二：HBase Java API

```
Table table = connection.getTable("chsTable");
// 不带HotOnly Hint的查询，可能会查询到冷数据
Get get = new Get("row1".getBytes());
System.out.println("result: " + table.get(get));
// 带HotOnly Hint的查询，只会查热数据部分，如row1是在冷存储中，该查询会没有结果
get = new Get("row1".getBytes());
get.setAttribute(AliHBaseConstants.HOT_ONLY, Bytes.toBytes(true));
// 带TimeRange的查询，系统会根据设置的TimeRange与COLD_BOUNDARY冷热分界线进行比较来决定查询哪个区域的数据（注意TimeRange的单位为毫秒时间戳）
get = new Get("row1".getBytes());
get.setTimeRange(0, 1568203111265)
```

  - 方式三：Cassandra CQL

```
//关键字hotdata表示读取热数据，暂时cql只支持读取热数据或者全部读
select hotdata * from tb;
```

- **范围查询Scan**

> **说明** 如果scan不设置Hot Only，或者TimeRange包含冷区时间，则会并行访问冷数据和热数据来合并结果，这是由于Lindorm的Scan原理决定的。

- 方式一：HBase Shell

```
// 不带HotOnly Hint的查询，一定会查询到冷数据
Lindorm(main):017:0> scan 'chsTable',  {STARTROW =>'row1', STOPROW=>'row9'}
// 带HotOnly Hint的查询，只会查询热数据部分
Lindorm(main):018:0> scan 'chsTable',  {STARTROW =>'row1', STOPROW=>'row9',  HOT_ONLY=>true}
// 带TimeRange的查询，系统会根据设置的TimeRange与COLD_BOUNDARY冷热分界线进行比较来决定查询哪个区域的数据（注意TimeRange的单位为毫秒时间戳）
Lindorm(main):019:0> scan 'chsTable',  {STARTROW =>'row1', STOPROW=>'row9',  TIMERANGE => [0, 1568203111265]}
```

- 方式二：HBase Java API

```
TableName tableName = TableName.valueOf("chsTable");
Table table = connection.getTable(tableName);
// 不带HotOnly Hint的查询，一定会查询到冷数据
Scan scan = new Scan();
ResultScanner scanner = table.getScanner(scan);
for (Result result : scanner) {
    System.out.println("scan result:" + result);
}
// 带HotOnly Hint的查询，只会查询热数据部分
scan = new Scan();
scan.setAttribute(AliLindormConstants.HOT_ONLY, Bytes.toBytes(true));
// 带TimeRange的查询，系统会根据设置的TimeRange与COLD_BOUNDARY冷热分界线进行比较来决定查询哪个区域的数据（注意TimeRange的单位为毫秒时间戳）
scan = new Scan();
scan.setTimeRange(0, 1568203111265);
```

> **说明**
> - 冷热分离表中的冷区只是用来归档数据，查询请求应该非常的少，用户查询冷热分离表的绝大部分请求应该带上HOT_ONLY的标记（或者设置的TimeRange只在热区）。如果用户有大量请求需要去查冷区数据，则可能得考虑COLD_BOUNDARY冷热分界线的设置是否合理。
> - 如果一行数据已经在冷数据区域，但这一行后续有更新，更新的字段先会在热区，如果设置HOT_ONLY去查询这一行（或者设置的TimeRange只在热区），则只会返回这一行更新的字段（在热区）。只有在查询时去掉HOT_ONLY Hint，去掉TimeRange，或保证TimeRange覆盖了该行数据插入和更新时间，才能完整返回这一行。因此不建议对已经进入冷区的数据进行更新，如果有频繁更新冷数据的需求，则可能得考虑COLD_BOUNDARY冷热分界线的设置是否合理。

查看冷热数据的大小

> **注意** 如果数据还没有进入冷存储，有可能数据还在内存中，请执行flush，将数据刷写到盘上，再请执行major_compact完成后再查看。

在登录集群管理系统的表Tab中，可以显示某一张表的冷存储使用大小和热存储使用大小。

优先查询热数据

在范围查询（Scan）场景下，查询的数据可能横跨冷热区，比如查询一个用户的所有订单、聊天记录等。但查询的展示往往是从新到旧的分页展示，最先展示的往往是最近的热数据。在这个场景下，普通的Scan（不带Hot_Only）会并行地扫描冷热数据，导致请求性能下降。而在开启了优先查询热数据后，会优先只查热数据，只有热数据的条数不够显示（如用户点了下一页查看），才会去查询冷数据，减少冷存储的访问，提升请求响应。

开启热数据优先查询，只需在Scan上设置 `COLD_HOT_MERGE` 属性即可。该属性的含义是优先查询热存储中的数据，若热存储中的数据查完了，用户仍然在调用获取下一条数据，则会开始查询冷数据。

- 方式一：HBase Shell

```
Lindorm(main):002:0> scan 'chsTable', {COLD_HOT_MERGE=>true}
```

- 方式二：HBase Java API

```
scan = new Scan();
scan.setAttribute(AliHBaseConstants.COLD_HOT_MERGE, Bytes.toBytes(true));
scanner = table.getScanner(scan);
```

说明

- 若某一行同时包含热数据和冷数据（部分列属于热数据，部分列属于冷数据，比如部分列更新场景），开启热数据优先功能，会使得该行的查询结果会分两次返回，即scanner返回的Result集合中，对于同一个Rowkey会有两个对应的Result。
- 由于是先返回热数据，再返回冷数据，开启热数据优先功能后，无法保证后返回的冷数据结果的Rowkey一定大于先返回的热数据结果的Rowkey，即Scan得到的Result集不保序， 但热数据和冷数据的各自返回集仍保证按Rowkey有序（参见下面的demo）。在部分实际场景中， 用户可以通过Rowkey设计，保障scan的结果仍然保序，比如订单记录表，Rowkey=用户ID+订单创建时间，扫描某个用户的订单数据是有序的。

```
//假设rowkey为"coldRow"的这一行是冷数据，rowkey为"hotRow"的这一行为热数据
//正常情况下，由于Lindorm的row是字典序排列，rowkey为"coldRow"的这一行会比"hotRow"这一行先返回。
HBase(main):001:0> scan 'chsTable'
ROW                                                    COLUMN+CELL
 coldRow                                               column=f:value, timestamp=156057840000
0, value=cold_value
 hotRow                                                column=f:value, timestamp=156584880000
0, value=hot_value
2 row(s)
// 设置COLD_HOT_MERGE时，scan的rowkey顺序被破坏，热数据比冷数据先返回，因此返回的结果中，"hot"排在了"cold"的前面
HBase(main):002:0> scan 'chsTable', {COLD_HOT_MERGE=>true}
ROW                                                    COLUMN+CELL
 hotRow                                                column=f:value, timestamp=156584880000
0, value=hot_value
 coldRow                                               column=f:value, timestamp=156057840000
0, value=cold_value
2 row(s)
```

按列冷热分离

您可以指定一行中的某一列的数据作为这一行冷热分离的边界，而不是依赖数据写入的时间戳，使用方式如下：

```
use test
//创建冷热分离的表
create table dt (
  p1 integer, p2 integer, c1 varchar, c2 bigint,  constraint pk primary key(p1 desc))
'COMPRESSION'='ZSTD', 'CHS'='86400', 'CHS_L2' = 'storagetype=COLD';
//设置列c2为冷热分离的基准列
alter table dt set 'CHS_COLUMN'='COLUMN=c2'
```

说明

- 按列冷热分离目前只支持LindormSQL，不支持HBase兼容使用方式。
- 指定的列需要为Long类型，同时存的值应该为13位的时间戳，如果指定列不存在，数据没有写入，或者写入的数据类型不对，则不进行冷热分离，数据都写入热存。
- 指定按列做冷热分离，则当这一列里的值超过了设定的冷热分界线之后，会将整行数据归档到冷存。比如c2里存的long 与当前时间差为2天，冷热分界线是1天，c1的时间戳（没有指定则默认为写入到系统的时间）与当前时间差为1小时，则这一行都会在major compaction的时候归档到冷存。
- 按列做冷热分离不适用于按冷热边界自动归档。只能在major compaction时做数据归档，如您需要调整冷热归档频率，请修改major compaction周期。

Q&A

- 数据什么时候进冷存？

Lindorm通过compaction机制异步得将冷数据从热存中归档到冷存，触发时间默认为冷热分界的一半，最小为1天。比如您设置冷热边界是3天，那么默认1.5天就会自动触发一次compaction；如果设置冷热边界为1天，后台会1天触发一次compaction。

- 可以手动强制触发compaction吗？
  可以的，您可以通过HBase shell对表执行 major_compact 'tableName' 命令来强制触发major compaction，不仅把热存中的冷数据归档到冷存中，同时也会合并所有冷区热区的文件。但请不要频繁使用该命令，它会加重IO负载。
- 为什么表冷数据归档的慢？
  请检查您的Lindorm版本，如果版本不大于2.1.20，可以升级到最新的版本，冷数据会根据 1中提到的周期进入冷存。如果您选择暂缓升级，可以联系技术支持来帮您优化参数。

注意事项

请参见冷存储介绍的使用注意事项。

7.4.3. 管理动态列

云原生多模数据库Lindorm宽表引擎支持动态列功能，您可以直接使用Lindorm SQL对动态列进行管理操作。本文介绍通过Lindorm SQL管理动态列，包括开启动态列、写入动态列和查询动态列。

背景信息

传统关系型数据库表的Schema必须预先定义，如果要增加列，则需要变更表属性。变更大表的表属性是一个非常耗时的操作。同时，预定义的Schema给业务的设计会带来很多不便。但是Lindorm宽表引擎原生支持动态列，列无需提前定义，您可以直接使用Lindorm SQL来对动态列进行读写操作。

注意事项

如果您需要使用动态列功能，需要注意以下几点：

- 确保云原生多模数据库Lindorm宽表引擎为2.2.19及以上版本，具体操作请参见升级小版本。
- Lindorm宽表中动态列类型均为VARBINARY，查询动态列和写入动态列时都必须将动态列类型转化成BYTE数组。
- 通过Lindorm SQL管理动态列，查询动态列和写入动态列操作支持以云数据库HBase兼容方式创建和写入的表。

开启动态列

> 注意　动态列功能开启后无法关闭。

动态列的开启可以通过以下两种方式：

- 在创建表格时通过设置'DYNAMIC_COLUMNS' = 'true'开启动态列功能。

```
CREATE TABLE tablename (p1 INT, c1 INT, c2 VARCHAR, PRIMARY KEY(p1)) 'DYNAMIC_COLUMNS' = 'true';
```

- 通过修改表的属性开启动态列功能。

```
ALTER TABLE tablename SET 'DYNAMIC_COLUMNS' = 'true';
```

> **说明**　开启动态列后，您可以修改表的属性或者在表的Schema中增加新的列，例如执行以下语句表示新增c3列，数据类型为INT。
>
> ```
> ALTER TABLE t_dynamic_columns ADD COLUMN c3 int;
> ```
>
> 如果您在此之前写入c3动态列，由于写入的数据类型均为VARBINARY，查询数据和写入数据时会抛出 `IllegalDataException` 异常，所以变更动态列表的Schema时需要注意数据类型不同的情况。如果新增列的数据类型为VARBINARY就不会出现这个异常。

写入动态列

写入动态列表的语法与写入普通列表的语法一致，开启动态列之后，可以写入没有预先在Schema中定义的列，但是动态列的类型只能为VARBINARY （即BYTE数组）。如果您需要将字符串或者数值型的值写入动态列中，同时将这些值编码成BYTE数组，那么推荐使用以下代码写入动态列。

```
Connection conn = DriverManager.getConnection(lindorm-jdbc-url);
String createTable = "CREATE TABLE testTable (p1 VARCHAR, c1 INT, PRIMARY KEY(p1)) 'DYNAMIC_COLUMNS' = 'true
'";
Statement statement = conn.createStatement();
statement.execute(createTable);
//插入3列，其中p1，c1为Schema预先定义好的列，c2没有预先定义，为动态列写入。
String sqlUpsert = "upsert into " + tableName + "(p1, c1, c2) values(?, ?, ?)";
try (PreparedStatement stmt = conn.prepareStatement(sqlUpsert)) {
    stmt.setString(1, "pk");
    stmt.setInt(2, 4);
    stmt.setBytes(3, new byte[] {0,1});
    int updated = stmt.executeUpdate();
    Assert.assertEquals(1, updated);
}
```

如果您使用Lindorm-cli或者使用SQL写入动态列时，动态列的值必须为HexString（HexString表示字节数据的十六进制的字符串）。一个字节的范围是0～255，十六进制表示为0x00~0xFF，那么对于一个字节数组{0x00, 0xFF}，它的HexString就是00FF，将BYTE数组转换为HexString的代码请参见BYTE数组转换为HexString。

写入动态列的场景示例如下：

- 执行以下语句在 `t_dynamic_columns` 表中写入c3列，c3列为动态列，写入成功。

  ```
  UPSERT INTO t_dynamic_columns (p1, c2, c3) VALUES (1, '1', 'ff');
  ```

- 执行以下语句在 `t_dynamic_columns` 表中写入c4列，c4列为动态列，写入成功。

  ```
  UPSERT INTO t_dynamic_columns (p1, c4) VALUES (2, 'ef0011');
  ```

- 执行以下语句在 `t_dynamic_columns` 表中写入c5列，c5列为动态列。由于动态列c5的值 `f` 不是偶数长度的HexString，所以写入失败，需要将 `f` 修改为 `0f` 。

  ```
  UPSERT INTO t_dynamic_columns (p1, c5) VALUES (3, 'f');
  ```

- 执行以下语句在 `t_dynamic_columns` 表中写入c6列，c6列为动态列。由于动态列c6的值 `gf` 不是HexString，所以写入失败。

  ```
  UPSERT INTO t_dynamic_columns (p1, c6) VALUES (4, 'gf');
  ```

查询动态列

查询动态列的场景分为以下几种：

- 查询字段为动态列。查询动态列表的语法与查询普通列表的语法一致，开启动态列之后，可以查询没有预先在Schema中定义的列，如下示例，c3和c4为动态列。

  ```
  SELECT p1, c3, c5 FROM t_dynamic_columns WHERE p1 = 1;
  ```

  返回结果如下：

  ```
  +-----+-------+---------+
  | p1  |   c3  |   c5    |
  +-----+-------+---------+
  |  1  | [255] |  null   |
  +-----+-------+---------+
  1 rows in set (43 ms)
  ```

- 使用 `SELECT *` 查询动态列表时，Lindorm SQL为了保证结果集元数据的正确性，需要在查询语句后加Limit。

  ```
  SELECT * FROM t_dynamic_columns LIMIT 10;
  ```

  返回结果如下：

```
+-----+-------+---------+-------+------------+
| p1  |   c1  |   c2    | c3    |     c4     |
+-----+-------+---------+-------+------------+
|  1  | null  |  1      | [255] | null       |
|  2  | null  |  null   | null  | [239 0 17] |
+-----+-------+---------+-------+------------+
2 rows in set (141 ms)
```

> **说明** 对于动态列表的 `SELECT *` 查询操作，LIMIT的默认最大值为5000，超过最大值会报错。

- 在 `WHERE` 条件中使用动态列。为了确保查询语句的性能 `WHERE` 条件中需包含主键或索引列，如果您使用Lindorm-cli或者使用SQL查询动态列时，动态列的值必须为HexString，将BYTE数组转换为HexString的代码请参考BYTE数组转换为HexString。c4为动态列，查询成功的语句如下示例：

```
SELECT p1, c4 FROM t_dynamic_columns WHERE p1 >= 1 AND p1 < 3 AND c4 = 'ef0011';
```

由于动态列c4的值 `1` 不是HexString，所以查询失败。

```
SELECT p1, c1, c4 FROM t_dynamic_columns WHERE p1 = 2 AND c4 = '1';
```

BYTE数组转换为HexString

使用以下代码可以将BYTE数组转换为HexString。

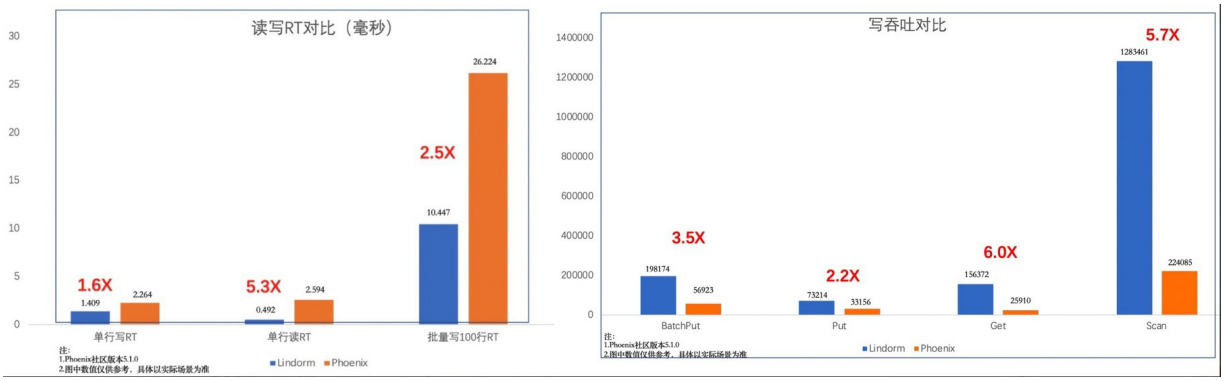
```
private static final char[] DIGITS = {
  '0', '1', '2', '3', '4', '5', '6', '7',
  '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
  };
private static String toHexString(byte[] bytes) {
  char[] chars;
  int j = 0;
  chars = new char[bytes.length * 2];
  for (byte b : bytes) {
    chars[j++] = DIGITS[(b & 0xF0) >> 4];
    chars[j++] = DIGITS[b & 0x0F];
  }
  return new String(chars, 0, j);
}
public void testToHexString() {
  String s = "Hello, world";
  // 对于字符串类型，可以直接使用String的getBytes方法获得对象对应的byte[]
  byte[] bytes = s.getBytes(Charset.forName("UTF-8"));
  String hexString = toHexString(bytes);
  System.out.println(hexString); //打印结果为: 48656c6c6f2c20776f726c64
}
```

7.4.4. 高性能原生二级索引

Lindorm宽表支持Tabular模型下的二级索引功能，此功能在非主键匹配的查询场景下，可以降低应用的开发复杂性、保证数据的一致性和提高写入效率。本文介绍Lindorm Tabular模型下二级索引的基本特性和使用示例。

背景信息

对于Lindorm Tabular模型，Lindorm宽表有表结构且表对应的列有类型预定义的。Lindorm原生二级索引功能在阿里云应用多年，经历了多次双11的考验，更加适用于解决海量数据的全局索引。Lindorm与Phoenix在索引场景下的性能对比如下图所示。



特性介绍

Lindorm二级索引支持为单表建多个索引，每个索引在物理上映射为一张数据表，与主表相互独立，每个索引有不同的存储策略（如采用不同的压缩算法、冷热分离策略）等属性。写主表时，Lindorm会自动更新所有索引表，并确保主表和索引表数据的一致性。读数据时，您只需针对主表发起查询，Lindorm会根据WHERE条件和SCHEMA自动选择合适的索引（包括主表）执行查询操作（支持HINT来干预优化器行为）。Lindorm二级索引的基本特性如下。

- 支持单个主表建多个索引。
- 支持组合索引（单列和多列）。
- 支持冗余索引，全冗余索引可自动冗余主表新增的列。
- 查询优化：根据WHERE语句自动选择索引，支持HINT来干预优化器的选择。
- Online Schema Change：索引变更不影响主表的正常读写，可以随时新增、删除、更新索引。
- 支持数据有效期（简称：TTL）：索引表自动继承主表的TTL设置，主表和索引表数据同时过期。
- 支持动态列：支持写入动态列和冗余动态列。
- 支持自定义数据版本：自定义时间戳后自动写入数据。
- 方便使用：与RDBMS的二级索引比较，使用Lindorm的二级索引更加方便，当您购买完Lindorm就可以直接通过Lindorm宽表SQL使用二级索引。

使用要求

- 服务器：使用Lindorm实例。
- 客户端：更多信息，请参见通过Lindorm宽表SQL使用宽表引擎。
- Lindorm-Cli客户端：更多信息，请参见通过Lindorm-cli连接并使用宽表引擎。

基本概念

- 强一致性：是指主表和索引表的数据一致性。为了满足数据一致性的需求，最大程度减少二级索引的额外开销，提升高吞吐的写入能力，Lindorm二级索引在强一致性中具有如下约束。
  - 不支持快照隔离性。在数据写入过程中不保证同时可见，但是返回客户端写入成功后主表和索引表数据可见。
  - 在返回客户端超时或IO出错情况下，该数据在主表和索引表中不保证可见，但最终主表和索引表数据会保持一致。
- 可选的索引组织成本：在主表有一个索引的情况下，写放大有四次操作（读主表，删除索引老数据，写索引，写主表），会大大增加维护索引的成本。但不是所有场景都会产生写放大，比如日志场景只有数据插入没有更新操作，此时索引表不存在老数据，只需要做写索引和写主表操作。所以Lindorm提出了Mutability的概念。Mutability是指对主表的写入模式进行分类，并以此组织索引数据，针对不同的需求实现最低的索引组织成本。Mutability分类和描述如下表所示。Mutability属性需要在创建表或修改表时通过Table_Options参数进行设置，具体操作，请参见CREATE TABLE语法介绍。

Mutability分类	约束	操作成本	操作说明
无索引	无。	1	没有索引的情况下，直接写索引，为1次操作。
IMMUTABLE	整行写入，不可更新或删除。	2	写主表，写索引表。所有场景中成本最低，性能最好。

Mutability分类	约束	操作成本	操作说明
IMMUTABLE_ROWS	整行写入，不可更新，删除。	2-3	正常写入时：写主表，写索引表。在删除场景下需额外读一次主表。性能仅次于IMMUTABLE。
MUTABLE_LATEST	可更新，可删除。	4	读主表，删索引，写索引，写主表。
MUTABLE_ALL	无限制，可使用自定义时间戳写入。	4	读主表，删索引，写索引，写主表。

> **说明**
> - IMMUTABLE和IMMUTABLE_ROWS不涉及数据更新，无写放大问题，成本最低。适合高吞吐写的场景，如日志、监控等。
> - IMMUTABLE不涉及删除，所以可充分利用Lindorm的多IDC部署，实现多活的数据访问。
> - 选择两类IMMUTABLE可以有效降低索引场景下的写时延，提高整体吞吐。实际的业务中，如果不满足IMMUTABLE场景，可以通过数据冗余来改造成IMMUTABLE场景。

- 更新自定义时间戳的索引：Lindorm支持自定义时间戳进行写入，可以在任意时间戳进行数据更新，由系统来判断只有时间戳最大的数据生效。自定义时间戳特性在控制数据有效期、乱序、幂等等场景中发挥着重要的作用，在HBase中有广泛的应用。Lindorm支持列级时间戳，主表支持自定义时间戳写入数据。但在支持二级索引和时间戳的NoSQL系统中，支持自定义时间戳索引更新的，就比较罕见了。因为时间戳乱序写入很难有效维护索引数据的更新和删除。Lindorm全局二级索引解决了这个问题，支持列级别自定义时间戳更新。下面是两个使用自定义时间戳的实际业务场景。
  - 导入与实时并存：在需要同时实时更新和历史数据导入的场景下，实时更新可以使用当前时间，而历史数据导入可以使用昨天23:59:59这个时间。所以当天未更新过的数据可以通过导入操作而更新，已更新过的数据也不会被导入覆盖。
  - 追消息：业务系统通过消息来触发一系列处理逻辑，在消息出现积压时，系统可以跳过积压的消息，直接处理当前消息，并在事后通过追消息来处理之前积压的任务。或者，当业务逻辑有问题时，系统也可以跳过一部分消息来规避问题，在业务修复后重新处理。此时，业务可以通过使用消息本身的时间来写入数据，以此来保证追的消息和正常消息的准确覆盖关系。
- 全冗余索引：为了避免查询索引之后再回查主表，通常会在索引表中冗余一部分主表列，也称为冗余索引或覆盖索引。全冗余索引是比较常用的冗余方案。Lindorm支持三种冗余模式，可以在主表Schema变化以及动态列场景下简单的实现全冗余索引。
  - 冗余指定的列：显示指定要冗余主表的哪些列。
  - 冗余主表Schema中的所有列：当您需要全冗余索引时，不需要在CREATE INDEX中将主表的每一列都显式添加进来，而是通过一个常量来描述冗余所有列，当主表新增列时，全冗余索引表会自动冗余这个新增列，无需重建索引。也无需担心新增列的查询会导致回查主表了。
  - 冗余动态列：Lindorm支持固定Schema和松散Schema（即动态列）。通过DYNAMIC冗余模式，索引表能够自动冗余主表中的所有动态列，也会冗余主表Schema中的所有列。

语法参考

关于Lindorm宽表SQL的语法使用，请参见Lindorm宽表SQL语法手册。

创建二级索引

创建完Lindorm主表后，可以为主表的相应列创建二级索引。以下是创建二级索引的示例。

```
--  创建主表
create table shop_item_relation (
    shop_id varchar,
    item_id varchar,
    status varchar,
constraint pk primary key(shop_id, item_id)) ;
-- 对第二列主键建索引，冗余所有列
create index idx1 on shop_item_relation (item_id) 'INDEX_COVERED_TYPE'='COVERED_ALL_COLUMNS_IN_SCHEMA';
-- 基于索引表进行查询，因为对item_id构建索引，当指定item_id 进行查询会对应命中索引表
select * from shop_item_relation where item_id = 'item2';
```

> **说明**
> - 创建索引有同步创建和异步创建两种方式，如果在存量数据不大的情况下，可以使用同步创建。其他情况下可以使用异步创建。具体的语法请参见CREATE INDEX。
> - 在已有数据的表中添加新的索引时，CREAT_INDEX命令会同时将主表的历史数据同步到索引表中，如果主表很大时，CREAT_INDEX会非常耗时间（数据同步任务是在服务端执行的，即使删除Lindorm Shell进程也不会影响数据同步任务）。

查看二级索引

通过Lindorm宽表SQL可以查看创建的二级索引状态。以下是查看二级索引的示例。

```
show index from shop_item_relation;
```

> **说明** 通过示例可以展示出shop_item_relation主表下创建的索引名和索引类型。

修改二级索引状态

创建完二级索引后，如果主表有存量数据，需要手动对索引进行一次Rebuild操作，具体语法请参见BUILD INDEX；若主表没有存量数据，则可以直接使用修改二级索引语法来修改二级索引表状态。以下是修改二级索引状态的示例。

```
alter index idxs1 on shop_item_relation ACTIVE;
alter index idxs1 on shop_item_relation DISABLED;
```

删除二级索引

通过如下示例删除对应主表中的相关二级索引。

```
drop index idxs1 on shop_item_relation ;
```

> **说明** 删除索引操作需要您有Trash权限。

查询优化

Lindorm依据RBO（Rule Based Optimization）策略进行二级索引选择。根据查询条件匹配索引表的前缀，选择匹配程度最高的索引表作为本次查询使用的索引。通过以下示例可以更好的理解。

```
//主表和索引表如下
create table dt (rowkey varchar, c1 varchar, c2 varchar, c3 varchar, c4 varchar, c5 varchar, constraint pk p
rimary key(rowkey));
create index idx1 on dt (c1);
create index idx2 on dt (c2, c3, c4);
create index idx3 on dt (c3) include (c1, c2, c4);
create index idx4 on dt (c5) 'INDEX_COVERED_TYPE'='COVERED_ALL_COLUMNS_IN_SCHEMA';
//查询优化如下
select rowkey from dt where c1 = 'a';
select rowkey from dt where c2 = 'b' and c4 = 'd';
select * from dt where c2 = 'b' and c3 >= 'c' and c3 < 'f';
select * from dt where c5 = 'c';
```

> 说明
> - select rowkey from dt where c1 = 'a'表示选择索引表idx1。
> - select rowkey from dt where c2 = 'b' and c4 = 'd'表示选择索引表idx2，从中查找所有满足c2=b条件的行，然后逐行按c4=d进行过滤。虽然c4是索引列之一，但因WHERE条件中缺少c3列，无法匹配idx2的前缀。
> - select * from dt where c2 = 'b' and c3 >= 'c' and c3 < 'f'表示选择索引表idx2，由于是select *，而索引表里并未包含主表的所有列，因此在查询索引之后，还要回查一次主表。回查主表时，回查的Rowkey可能散布在主表的各个地方，因此，可能会消耗多次RPC。回查的数据量越大，RT越长。
> - select * from dt where c5 = 'c'表示选择索引表idx4，idx4是全冗余索引，所以select *不需要回查主表。

使用限制

- 不同主表可以有同名索引，如dt表有索引Idx1，foo表也有索引Idx1，但同一主表下不允许有同名索引。
- 只能为Version为1的表建索引，不支持为多版本的表建索引。
- 对有TTL的主表建索引，不能单独为索引表设置TTL，索引表会自动继承主表的TTL。
- 索引列最多不超过3个。
- 索引列和主表主键，总长度不能超过30 KB。不建议使用大于100字节的列作为索引列。
- 单个主表的索引表个数最多不超过5个，过多索引会造成存储成本过高，以及写入耗时过长。
- 一次查询最多能命中一个索引，不支持多索引联合查询（Index Merge Query）。
- 创建索引时会将主表的数据同步到索引中，对数据多的表建索引会导致Create_Index命令耗时过长。
- 二级索引不支持batch increase功能。
- 命中二级索引的排序与主表不一样。

问题答疑

对于索引使用上的任何问题，您可以通过钉钉联系云Lindorm答疑或工单咨询，具体操作，请参见技术支持。

7.4.5. 搜索索引

7.4.5.1. 特性介绍

索引是加速数据库查询的重要方法。云原生多模数据库Lindorm宽表引擎除了支持高性能原生二级索引，也支持一种新的索引类型，称为搜索索引（SearchIndex），搜索索引主要面向复杂的多维查询场景，能够覆盖分词、模糊查询、聚合分析、排序翻页等场景。本文介绍搜索索引的功能和应用场景。

功能介绍

搜索索引是宽表的一个特性，在开发使用体验上与高性能原生二级索引保持一致，您可以使用SQL为宽表创建搜索索引。示例如下，为dt表的c1, c2, c3列创建搜索索引 `idx` ，c1, c2, c3列没有顺序要求，c3指定按照IK分词器进行分词。

```
CREATE SEARCH INDEX idx ON dt(c1, c2, c3(type=text, analyzer=ik));
```

搜索索引可以满足以下业务需求：

- 多维组合查询。c1, c2, c3列的随机组合，快速返回查询数据。

  ```
  SELECT * FROM dt WHERE c1=?;
  SELECT * FROM dt WHERE c2=? AND c3=?;
  ```

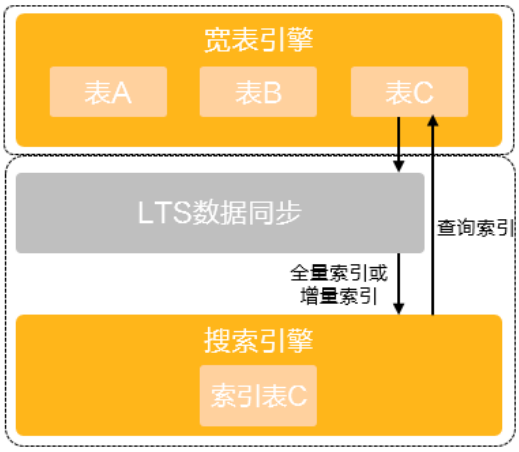
- 分词查询。c3列为分词字段，可以通过等值查询相关性高的结果集。例如查询c3列中包含阿里巴巴、阿里或者巴巴的数据。

  ```
  SELECT * FROM dt WHERE c3='阿里巴巴';
  ```

- 聚合分析。支持COUNT、SUM、MIN、MAX、AVG等聚合函数。
- 排序分页。支持任意索引列的排序（ORDER BY语句）。

架构介绍

SearchIndex是宽表引擎和搜索引擎深度融合后提供的新型索引。整体数据流如下图。在此架构下，宽表引擎、LTS和搜索引擎都是以独立服务的方式存在，您可以分别对各个引擎进行管理。如果搜索引擎处理能力不足，只需要扩容搜索引擎。如果LTS同步能力不足，可以单独扩容LTS。宽表引擎、LTS和搜索引擎可以针对不同的使用场景选择不同的机型，独立的部署形态大幅提升了系统的稳定性。



数据写入的流程如下：

1. 数据写入至Lindorm宽表引擎，原始数据记录在宽表WAL中，写入结果返回客户端。
2. 数据同步服务LTS实时监听宽表WAL，筛选出宽表WAL中带有SearchIndex的表，并将筛选的数据写入至搜索引擎。
3. 搜索引擎收到数据后实时构建倒排索引。

数据查询的流程如下：

1. 查询操作发送至Lindorm宽表引擎，计算层自动进行查询编译，基于优化器选择合适的SearchIndex。
2. 查询重定向至搜索引擎，查询满足条件的数据。
3. 汇总搜索引擎的数据，如果需要会自动回查宽表中的数据进行补齐，最后查询结果返回客户端。

应用场景

在宽表引擎和搜索索引配合下，不但可以提供高并发低延迟的KV查询，而且能够提供丰富的多维查询、分词查询、聚合分析等。如果您的业务有以下两个场景的需求，建议使用搜索索引。

- 架构改造
  如果您当前的业务架构中使用了HBase、MySQL或者MongoDB，同时也在使用搜索引擎Elasticsearch或者Solr。那么可以通过SearchIndex的一体化解决方案替换多个系统组合的架构。
- 业务选型
  - 金融行业，账单场景，详情请参见收钱吧订单查询&搜索。
  - 物流行业，订单场景，详情请参见申通快递迁移Oracle到Lindorm。

常见问题

- Q：搜索索引与二级索引有什么区别？
  A：二级索引是Lindorm宽表内置的特性，不需要开通即可使用，查询场景适合较为固定的业务需求，默认二级索引个数最多5个。搜索索引是宽表引擎与搜索引擎深度融合的特性，需要单独开通购买，核心功能为倒排索引和列存，适合较为复杂的多维查询，索引列个数最多1000个。
- Q：数据写入后大概多久可以通过搜索索引查询？
  A：搜索索引的数据一致性默认为最终一致，成功写入至宽表的数据，索引数据可以在1~15秒内可见。如果您将搜索索引的数据一致性设置为强一致模型，成功写入至宽表的数据，索引数据实时可见。

7.4.5.2. 开通搜索索引

云原生多模数据库Lindorm的搜索索引（SearchIndex）主要适用于复杂的多维查询场景。本文介绍开通搜索索引的方法。

注意事项

- 开通Lindorm实例的搜索索引服务过程中可能会造成业务访问抖动，建议您在业务低谷操作。
- 在开通搜索索引服务之前，您需要仔细评估实例的资源容量。为了帮助您进行资源规划，给出一些通用的建议参考，具体请

参见集群容量规划。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**宽表引擎 > 搜索索引（SearchIndex）**。
5. 单击**立即开通**。
6. 在**变配**页面，配置以下参数。



参数	说明
搜索节点规格(Solr API)	建议选择较高的搜索节点规格，例如选择16核64 GB，表示该规格能够承载的查询QPS通常在500以上，写入TPS通常在50000以上。 **说明** 如果您在购买Lindorm实例时已经购买了搜索引擎，则无需再选择此参数。
搜索节点数量	建议至少选择2个节点，避免出现单节点故障的问题。搜索节点数量的范围为0~60。 **说明** 如果您在购买Lindorm实例时已经购买了搜索引擎，则无需再选择此参数。
BDS数据同步Core规格	首次购买阶段，建议选择4核16 GB，后续可以根据业务需求升配或者扩容LTS数据同步服务。
BDS节点数量	建议选择2个LTS节点数量，LTS节点数量的范围为0~50。
存储空间	选择搜索引擎服务的存储空间大小。 **说明** 如果您在购买Lindorm实例时已经购买了搜索引擎，则无需再选择此参数。

在**变配**页面右下角显示的**配置费用**表示当前实例的购买总价格，包括宽表引擎、LTS同步服务、搜索引擎费用，详情请参见产品价格。

7. 阅读并勾选服务协议。
8. 单击**立即购买**。

7.4.5.3. 使用手册

7.4.5.3.1. 管理搜索索引

本文介绍管理搜索索引的操作方法。

前提条件

- 已开通云原生多模数据库Lindorm实例的搜索索引服务，具体操作请参见开通搜索索引。
- 已通过Lindorm-cli连接Lindorm宽表引擎，具体操作请参见通过Lindorm-cli连接并使用宽表引擎。

数据准备

管理搜索索引前需要先创建目标表并写入测试数据。

1. 执行以下语句创建目标表，表名为 `search_table` 。

```
CREATE DATABASE searchindex_db;
USE searchindex_db;
CREATE TABLE IF NOT EXISTS search_table (user_id bigint, name varchar, age smallint, gender varchar, add
ress varchar, email varchar, city varchar, constraint primary key (user_id asc));
```

2. 执行以下语句在目标表中写入四条数据。

```
UPSERT INTO search_table (user_id,name,age,gender,address,email,city) VALUES (1, '张先生', 18, 'M', '北京
市朝阳区', 'a***@example.net', '北京');
UPSERT INTO search_table (user_id,name,age,gender,address,email,city) VALUES (6, '李先生', 32, 'M', '杭州
市余杭区', 'a***@example.net', '杭州');
UPSERT INTO search_table (user_id,name,age,gender,address,email,city) VALUES (20, '王先生', 28, 'M', '杭州
市滨江区', 'a***@example.net', '杭州');
UPSERT INTO search_table (user_id,name,age,gender,address,email,city) VALUES (28, '陈女士', 36, 'F', '深圳
市南山区', 'a***@example.net', '深圳');
```

3. 执行以下语句查询目标表中的数据。

```
SELECT * FROM search_table LIMIT 10;
```

查询结果如下：

```
+---------+--------+-----+--------+-------------+------------------+------+
| user_id |  name  | age | gender |   address   |      email       | city |
+---------+--------+-----+--------+-------------+------------------+------+
| 1       | 张先生  | 18  | M      | 北京市朝阳区  | a***@example.net | 北京  |
| 6       | 李先生  | 32  | M      | 杭州市余杭区  | a***@example.net | 杭州  |
| 20      | 王先生  | 28  | M      | 杭州市滨江区  | a***@example.net | 杭州  |
| 28      | 陈女士  | 36  | F      | 深圳市南山区  | a***@example.net | 深圳  |
+---------+--------+-----+--------+-------------+------------------+------+
```

管理搜索索引

以上述测试数据为例，创建的搜索索引需要满足以下需求：

- 对任意一列都可以进行快速检索。
- 对 `address` 字段进行分词查询。
- 对 `email` 字段进行模糊查询。

1. 创建搜索索引，关于创建搜索索引的语法详情请参见CREATE SEARCH INDEX。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on search_table (name,age,gender,address(type=text, analyzer=ik),e
mail,city);
```

> **说明**
> - 执行创建搜索索引语句后如果出现 `The Lindorm Search cluster address is null` 报错，请确保已开通Lindorm实例的搜索索引服务。
> - 创建搜索索引语句中， `address` 字段按照ik分词器进行分词。由于 `user_id` 是表的主键，默认带主键索引，所以没有将 `address` 字段设置为索引列。
> - 每张表只能创建一个搜索索引。
> - 创建搜索索引时系统默认会执行同步构建搜索索引，历史数据越多，执行过程所需的时间越长。如果您不希望执行同步构建搜索索引，可以在创建搜索索引语句后加 `ASYNC` ，创建完成后修改搜索索引属性为 `USABLE` ，修改方法请参见修改搜索索引属性。

2. 执行以下语句查询指定表的搜索索引信息。

```
SHOW SEARCH INDEX FROM search_table;
```

查询结果如下：

```
+--------------+--------------+------------+-------------+------------+---------------+-----------------
------------------+-----------+
| TABLE_SCHEMA |  DATA_TABLE  | INDEX_NAME | INDEX_STATE | INDEX_TYPE | INDEX_COVERED |            INDEX
_COLUMN            | INDEX_TTL |
+--------------+--------------+------------+-------------+------------+---------------+-----------------
------------------+-----------+
| test         | search_table | idx        | BUILDING    | SEARCH     | NA            | address,city,age
,gender,name,email | 0         |
+--------------+--------------+------------+-------------+------------+---------------+-----------------
------------------+-----------+
```

3. （可选）构建搜索索引。如果采用 `ASYNC` 异步构建搜索索引，默认不会对历史数据构建搜索索引。如果需要对历史数据进行检索，请执行以下命令构建搜索索引，当搜索索引构建结束后索引状态会设置为 `ACTIVE` ，可以通过 `SHOW SEARCH INDEX FROM search_table;` 命令查看。

```
ALTER SEARCH INDEX IF EXISTS idx ON search_table REBUILD;
```

> **说明**
> 关于构建搜索索引的语法详情请参见ALTER SEARCH INDEX。

4. 查询数据，更多查询场景的方法请参见通过搜索索引查询宽表数据。

```
SELECT * FROM search_table WHERE name='王先生' AND age > 18 AND city='杭州';
```

查询结果如下：

```
+---------+--------+-----+--------+--------------+------------------+------+
| user_id |  name  | age | gender |   address    |      email       | city |
+---------+--------+-----+--------+--------------+------------------+------+
| 20      | 王先生 | 28  | M      | 杭州市滨江区 | a***@example.net | 杭州 |
+---------+--------+-----+--------+--------------+------------------+------+
```

5. 如果您不需要搜索索引，可以执行以下语句删除搜索索引，关于删除搜索索引的语法详情请参见DROP SEARCH INDEX。

```
ALTER SEARCH INDEX IF EXISTS idx ON search_table DISABLED;
DROP SEARCH INDEX IF EXISTS idx ON search_table;
```

使用动态列

1. 开启动态列功能，具体请参见开启动态列。

2. 创建search index时，指定动态列c2和c3。

```
CREATE SEARCH INDEX IF NOT EXISTS idx ON search_table(p1,c1,c2,c3) with(indexState=ACTIVE);
```

3. 指定动态列，创建search index后，可执行以下命令，查看动态列是否添加成功。

```
SHOW SEARCH INDEX FROM search_table;
```

```
+--------------+--------------+------------+-------------+------------+---------------+--------------+-----------+
| TABLE_SCHEMA |  DATA_TABLE  | INDEX_NAME | INDEX_STATE | INDEX_TYPE | INDEX_COVERED | INDEX_COLUMN | INDEX_TTL |
+--------------+--------------+------------+-------------+------------+---------------+--------------+-----------+
| test         | search_table | idx        | ACTIVE      | SEARCH     | NA            | p1,c3,c2,c1  |         0 |
+--------------+--------------+------------+-------------+------------+---------------+--------------+-----------+
```

4. 已创建search index，如需要增加动态列，可以通过以下命令添加。

```
ALTER SEARCH INDEX idx ON search_table  ADD COLUMNS(c4);
```

结果如下：

```
lindorm:test> alter search index idx on search_table  add columns(c4);
0 row affected (2186 ms)
lindorm:test> show search index from search_table;
+--------------+--------------+------------+-------------+------------+---------------+-----------------+-----------+
| TABLE_SCHEMA |  DATA_TABLE  | INDEX_NAME | INDEX_STATE | INDEX_TYPE | INDEX_COVERED |  INDEX_COLUMN   | INDEX_TTL |
+--------------+--------------+------------+-------------+------------+---------------+-----------------+-----------+
| test         | search_table | idx        | ACTIVE      | SEARCH     | NA            | c4,p1,c3,c2,c1  |         0 |
+--------------+--------------+------------+-------------+------------+---------------+-----------------+-----------+
```

5. 写入动态列和查询动态列，具体请参见管理动态列。

7.4.5.3.2. 分区索引

分区索引是为了解决大宽表的存储和高并发访问问题而设计的一种新特性。创建搜索索引时可以指定数据分区策略，服务端自动将数据进行拆分并存储，查询数据时系统自动进行分区裁剪。本文介绍数据分区的策略和使用方法。

前提条件

- 已开通云原生多模数据库Lindorm实例的搜索索引服务，具体操作请参见开通搜索索引。
- 已通过Lindorm-cli连接Lindorm宽表引擎，具体操作请参见通过Lindorm-cli连接并使用宽表引擎。
- 已准备测试数据，详情请参见数据准备。

使用场景

- 业务数据具有时间属性，例如车联网数据、订单详情、消息日志等。
- 业务数据有明显的聚类特性，例如商家数据表以商家ID为聚类，查询条件中包括商家ID。IoT设备数据表以设备ID为聚类，查询条件中包括设备ID。

数据分区策略

- HASH分区
- 时间范围分区
- HASH分区和时间范围分区组合

HASH分区

HASH分区将数据进行散列存储，从而避免出现数据的热点问题。在数据写入量较大的场景中可以很好的实现数据均衡。搜索索引默认按照Lindorm宽表的主键进行HASH分区，同时也支持自定义分区键。

HASH分区的语法示例如下：

- 创建搜索索引，默认按照Lindorm宽表的主键进行HASH分区，分区数量为16。

  ```
  CREATE SEARCH INDEX idx ON search_table (storeId, goodsId, goodsPrice) partitions 16;
  ```

- 创建搜索索引，按照Lindorm宽表的storeId列进行HASH分区（也就是一级HASH分区），分区数量为16。

```
CREATE SEARCH INDEX idx ON search_table (storeId, goodsId, goodsPrice) partition by hash(storeId) partitio
ns 16;
```

> **说明** 对Lindorm宽表的某一列进行HASH分区（即一级HASH分区）时，如果自定义的分区键（也就是Lindorm宽表的某一列）存在热点问题，那么可能导致大量数据写入同一个分区中，影响查询和写入性能。建议考虑对Lindorm宽表的多个列组合进行HASH分区（即二级HASH分区或者三级HASH分区）。

- 按照Lindorm宽表的storeId列和goodsId列组合进行HASH分区（也就是二级HASH分区），加盐因子为2，分区数量为16。

```
CREATE SEARCH INDEX idx ON search_table (storeId, goodsId, goodsPrice) partition by hash(storeId(salt_fact
or=2),goodsId) partitions 16;
```

> **注意**
> - 加盐因子（salt_factor）是进一步散列相同storeId列值的数据，通常设置为一个较小的数值，并且数值与分区数量是两倍的数量关系。如果分区数量为16，当salt_factor大于4时，则无法散列数据。加盐因子的其他值表示如下：
>   - salt_factor=1：表示相同storeId列的数据，不同goodsId列的数据会被散列到分区总数量的一半中。
>   - salt_factor=2：表示相同storeId列的数据，不同goodsId列的数据会被散列到分区总数量的1/4中。
>   - salt_factor=3：表示相同storeId列的数据，不同goodsId列的数据会被散列到分区总数量的1/8中。
> - 多级HASH分区（二级HASH分区或者三级HASH分区）除了可以散列数据，在查询数据场景中也大大提高了查询效率。例如在上述二级HASH分区中，查询数据时同时输入storeId列和goodsId列的过滤信息会确定在一个分区中进行数据检索，减少数据的扫描范围从而提高查询效率。
> - 使用二级HASH分区时，一级分区键（Lindorm宽表的第一列即storeId列）必须设置加盐因子。使用三级HASH分区时，一级分区键和二级分区键都必须设置加盐因子。
> - 自定义分区键的值有以下限制：
>   - 分区键值不可更改。
>   - 分区键值不能为空。

时间范围分区

对于一些时间序列的数据，可以按照时间范围分区，例如按照周或者月进行分区，同一时间范围内的数据将会聚集存储，并且可以自动淘汰旧分区的数据。

按照时间范围分区的语法示例如下：

- 创建索引，按照业务的时间列orderTime分区，从30天前开始，每7天自动分区，默认保留90天的分区数据。

```
CREATE SEARCH INDEX idx ON search_table (storeId, goodsId, goodsPrice, orderTime)
partition by range time(orderTime) partitions 16
with (indexState=ACTIVE, RANGE_TIME_PARTITION_START='30', RANGE_TIME_PARTITION_INTERVAL='7', RANGE_TIME_PA
RTITION_TTL='90', RANGE_TIME_PARTITION_MAX_OVERLAP='90');
```

- 创建索引，按照业务的时间列orderTime分区，从半年前开始，每1个月自动分区，默认保留半年的分区数据，分区字段单位设置为秒。

```
CREATE SEARCH INDEX idx ON search_table (storeId, goodsId, goodsPrice, orderTime)
partition by range time(orderTime) partitions 16
with (indexState=ACTIVE, RANGE_TIME_PARTITION_START='180', RANGE_TIME_PARTITION_INTERVAL='30', RANGE_TIME_
PARTITION_TTL='180', RANGE_TIME_PARTITION_MAX_OVERLAP='180', RANGE_TIME_PARTITION_FIELD_TIMEUNIT='s');
```

按照时间范围分区的参数说明如下表：

参数	说明
RANGE_TIME_PARTITION_START	表示创建索引操作前多少天开始创建分区。适用于有历史数据的场景，如果历史数据的时间戳比开始分区的时间还要小，则会报错。
RANGE_TIME_PARTITION_INTERVAL	表示间隔多少天创建新分区，例如 `RANGE_TIME_PARTITION_INTERVAL='7'` ，表示每隔一周创建一个新分区。

参数	说明
RANGE_TIME_PARTITION_TTL	表示保留多少天的分区数据，例如 `RANGE_TIME_PARTITION_TTL='180'` ，表示保留半年的分区数据，历史分区数据会被自动清理掉。
RANGE_TIME_PARTITION_MAX_OVERLAP	如果写入的数据时间点是将来的时间，这个参数表示最多允许与当前时刻的时间间隔，单位为天。
RANGE_TIME_PARTITION_FIELD_TIMEUNIT	表示业务指定的时间分区字段单位，默认单位为毫秒（ms）。 • 分区字段单位设置为秒（s），数字长度为10位。 • 分区字段单位设置为毫秒（ms），数字长度为13位。

HASH分区和时间范围分区组合

根据业务需求，如果需要将数据按照HASH分区和时间范围分区，可以将两个分区策略组合使用。HASH分区和时间范围分区组合的语法示例如下：

创建组合索引，先按照时间范围进行分区，再按照HASH分区。

```
CREATE SEARCH INDEX idx ON search_table (storeId, goodsId, goodsPrice, orderTime)
partition by range time(orderTime)
partition by hash(storeId) partitions 16
with (indexState=ACTIVE, RANGE_TIME_PARTITION_START='30', RANGE_TIME_PARTITION_INTERVAL='7', RANGE_TIME_PART
ITION_TTL='90', RANGE_TIME_PARTITION_MAX_OVERLAP='365');
```

7.4.5.3.3. 通过搜索索引查询宽表数据

本文介绍在多种查询场景下通过搜索索引查询Lindorm宽表数据，包括多维查询、排序翻页、分词查询、模糊查询和聚合分析。

前提条件

- 已开通云原生多模数据库Lindorm实例的搜索索引服务，具体操作请参见开通搜索索引。
- 已通过Lindorm-cli连接Lindorm宽表引擎，具体操作请参见通过Lindorm-cli连接并使用宽表引擎。
- 已准备测试数据并创建搜索索引，详情请参见管理搜索索引。

多维查询

执行以下语句实现多维查询。

```
SELECT * FROM search_table WHERE gender='M' AND city='杭州' OR city='北京';
```

查询结果如下：

```
+---------+--------+-----+--------+-------------+-----------------+------+
| user_id |  name  | age | gender |   address   |      email      | city |
+---------+--------+-----+--------+-------------+-----------------+------+
| 1       | 张先生 | 18  | M      | 北京市朝阳区 | a***@example.net | 北京 |
| 6       | 李先生 | 32  | M      | 杭州市余杭区 | a***@example.net | 杭州 |
| 20      | 王先生 | 28  | M      | 杭州市滨江区 | a***@example.net | 杭州 |
+---------+--------+-----+--------+-------------+-----------------+------+
```

```
SELECT * FROM search_table WHERE age > 30 AND city != '杭州';
```

查询结果如下：

```
+---------+--------+-----+--------+-------------+-----------------+------+
| user_id |  name  | age | gender |   address   |      email      | city |
+---------+--------+-----+--------+-------------+-----------------+------+
| 28      | 陈女士 | 36  | F      | 深圳市南山区 | a***@example.net | 深圳 |
+---------+--------+-----+--------+-------------+-----------------+------+
1 rows in set (348 ms)
```

排序翻页

- 搜索索引支持通过 `limit` 或者 `offset` 方式进行数据翻页，执行以下语句实现数据翻页查询。

```
SELECT * FROM search_table  WHERE gender='M' LIMIT 1,10;
```

查询结果如下：

user_id	name	age	gender	address	email	city
6	李先生	32	M	杭州市余杭区	a***@example.net	杭州
20	王先生	28	M	杭州市滨江区	a***@example.net	杭州

- 搜索索引支持任意列的排序，执行以下语句实现数据排序查询。

```
SELECT * FROM search_table  WHERE gender='M' ORDER BY age DESC;
```

查询结果如下：

user_id	name	age	gender	address	email	city
6	李先生	32	M	杭州市余杭区	a***@example.net	杭州
20	王先生	28	M	杭州市滨江区	a***@example.net	杭州
1	张先生	18	M	北京市朝阳区	a***@example.net	北京

分词查询

分词查询的列使用 `=` 查询，示例如下：`address` 列是分词字段，执行以下语句查询 `age` 的范围在 `(10,50]` 并且 `address` 在余杭区的数据。

```
SELECT * FROM search_table WHERE age > 10 AND age <= 50 AND address='余杭区';
```

查询结果如下：

user_id	name	age	gender	address	email	city
6	李先生	32	M	杭州市余杭区	a***@example.net	杭州

模糊查询

执行以下语句查询 `email` 列中包含gmail的数据。

```
SELECT * FROM search_table WHERE email LIKE '%example%';
```

查询结果如下：

user_id	name	age	gender	address	email	city
1	张先生	18	M	北京市朝阳区	a***@example.net	北京
6	李先生	32	M	杭州市余杭区	a***@example.net	杭州
20	王先生	28	M	杭州市滨江区	a***@example.net	杭州
28	陈女士	36	F	深圳市南山区	a***@example.net	深圳

> **说明** 模糊查询的性能受限于表的数据量，推荐使用分词查询。

聚合分析

搜索索引支持常用的聚合函数，例如COUNT、SUM、AVG、MIN、MAX。执行以下语句查询 `city` 为杭州的数据总个数。

```
SELECT COUNT(*) FROM search_table WHERE city='杭州';
```

查询结果如下：

COUNT(*)
2

1 rows in set (35 ms)

> 说明 聚合查询时，如果未指定WHERE条件，将无法命中搜索索引。此时可以在WHERE关键字前加上 `force index(search_index_name)` 强制走搜索索引，详情可参见SELECT语法。

相关文档

- 关于查询语法请参见SELECT。
- 通过SQL连接并使用搜索引擎请参见通过SQL连接并使用搜索引擎。

7.4.5.3.4. 高级查询语法

为了满足更多的查询检索需求，SELECT查询语句中支持 `search_query` 关键字，使用与Lucene（全文检索引擎工具包）查询语法相似。本文介绍通过 `search_query` 关键字查询数据的使用示例。

前提条件

- 已开通云原生多模数据库Lindorm实例的搜索索引服务，具体操作请参见开通搜索索引。
- 已通过Lindorm-cli连接Lindorm宽表引擎，具体操作请参见通过Lindorm-cli连接并使用宽表引擎。
- Lindorm宽表引擎小版本为2.2.17及以上，关于升级小版本的操作请参见升级小版本。
- 已准备测试数据并创建搜索索引，详情请参见管理搜索索引。

语法

```
SELECT selectors
  FROM table
  WHERE (search_query = 'search_expression')
  [ LIMIT n ]
  [ ORDER BY column_name ] ;
```

示例

- `OR` 逻辑查询

  ```
  SELECT * FROM search_table WHERE search_query = 'name:张先生 age:[10 TO 40]';
  ```

  与以下语句功能相同。

  ```
  SELECT * FROM search_table WHERE name='张先生' OR (age >= 10 AND age <= 40);
  ```

  查询结果如下：

  ```
  +---------+--------+-----+--------+-----------+-----------------+------+
  | user_id |  name  | age | gender |  address  |      email      | city |
  +---------+--------+-----+--------+-----------+-----------------+------+
  | 1       | 张先生 | 18  | M      | 北京市朝阳区 | a***@example.net | 北京 |
  | 6       | 李先生 | 32  | M      | 杭州市余杭区 | a***@example.net | 杭州 |
  | 20      | 王先生 | 28  | M      | 杭州市滨江区 | a***@example.net | 杭州 |
  | 28      | 陈女士 | 36  | F      | 深圳市南山区 | a***@example.net | 深圳 |
  +---------+--------+-----+--------+-----------+-----------------+------+
  ```

- `AND` 逻辑查询

  ```
  SELECT * FROM search_table WHERE search_query = '+name:张先生 +age:[10 TO 40]';
  ```

  与以下语句功能相同。

  ```
  SELECT * FROM search_table WHERE name='张先生' AND (age >= 10 AND age <= 40);
  ```

  查询结果如下：

  ```
  +---------+--------+-----+--------+-----------+-----------------+------+
  | user_id |  name  | age | gender |  address  |      email      | city |
  +---------+--------+-----+--------+-----------+-----------------+------+
  | 1       | 张先生 | 18  | M      | 北京市朝阳区 | a***@example.net | 北京 |
  +---------+--------+-----+--------+-----------+-----------------+------+
  ```

- 聚合统计

  ```
  SELECT count(*) FROM search_table WHERE search_query = 'age:[10 TO 40]';
  ```

查询结果如下：

```
+----------+
| COUNT(*) |
+----------+
| 4        |
+----------+
1 rows in set (38 ms)
```

- 分词查询

```
SELECT * FROM search_table WHERE search_query='address:杭州';
```

查询结果如下：

```
+---------+--------+-----+--------+--------------+-----------------+------+
| user_id | name   | age | gender | address      | email           | city |
+---------+--------+-----+--------+--------------+-----------------+------+
| 6       | 李先生 | 32  | M      | 杭州市余杭区 | a***@example.net | 杭州 |
| 20      | 王先生 | 28  | M      | 杭州市滨江区 | a***@example.net | 杭州 |
+---------+--------+-----+--------+--------------+-----------------+------+
```

- 结果排序，使用 `search_query` 关键字排序的返回结果默认按照Lucene打分排序，如果需要按照指定的 `age` 字段进行排序，请执行以下语句。

```
SELECT * FROM search_table WHERE search_query='{"q":"address:杭州", "sort":"age asc"}';
```

> **说明** 上述语句中 `q` 和 `sort` 为系统关键字，不需要修改。

查询结果如下：

```
+---------+--------+-----+--------+--------------+-----------------+------+
| user_id | name   | age | gender | address      | email           | city |
+---------+--------+-----+--------+--------------+-----------------+------+
| 20      | 王先生 | 28  | M      | 杭州市滨江区 | a***@example.net | 杭州 |
| 6       | 李先生 | 32  | M      | 杭州市余杭区 | a***@example.net | 杭州 |
+---------+--------+-----+--------+--------------+-----------------+------+
```

7.4.5.4. 使用限制

云原生多模数据库Lindorm的搜索索引（SearchIndex）服务在使用过程中存在一些限制，本文介绍搜索索引的使用限制。
搜索索引（SearchIndex）服务在使用过程中存在以下限制：

- 数据类型
  索引列支持的数据类型包括：TINYINT、SMALLINT、INTEGER、INT、BIGINT、LONG、FLOAT、DOUBLE、VARCHAR、BINARY、VARBINARY、BOOLEAN。不支持的数据类型为DECIMAL。
- 生存时间（Time-To-Live，简称TTL）
  搜索索引的TTL默认与主表保持一致，索引会定期清理历史数据，无法保证TTL的实时生效。您也可以设置搜索索引的TTL，具体操作请参见ALTER SEARCH INDEX。
- 数据多版本
  搜索索引不支持索引列数据的多版本。
- 自定义时间戳
  默认情况下，Lindorm服务端会为每行数据生成一个毫秒时间戳，相当于数据版本。如果在写数据时自定义时间戳，那么服务端会以应用端的时间戳为准。不同客户端自定义时间戳的方法如下：
  - HBase客户端按照下面的接口实现自定义时间戳。

    ```
    public Put(byte[] row, long ts)
    public Put addColumn(byte[] family, byte[] qualifier, long ts, byte[] value)
    ```

  - Lindorm SQL客户端按照下面的接口实现自定义时间戳。

    ```
    UPSERT INTO testTable (pk,c1,c2,_l_ts_) values('1','2','3',1640966400000);
    ```

  如果Lindorm宽表创建了搜索索引，在自定义时间戳写入的场景中，需要提前修改表属性。修改语句如下：
  - 通过HBase Shell修改表属性。

    ```
    ALTER 'testTable', MUTABILITY=> 'MUTABLE_ALL'
    ```

- 通过Lindorm SQL修改表属性。

```
ALTER TABLE testTable SET 'MUTABILITY' = 'MUTABLE_ALL';
```

> **说明** MUTABILITY属性的默认值为 `MUTABLE_LATEST` ，有关Mutability分类的说明请参见基本概念。

如果您在使用过程中遇到任何问题，请提交工单处理，工单链接为提交工单。

7.4.5.5. 开源客户端访问（兼容云数据库HBase增强版）

7.4.5.5.1. 通过开源客户端访问并使用搜索索引

云原生多模数据库Lindorm宽表引擎推出一种新的索引类型，称为搜索索引（SearchIndex）。本文介绍使用开源HBase API和Solr API访问并使用搜索索引。

背景信息

搜索索引通过融合Lindorm宽表引擎（兼容开源HBase API）和Lindorm搜索引擎（兼容开源Solr API）对外提供统一易用的访问接口，可以满足丰富的查询需求。标准使用方式是通过Lindorm SQL，为了方便更多的用户，也可以通过开源客户端使用搜索索引，有关搜索索引的具体介绍请参见特性介绍。

> **说明** 如果您是通过Lindorm SQL创建的表，请通过SQL API使用搜索索引。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已将客户端IP添加至Lindorm白名单中，具体操作请参见设置白名单。

操作步骤

1. 进入HBase Shell访问环境，具体操作请参见通过HBaseue Shell连接并使用宽表引擎。
2. 在HBase Shell创建宽表，表名为 `testTable` 。

```
create 'testTable', {NAME => 'f'}
```

3. 创建索引表，索引表的表名为 `democollection` 。您可以通过以下任一方式创建索引表。
   - 通过搜索引擎的集群管理系统创建索引表，具体操作请参见创建索引表。
   - 通过Search Shell创建索引表，具体操作请参见通过Search Shell连接并使用搜索引擎。
4. 映射宽表列和索引列。例如将表 `testTable` 中的 `f:name` （列族为f，列名为name）这一列映射到索引表 `democollection` 的 `name_s` 这一列。
   i. 在HBase Shell解压文件的*bin*目录中创建JSON文件（文件名为*schema*），将以下代码复制至JSON文件中。

```
{
  "sourceNamespace": "default",
  "sourceTable": "testTable",
  "targetIndexName": "democollection",
  "indexType": "SOLR",
  "rowkeyFormatterType": "STRING",
  "fields": [
    {
      "source": "f:name",
      "targetField": "name_s",
      "type": "STRING"
    }
  ]
}
```

> **说明** JSON文件中的参数说明请参见列映射的配置项说明。

ii. 进入HBase Shell环境，执行以下命令完成宽表列和索引表列的映射关系。

```
alter_external_index 'testTable', 'schema.json'
```

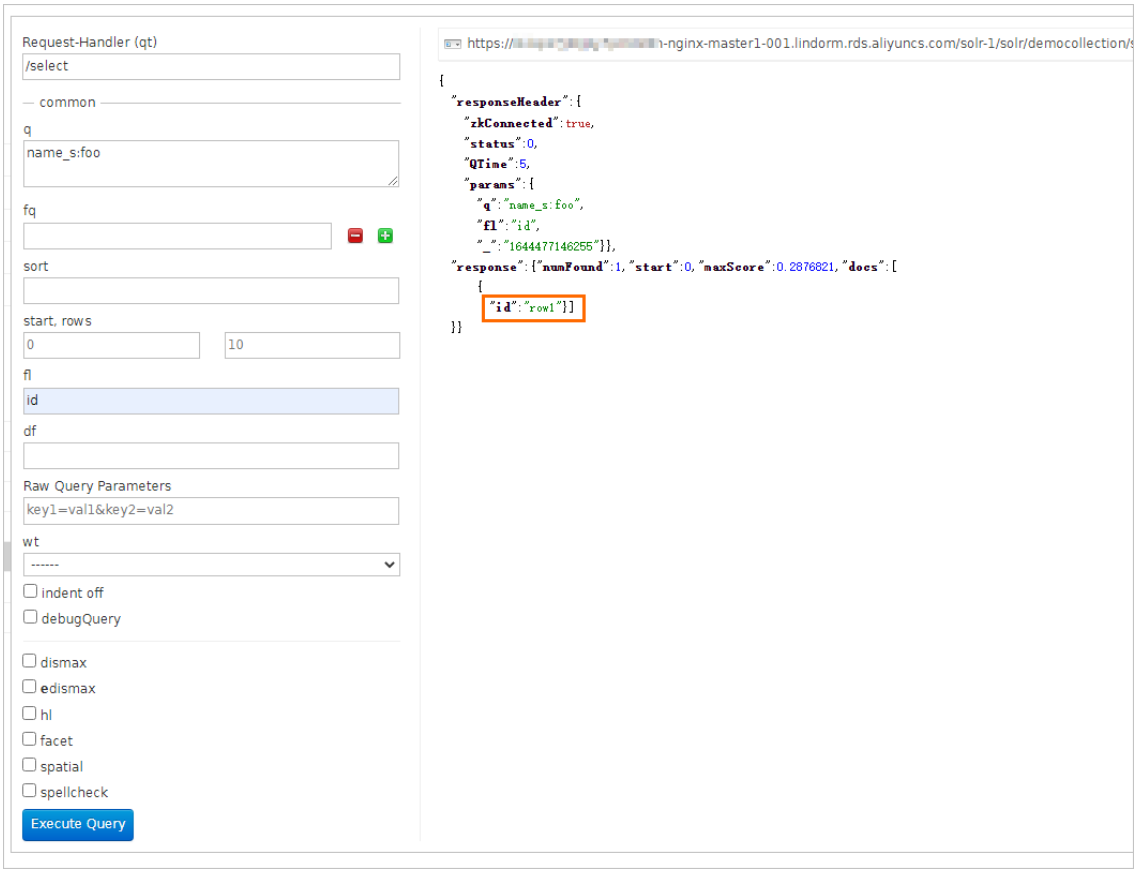
> **说明** 关于宽表列和索引表列的映射管理操作请参见管理列映射。

5. 在表 `testTable` 中写入一条数据。宽表和索引表的列映射关系配置完成后，实时写入宽表中的数据将自动同步到索引表中。对于宽表中的历史数据，需要手动执行全量构建索引才可以完成数据同步，具体操作请参见通过构建索引完成全量数据同步。

```
put 'testTable', 'row1', 'f:name', 'foo'
```

6. 查询索引数据。

i. 通过搜索引擎的集群管理系统查询索引数据，获取宽表主键 `row1` ，具体操作请参见查询索引数据。



ii. 进入HBase Shell环境，执行以下命令查询宽表数据。

```
get 'testTable','row1'
```

返回内容如下：

```
COLUMN                                        CELL
 f:name                                       timestamp=1644462597661, value=foo
1 row(s)
Took 0.0942 seconds
```

7. 通过LTS页面查看实时数据的同步任务状态。宽表和索引表的列映射关系配置完成后，实时写入宽表中的数据将自动同步到索引表中。

i. 登录LTS操作页面。在Lindorm控制台上单击目标实例ID，左侧导航栏选择**宽表引擎 > 搜索索引(SearchIndex)**，在**LTS WebUI 访问**区域中单击**ClusterManager**。

ii. 左侧导航栏选择**Lindorm Search > 实时同步**。



> **说明** 实时数据同步的延迟指标可以通过云监控控制台设置，最大任务延迟的报警阈值可以设置为600000毫秒，具体操作请参见云产品监控。

7.4.5.5.2. 列映射的配置项说明

通过开源客户端使用搜索索引需要将宽表的数据同步至索引表，同步的方法是建立宽表和索引表的列映射，本文介绍列映射的配置项说明。

配置示例

宽表和索引表的列映射采用JSON文件实现，列映射的配置示例如下，将宽表`testTable`的数据同步至索引表`democollection`，宽表的`f:name`这一列映射到索引表的`f_name_s`这一列，宽表的`f:age`这一列映射到索引表的`f_age_i`这一列。

```
{
  "sourceNamespace": "default",
  "sourceTable": "testTable",
  "targetIndexName": "democollection",
  "indexType": "SOLR",
  "rowkeyFormatterType": "STRING",
  "fields": [
    {
      "source": "f:name",
      "targetField": "name_s",
      "type": "STRING"
    },
     {
      "source": "f:age",
      "targetField": "age_i",
      "type": "INT"
    }
  ]
}
```

配置项说明

宽表和索引表的列映射配置参数说明如下：

参数	说明
sourceNamespace	宽表的Namespace名，如果宽表没有Namespace，默认Namespace名为default。
sourceTable	必选项，宽表的表名。
targetIndexName	必选项，Lindorm搜索引擎的索引表名。
indexType	默认配置为SOLR。

参数	说明
rowkeyFormatterType	表示宽表中每行主键映射到索引表主键ID（数据类型为String）的方式，支持以下两种： • STRING：适用于宽表的主键为String类型，例如主键为row1，order0001。该方式使用Bytes.toString(byte[])函数将宽表的主键转成索引表主键ID。用户在索引中基于条件查出结果数据的主键ID后，可以使用Bytes.toBytes(String)函数将主键ID转成宽表主键。 • HEX：适用于宽表的主键不是String类型，例如主键为12345。该方法使用org.apache.commons.codec.binary.Hex包中的encodeAsString(byte[])函数将宽表的主键转成索引表主键ID。用户在索引中基于条件查出结果数据的主键ID后，可以使用Hex.decodeHex(String.toCharArray())函数将主键ID转成宽表主键。

fields	指定映射的列和类型，由以下三个参数组成。如果有多个列的配置使用半角逗号（,）分隔。
	• source：宽表中需要映射的列名，列族名和列名使用半角冒号（:）分隔。
	• targetField：映射到索引表中的列名，推荐使用搜索引擎提供的动态列功能，搜索引擎通过动态列的后缀自动识别这一列的数据类型，例如：`name_s`表示这一列在索引表中的数据类型为`STRING`。有关动态列的格式请参见targetField参数说明。
	• type：宽表中需要映射的列的数据类型，配置时区分大小写，包括以下数据类型：
	  ◦ INT
	  ◦ LONG
	  ◦ STRING
	  ◦ BOOLEAN
	  ◦ FLOAT
	  ◦ DOUBLE
	说明 确保宽表的数据类型与配置的数据类型一致，否则会影响到索引数据的生成。在宽表引擎中需要您通过调用`Bytes.toBytes(string/long/int/...)`方法将对应的数据类型转化为Bytes存储至宽表中。

targetField参数说明

targetField参数表示宽表的列映射到索引表中的列。推荐使用搜索引擎提供的动态列功能，搜索引擎通过动态列的后缀自动识别这一列的数据类型，格式为`列名:列的数据类型`，列名以name为例，动态列支持的数据类型如下表：

数据类型	targetField示例值
INT	name_i
LONG	name_l
STRING	name_s
STRING（映射为中文分词的数据类型）	name_ik
BOOLEAN	name_b
FLOAT	name_f
DOUBLE	name_d

7.4.5.5.3. 管理列映射

通过开源客户端使用搜索索引需要将宽表的数据同步至索引表，同步的方法是建立宽表和索引表的列映射，列映射配置完成后，需要对列映射进行管理操作，包括创建列映射、查看列映射、增加列映射、删除列映射。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已通过开源客户端访问搜索索引并完成宽表和索引表的列映射配置文件，具体操作请参见通过开源客户端访问并使用搜索索

引。

创建列映射

执行以下命令创建宽表和索引表的列映射：

```
alter_external_index 'testTable', 'schema.json'
```

> **说明** 宽表名称为 `testTable`，`schema.json` 为宽表和索引表的列映射配置文件。

修改列映射

如果您需要修改宽表和索引表的映射关系，可以修改列映射配置文件*schema.json*，然后重新创建列映射。

```
alter_external_index 'testTable', 'new_schema.json'
```

> **说明** `new_schema.json` 为修改后的列映射配置文件。

查看列映射

执行以下命令可以查看宽表和索引表的映射关系。

```
describe_external_index 'testTable'
```

增加列映射

如果您需要在列映射关系上添加一列或者几列的映射关系，可以执行以下命令。

```
 add_external_index_field 'testTable', {FAMILY => 'f', QUALIFIER => 'money', TARGETFIELD => 'money_f', TYPE
=> 'FLOAT' }
```

> **说明** 执行创建列映射命令（`alter_external_index`）后，才能执行上述命令增加列映射。如果需要修改的列较多，推荐采用 `alter_external_index` 命令一次性提执行。

删除列映射

如果您需要在列映射关系上删除一列或者几列的映射关系，可以执行以下命令。

```
remove_external_index 'testTable', 'f:name', 'f:age'
```

删除所有的列映射

1. 删除列映射配置文件*schema.json*中所有的列映射配置信息（fields参数中的数据）。

   ```
   {
     "sourceNamespace": "default",
     "sourceTable": "testTable",
     "targetIndexName": "democollection",
     "indexType": "SOLR",
     "rowkeyFormatterType": "STRING",
     "fields": []
   }
   ```

2. 执行创建列映射命令。

   ```
   alter_external_index 'testTable', 'del_schema.json'
   ```

   > **说明** `del_schema.json` 为删除后的列映射配置文件。

相关文档

如果您需要使用Java API管理列映射，先通过Java API连接Lindorm宽表引擎，具体操作请参见通过HBase Java API连接并使用宽表引擎。

与索引管理相关的API在 `AliHBaseUETable` 类中，示例如下：

```
AliHBaseUETable table = (AliHBaseUETable)connection.getTable(TableName.valueOf("testTable"))
void addExternalIndex(ExternalIndexConfig config, List<ExternalIndexField> fields) throws IOException;
void removeExternalIndex(List<String> fields) throws IOException;
```

7.4.5.5.4. 通过构建索引完成全量数据同步

宽表和索引表的列映射关系配置完成后，实时写入宽表中的数据将自动同步到索引表中。对于宽表中的全量数据，需要手动执行全量构建索引才可以完成数据同步。本文介绍通过构建索引完成全量数据同步。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已通过开源客户端访问搜索索引并完成宽表和索引表的列映射配置文件，具体操作请参见通过开源客户端访问并使用搜索索引。
- Lindorm实例的LTS引擎为3.6.10及以上版本。

查看全量数据同步任务

1. 进入Shell访问环境，具体操作请参见通过HBaseue Shell连接并使用宽表引擎。
2. 执行以下命令为宽表 `testTable` 的全量数据构建索引，命令是异步执行的。

   ```
   build_external_index 'testTable'
   ```

3. 查看全量数据同步任务的进度。全量数据的构建由LTS完成，可以在LTS页面中查看。
   i. 登录LTS操作页面。在Lindorm控制台上单击目标实例ID，左侧导航栏选择**宽表引擎 > 搜索索引(SearchIndex)**，在**LTS WebUI 访问**区域中单击**ClusterManager**。
   ii. 左侧导航栏选择**Lindorm Search > 全量同步**。

4. （可选）取消宽表的全量数据构建索引。全量数据构建过程中是异步执行的，如果您需要停止全量数据同步任务，可以执行以下操作。

   ```
   cancel_build_external_index 'testTable'
   ```

通过指定时间范围全量同步数据

> **说明** Lindorm实例的LTS引擎为3.6.10及以上版本。

1. 登录LTS操作页面。在Lindorm控制台上单击目标实例ID，左侧导航栏选择**宽表引擎 > 搜索索引(SearchIndex)**，在**LTS WebUI 访问**区域中单击**ClusterManager**。
2. 单击页面右上角**配置管理**。
3. 将浏览器地址中的conf.jsp修改为cm.jsp并回车，进入**配置信息修改**页面。



4. 输入**配置名**和**配置值**。
   - 在**配置名**文本框中输入*search.index.build.timerange*。
   - 在**配置值**文本框中配置索引表和时间范围，格式为：*索引表名:时间列,时间单位,开始的时间值,结束的时间值*。以*myIndex:create_time,ms,1654486200000,1654488000000*为例，表示对于索引表myIndex按照create_time时间列的值进行过滤，将时间范围为[1654486200000,1654488000000]内的数据重新构建索引。说明如下：

配置项	说明	示例
索引表名	输入索引表的名称。	myIndex
时间列	指定时间列。如果没有时间列，可以使用默认列`update_version_l`，单位为毫秒。	create_time
时间单位	时间列的时间单位，默认为毫秒。可配置为秒和毫秒。	ms
开始的时间值	重新构建索引开始时间的毫秒值。	1654486200000，表示时间为2022年6月6号11点30分。
结束的时间值	重新构建索引结束时间的毫秒值。	1654488000000，表示时间为2022年6月6号12点00分。

> **说明** 如果需要设置多个索引表的重新构建配置值，配置值之间使用半角分号（;）分隔。

5. 单击**修改**。
6. 执行以下命令为宽表`testTable`的全量数据构建索引。

```
build_external_index 'testTable'
```

7. 等待全量索引任务执行完成，将**配置名**为*search.index.build.timerange*，**配置值**修改为空并单击**修改**，如下图所示。

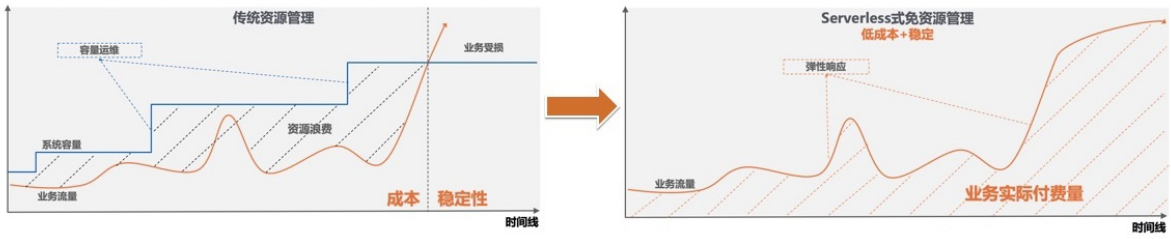


7.4.6. 宽表Serverless

本文介绍宽表引擎Serverless模式。

Serverless简介

在宽表Serverless模式下，用户无需规划硬件资源，也无需购买CPU和内存数。用户可以根据自身业务的请求量购买相应的CU（Capacity Unit，能力单元），计算方式请参见计费项，购买完即可开始使用，或者选择按量付费模式，即按照请求的多少付费，0请求0费用。Serverless为用户提供了弹性伸缩功能，当业务请求发生变化时，Serverless后台会调配合适的物理资源满足业务需求，用户可以获得瞬间的弹性能力，极大减少了运维成本和系统风险。
相比传统的模式，在Serverless模式下用户只需为使用的资源付费，给业务带来极大的成本节省。



Serverless适用场景

中小型轻量级的用户

Serverless版本非常合适中小客户，无需购买整个集群，按使用量收费。短暂使用成本可以压缩到原来的10%以下。是秒级别服务启动和销毁的速度也让客户的应用非常灵活。

定时处理任务的用户

对于定时处理业务的客户，Serverless天生的弹性能力非常符合业务需求。业务每天定时处理业务请求或者在夜间进行数据分析处理，可以使用Serverless的按量付费模式，使业务只为定时任务执行时的请求付钱，其他时间段无需付费。

业务波动大的用户

对于业务波动比较大的客户，如果提前准备大量的节点资源或者进行容量预估，就会费时费力费资金。在Serverless形态下，当大量请求来临前，可以通过调节购买的CU量，瞬间完成弹性能力。当业务高峰期过后，又可以将购买的CU调节回来，从而达到一个节省成本的目的。

免运维的业务

Lindorm宽表Serverless后台的运维团队是经过长期技术积累的阿里运维团队，可以非常专业的对集群进行运维，升级，调优和问题处理。用户无需再关心资源水位，软件版本等问题，可以做到完全0运维。

开通Serverless

前提条件

- 已注册阿里云账号。具体操作请参见注册阿里云账号。
- 已拥有阿里云专有网络（VPC）。如果没有VPC网络，请登录专有网络（VPC）控制台。
- 已登录Lindorm控制台。

创建Serverless模式的Lindorm实例

1. 单击**创建**按钮进入**Lindorm实例创建**页面。
2. 选择商品类型为**LindormServerless（包年包月）**。
3. 配置其他Lindorm参数。具体参数说明请参见创建实例。
4. 设置参数后，单击**立即购买**，进入**订单确认**界面。确认**参数配置**，勾选**服务协议**，单击**去支付**完成支付。

Serverless使用方式

宽表Serverless的使用方式与普通集群版完全一致，用户可以用HBase客户端和CQL客户端访问。使用方式参见SDK安装。数据导入请参见开通LTS。

Serverless计费方式

计费项

请求计费

Serverless读写计量单位以CU（Capacity Unit，能力单元）来计算，单行读操作返回数据大小大于4KB，则消耗读CU为实际数据大小按4KB整除向上取整。

1. 单行读操作返回数据大小大于4KB，则消耗读CU = 实际数据大小按4KB整除向上取整。
2. 对于Batch的多行读操作，消耗的读CU为所有单行读操作消耗的CU之和。每单行读操作消耗的读CU按规则1对该行进行计量。
3. 如果是Scan的批量读操作，则消耗读CU = 读取到的所有行主键的数据大小与每行实际读取的属性列数据大小之和除以4KB向上取整。这意味着如果Scan带Filter的操作在服务器上扫描了100行数据，返回了1行数据，计算CU使用的是这100行数据。
   举例说明：比如在某一秒内发生了10次单行读操作。
   其中5次读操作（每次操作读出当行数据大小为3.78KB），读出的数据大小为（3.78KB按4KB向上取整）*5。
   另外5次读操作（每次操作读出当行数据大小为4.26KB），读出的数据大小为（4.26KB按4KB向上取整）*5。
   这一秒内10次单行读操作所实际消耗的读CU值为（5 x 1）+（5 x 2）= 15。其中，3.78KB向上取整计为1个读CU，4.26KB向上取整计为2个读CU。

空间计费

Serverless空间计费使用的是写入Lindorm的每个原始KeyValue大小，而非压缩后的表大小。用户可以调用 `org.apache.hadoop.hbase.CellUtil.estimatedSerializedSizeOf()` 函数来估算每个KeyValue的大小。

由于Lindorm使用的是LSM（Log-structured merge-tree）存储结构。因此删除和TTL过期的数据不会立即删除，需要等待Compaction结束后才能删除，删除后空间才会减少。

付费模式

目前Serverless有两种付费方式，一种是包年包月，一种是按量付费。

- 包年包月模式：用户预购计算资源和存储资源。当使用的计算资源超过购买的CU量时，会遇到 `QuotaExceededException` 错误。当存储量超过购买量时，会导致Serverless实例写锁定。用户可以在实例界面随时调节购买的CU数和磁盘来应对业务变化。
- 在按量付费模式：用户无需提前购买计算资源和存储资源，完全按照用户使用的资源情况收费，0请求0费用

包年包月

在包年包月模式下，用户预购计算资源和存储资源。当使用的计算资源超过购买的CU量时，会遇到 `QuotaExceededException` 错误。当存储量超过购买量时，会导致Serverless实例写锁定。用户可以在实例界面随时调节购买的CU数和磁盘来应对业务变化。例如：

1. 购买10CU的计算资源代表每秒可以最多请求10个CU，使用超过会遇到限流。每秒没有使用完的CU**不会**累积到下一秒。
2. 购买了10GB的存储空间。当Serverless实例的空间使用超过10GB，实例会被写锁定。如果存储空间不足10GB，仍然按照每月10GB的购买量付钱。

按量付费

即将上线，敬请期待！

在按量付费模式下，用户无需提前购买计算资源和存储资源，完全按照用户使用的资源情况收费，0请求0费用。
目前按量付费模式下，每个实例限制20000 CU/s。如果超过此需求用户请提工单咨询。
按量付费会按照小时为单位发出账单。比如一位用户在一小时内，请求消耗了3w个CU，并且在这个小时内最高使用的存储空间为20GB，则这个小时的请求费用为30000 CU的价格。存储费用为20 GB的价格。如果下一小时该用户没有请求，存储量仍为20GB，则该用户下一小时只需要付存储费用，无计算费用。

如何选择付费模式

对于请求量比较固定的用户来说，包年包月的价格会比较划算，对于请求量波动比较大的用户来说，按量付费可以做到0请求0费用，在没有请求的时间段，用户可以无需为计算资源付费。

Serverless使用限制

- 最大请求大小不超过2 M。
- 最大scan请求大小不超过4 M。
- batch操作最大批量不超过100。
- 表名长度限制128B。
- namespace长度限制128B。
- cf长度限制32B。
- 建表初始化region个数限制64个。
- 单instance对应全局表数量限制64张。
- 单instance创建namespace限制5个。

7.4.7. 备份恢复

7.4.7.1. 开通备份恢复

云原生多模数据库Lindorm宽表引擎支持数据备份恢复功能，该功能是基于数据生态服务中的数据迁移将数据存储至阿里云对象存储服务OSS（Object Storage Service，简称OSS）中，定期全量备份数据，实时增量同步数据，来满足对数据备份和数据恢复的需求。

功能优势

- 实时增量同步，小于30s。数据恢复点目标
- 默认全库备份，支持对单个表或多个表进行备份恢复。

- 指定时间点恢复。

备份恢复空间评估

> **注意** 请合理评估备份恢复的使用空间，超过使用量后，会导致数据备份中断。

- 备份恢复存储空间大小是全量备份空间大小和增量备份空间大小之和。
- 全量备份空间大小 =（保留全备个数+1） x 单个全备大小。

> **说明** 保留全备个数+1的原因是只有在新备份数据完成后才会淘汰旧数据，在数据备份的过程中需要额外一份全备空间。

- 全量备份空间大小可以在Lindorm实例的集群管理页面获取。
- 增量备份空间大小 = 日志保留天数 x 每天增量LOG大小。例如实例写入的平均速度为1 MB/s，每天需要的增量备份空间大小为84 GB。

> **说明**
> - 实例写入速度可以通过监控获取或者请提交工单。
> - 对于写入量大的实例，LOG的日增量会很大，可以通过适当减小全备周期来减少LOG存储量。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏中选择**宽表引擎 > 备份恢复**。
5. 单击**立即开通**，跳转至**变配**页面。
6. 在**变配**页面，根据需求选择配置项。
   i. 选择**冷存储存储容量**：滑动滑块，将冷存储的存储容量调整为目标容量。
   ii. 选择**BDS数据同步Core规格**：选择LTS的Core规格，具体方法请参见Core规格选择。
   iii. 选择**BDS节点数量**：选择LTS的节点数量。
   iv. 阅读并勾选服务协议。
7. 配置完成后，单击**立即购买**。

7.4.7.2. 自动备份Lindorm数据

云原生多模数据库Lindorm宽表引擎提供的备份功能可以根据默认全量备份周期自动备份Lindorm宽表数据，您也可以根据业务需求设置全量备份周期。本文介绍自动备份Lindorm宽表数据的方法。

前提条件

已创建云原生多模数据库Lindorm实例，同时Lindorm实例已开通备份恢复功能，具体操作请参见开通备份恢复。

背景信息

云原生多模数据库Lindorm生成的备份文件存储在阿里云对象存储服务（Object Storage Service，简称 OSS）中，不会占用Lindorm实例的存储空间。关于对象存储服务的更多信息，请参见什么是对象存储OSS。

费用说明

按照备份存储使用量进行按量计费。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。

4. 在左侧导航栏中选择**宽表引擎 > 备份恢复**。
5. 在**全量备份**区域中，单击**创建备份设置**。
6. 在**创建备份设置**对话框中，根据业务需求设置备份参数。

配置	说明
备份表	备份全库数据或者Namespace中的表。 ○ 使用星号（*）备份全部的数据库数据。 ○ 备份表的格式为 `default:test` ，表示备份Namespace名称为default中的以test开头的表。
全量备份周期（天）	默认7天触发一次全量备份。可选时间范围是3~10天，时间设置太短可能在一个周期内无法完成备份，时间设置太长会影响数据恢复时间。
下次全量备份时间	下次全量备份开始的时间点。建议您将全量备份时间设置为业务低峰期的某个时间点。
全量备份保留个数	全量备份记录的保留个数，可选范围是1~12个。

7. 单击**确定**。

7.4.7.3. 恢复备份数据至当前实例

数据恢复功能可以最大程度地减少因数据库误操作引起的损失，本文介绍将备份数据恢复至当前实例的方法。

前提条件

- 已创建云原生多模数据库Lindorm实例，同时Lindorm实例已开通备份恢复功能，具体操作请参见开通备份恢复。
- 已完成备份数据操作，具体请参见自动备份Lindorm数据。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏中选择**宽表引擎 > 备份恢复**。
5. 在**全量备份**区域中，单击**发起数据恢复**。
6. 在**恢复**对话框中，根据业务需求设置数据恢复参数。

配置	说明
恢复集群	选择数据恢复的目标实例名称。 **说明** 数据库版本升级后，原版本Lindorm的备份数据不能用于恢复新版本的Lindorm数据。
时间点	选择数据恢复的时间点。
全库恢复	全库恢复选择**是**。
恢复表	当全库恢复选择**否**时，需要设置恢复表。一行写一个namespace，不支持使用星号（*）。 ○ 恢复表格，格式为 `namespace:table` ，例如 `default:testTable` 。 ○ 恢复到其他表，格式为 `namespace:table/namespace:table2` ，例如 `default:testTable/default:testTable2` 。

7. 单击**确定**。您可以在**恢复列表**页签中单击**操作**列中的**详情**查看数据恢复进度。

FAQ

- 故障时最近可以恢复到哪个时间点？
  正常不积压情况下，取决于日志WAL备份至OSS的周期，默认是30M或30s刷新一次，最多丢30s数据。
- 多长时间可以恢复？
  取决于数据量和带宽限制。
  - 全量数据恢复速度：OSS最大带宽1 GB/s，BDS单机100 MB/s。
  - 增量数据恢复速度：Lindorm目的集群单机30~40 MB/s写入，BDS单机100 MB/s。
- 支持单表恢复吗？
  支持，如果有同名表还可以恢复到另外的表中。

7.4.8. 开启透明数据加密（内测中）

云原生多模数据库Lindorm支持透明数据加密TDE（Transparent Data Encryption）功能。启用TDE功能后，Lindorm通过密钥层级结构对数据文件进行加密，所有存储在文件系统中的数据都以密文形式存在。为提高数据安全性，您可以通过控制台启用TDE功能，对实例数据进行加密。本文介绍启用TDE功能的具体步骤。

前提条件

- 云原生多模数据库Lindorm实例的部署方案为**单可用区**，具体操作请参见创建实例。
- 已开通密钥管理服务KMS（Key Management Service），具体操作请参见开通密钥管理服务。
- 已授权Lindorm访问KMS，需要通过服务关联角色获取访问权限，具体操作请参见Lindorm服务关联角色。

背景信息

云原生多模数据库Lindorm的TDE功能支持AES算法和SMS4算法。TDE加密使用的密钥由密钥管理服务（KMS）产生和管理，KMS为您提供一个主密钥，通过该密钥加密数据密钥，再通过数据密钥加密数据，这个流程叫做信封加密。在解密过程中先读取数据密钥密文，通过主密钥来解密数据密钥密文得到数据密钥明文，再通过数据密钥明文来解密数据。KMS提供的主密钥是通过API创建的，如果您提供了密码还可以对主密钥进行加密，保证主密钥的安全性。

注意事项

- 开通TDE的过程中，实例会重启一次并出现连接闪断，建议您在业务低峰期操作并确保应用有重连机制。
- TDE功能开启后将无法关闭。

操作步骤

1. 登录Lindorm控制台。
2. 单击目标实例ID，进入**实例列表**页面。
3. 选择左侧导航栏**宽表引擎 > 数据安全**，默认打开**数据加密TDE**页面。
4. 打开**当前状态**开关。
5. 在**开启TDE**对话框中，选择**使用自动生成密钥**或**使用自定义密钥**。



   - 如果您选择**使用自动生成密钥**，单击**确定**，开通TDE。
   - 如果您选择**使用自定义密钥**，从**设置密钥**列表中，选择由密钥管理服务（KMS）产生的密钥，单击**确定**，开通TDE。

> **说明** 如需使用自己传入的自定义密钥，请单击**管理密钥**，在密钥管理服务控制台创建密钥并导入自带的密钥材料，详情请参见创建密钥。

后续步骤

开启TDE后，您需要使用Lindorm-cli连接实例，具体操作请参见通过Lindorm-cli连接并使用宽表引擎，对Lindorm的表进行DDL操作才能使得数据加密，具体如下：

- 对于新建的表，建表时加上 `WITH (FILEVERSION = 5, ENCRYPTION = AES)` 语句。

```
CREATE table testsql (p1 varchar, p2 varchar, p3 bigint, c1 varchar, c2 double, constraint primary key (p1, p2, p3 desc)) WITH (FILEVERSION = 5, ENCRYPTION = AES);
```

- 对于现存的表，您需要使用 `ALTER` 关键字和 `WITH (FILEVERSION = 5, ENCRYPTION = SMS4)` 语句。

```
ALTER table testsql WITH (FILEVERSION = 5, ENCRYPTION = SMS4) ;
```

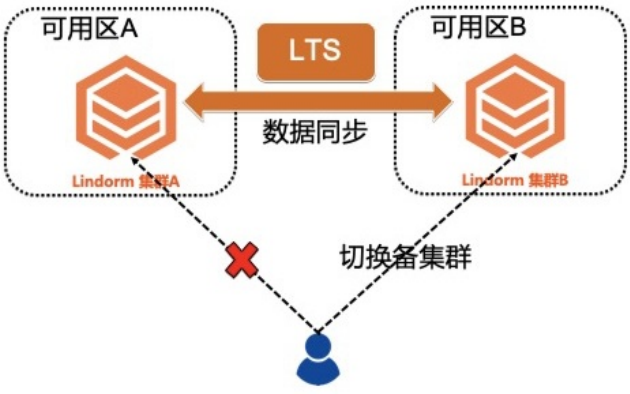
> **说明**
> - `FILEVERSION` 表示文件版本，文件版本设置为5才可以使用加密功能。
> - `ENCRYPTION` 表示加密算法类型，取值为AES或SMS4。
> - 您可以通过 `ENCRYPTION=NONE` 语句使得数据解密。
> - 您可以转换加密算法类型，例如由AES转换为SMS4，但是此方法不推荐，执行操作如下：
>   i. 通过 `ENCRYPTION=NONE` 语句使得表的数据解密。
>   ii. 对表进行 `major_compact` 操作并等待完成。
>   iii. 通过 `ENCRYPTION=SMS4` 语句设置数据加密类型为SMS4。

7.4.9. 多可用区部署

云原生多模数据库Lindorm支持创建多可用区的实例。该方案将一个Lindorm实例部署在多个可用区，多可用区实例具备更高的容灾能力，同时Lindorm实例可以实现多个可用区之间数据的强一致，也可以在数据最终一致下发出请求返回最快的结果，从而提高在线业务的服务质量。

传统的主备容灾概述

传统的主备容灾为了实现高可用性，通常的原理是分别在两个不同的可用区（可用区A和可用区B）中购买一个Lindorm实例（主实例1和备实例2），使用数据通道服务（简称LTS）实现Lindorm实例间的双向同步。当主实例1发生故障或者可用区A不可用时，用户将访问的连接切换至备实例2或者可用区B，从而实现高可用，主备容灾的高可用架构图如下所示。



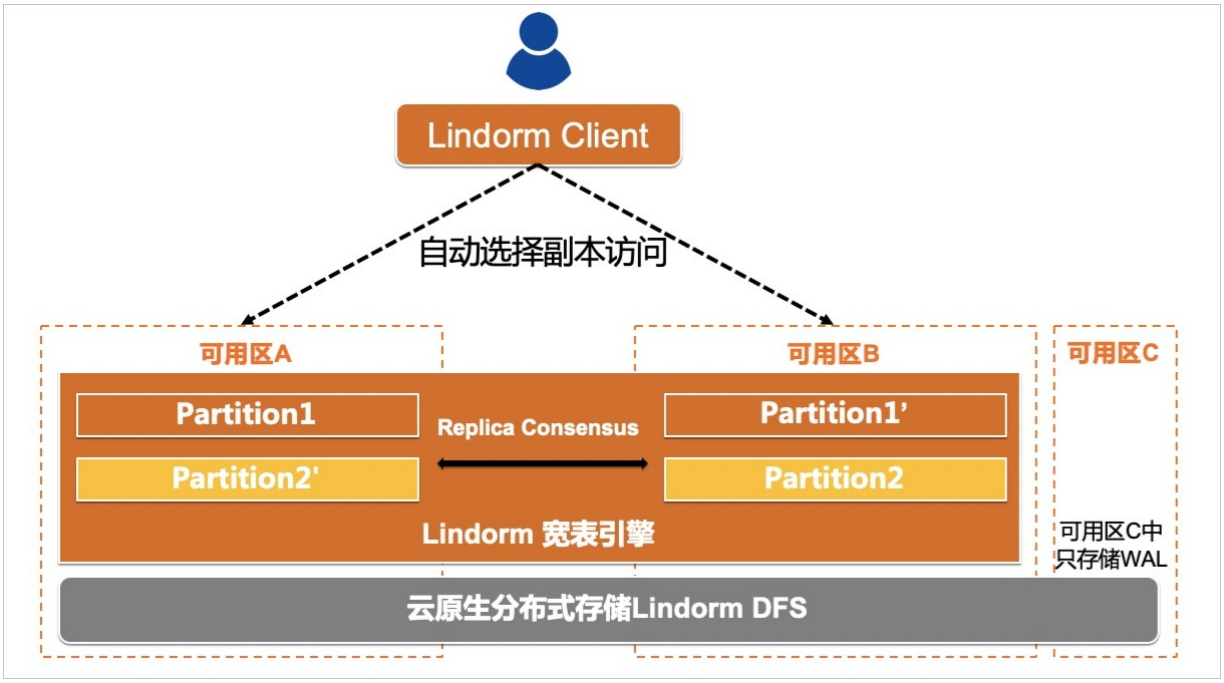
主备容灾的方案虽然能够满足大部分用户的高可用需求，但是这种主备容灾方案并不适用所有的业务，存在以下不足之处：

- 主备实例的数据同步存在延迟，无法满足强一致需求。
  主备实例的数据同步链路为异步链路，也就是当业务数据写入主实例1后，数据在备实例2不是立即可见的，存在一定的延迟。当发生主备切换操作时，业务并不能立刻读到最新的数据，这是一些业务无法接受的。如果业务使用了Increment、CheckAndPut等先读后写的原子语义接口，并且没有读到最新数据，那么就会导致数据错乱或者回退。如果可用区A的网络存在故障，由于同步延迟问题，在可用区A网络恢复之前的时间段内可用区B的数据会一直处于缺失的状态。

- 备实例资源利用率不高。
  在主备容灾下，大部分时间备实例的资源不会被使用，只有在主备切换操作的时候才会被访问。
- 主备切换需要数据库管理员实施。
  目前主备容灾完全由业务自行完成，业务的DBA需要时刻注意主实例的运行情况。在主实例发生故障时，DBA决定是否需要切换为备实例。并且切换的逻辑和方案都需要用户自行去定制，对业务不透明。

总的来说，传统的主备容灾存在以上问题，但云原生多模数据库Lindorm多可用区部署可以完全解决这些不足之处。

多可用区架构



云原生多模数据库Lindorm支持多可用区部署，Lindorm实例宽表的Partition会在每个可用区中存在一个独立的副本，同时日志WAL会存储至底层的可用区C云原生分布式存储Lindorm DFS中。如果可用区A完全不可用，可用区B通过可用区C的数据完成数据的恢复。副本间的数据同步由Lindorm内置的Replica Consensus协议完成，Replica Consensus协议只需要存储至少两个副本的数据就可以完成数据同步。在多可用区部署模式下，Lindorm支持设置表的一致性级别来实现不同的业务需求，包括强一致和最终一致。

- 强一致：如果将表设置为强一致，那么只有表的主Partition可以读写，备Partition只能通过Replica Consensus协议接收数据。当主可用区的地域所有机器都终止工作或者可用区不可用时，Lindorm会自动触发选择主可用区操作，备可用区需要一定的时间恢复才能切换为主可用区。强一致模式下，用户可以读取最新写入的数据。
- 最终一致：如果将表设置为最终一致（也称为弱一致），那么表的主备Partition都可以读写，Replica Consensus协议是将可用区A写入的数据同步至可用区B中，由于数据同步是异步的，所以两个副本之间的数据会存在不一致（通常在100ms量级）。最终一致模式下，由于表的主备Partition都可以读写，当可用区A不可用、表读写出现毛刺和机器终止工作等故障时，超过一定的时间不会返回访问结果，无需等待和系统切换主可用区的过程，Lindorm会自动选择可用区B发送请求，达到高可用和降低读写毛刺的效果。

功能特性

虽然云原生多模数据库Lindorm是分布式架构，本身具有高可用性，但是一些对可用性要求高的业务，会有更高的要求。业务不但需要Lindorm能应对部分机器故障的问题，还必须能够应对网络不可用以及城市灾难等极端问题。为了保障数据库在各种意外情况下都能持续提供服务，云原生多模数据库Lindorm多可用区部署支持以下功能特性：

- 具备机房级或者城市级容灾能力。
- 满足Lindorm实例的数据强一致或者最终一致需求。
- 故障识别和切换操作都是由Lindorm自动触发，易用性高。

云原生多模数据库Lindorm在多可用区、主备容灾以及基于Paxos/Raft一致性协议的场景中功能特性对比如下：

功能特性	多可用区		主备容灾	基于Paxos/Raft一致性协议
	强一致	最终一致		
数据丢失（RPO）	0	<100ms	<1s	0
服务恢复（RTO）	1分钟	10~30s	由主备切换的时间决定。	30s~3分钟
访问RT	主可用区访问，可能存在跨多可用区读写。	就近访问，支持多区域读写可降低毛刺。	主可用区访问，可能存在跨多可用区读写。	主可用区访问，可能存在跨多可用区读写。
易用性	对业务透明。	对业务透明。	业务需改造，外置同步链路，用户自行切换。	对业务透明。
最小需要存储日志和数据的可用区数目	• 存储日志：3个 • 存储数据：2个	• 存储日志：2个 • 存储数据：2个	• 存储日志：2个 • 存储数据：2个	• 存储日志：3个 • 存储数据：3个

说明 无论是强一致还是弱一致，在Lindorm多可用区部署下，Lindorm实例宽表的故障识别，切换都由Lindorm实例自行决定，整个过程对用户透明，并且用户一直访问一个Lindorm实例，无需再开发中间件来连接多个实例和实例间的切换。

- 如果业务没有强一致的需求，建议选择Lindorm多可用区实例，并设置表为最终一致。
- 如果业务有强一致的需求，建议选择Lindorm多可用区实例，并设置表为强一致。

如何购买多可用区实例

您可以通过控制台购买云原生多模数据库Lindorm多可用区实例，具体操作请参见创建多可用区实例。

如何选择一致性

一般情况下，云原生多模数据库Lindorm实例新建的表都会默认设置为最终一致，最终一致可以满足大部分用户的需求，达到高可用和降低毛刺的效果。因此推荐使用最终一致性模型。事实上，绝大部分的业务都能接受读取100ms前的数据，而且Lindorm的客户端支持就近读写，如果用户的读写客户端在同一可用区，大部分情况下客户端读到的都是最新数据（没有因为毛刺或者机器终止工作导致当前可用区的partition不可用或者超时等情况）。如果您的业务有以下需求，需要将表设置为强一致。

- 必须要读到最新数据。
- 使用Increment、CheckAndPut等先读后写的原子语义接口。
- 需要为表建立二级索引。

说明 强一致模式下，Lindorm无法通过读取多副本的方式来减少抖动和毛刺，如果主可用区出现故障，备可用区需要一定的时间恢复才能切换为主可用区。

如何设置表一致性

说明 由于HBase API和HBase Shell不支持一致性概念，如果您使用HBase API访问Lindorm宽表引擎，请使用HBase API和HBase Shell创建表（创建的表默认设置为最终一致性）后，再使用以下步骤对表属性进行修改。

1. 通过Lindorm-cli连接并使用宽表引擎，使用SQL访问Lindorm宽表引擎。
2. 设置表一致性。
   - 创建表格时，通过以下语句设置表为强一致。

```
CREATE TABLE dt (p1 integer, p2 integer, c1 varchar, c2 bigint,
  constraint pk primary key(p1 desc))'CONSISTENCY'='strong';
```

   创建表格时，通过以下语句设置表为最终一致。

```
CREATE TABLE dt2 (p1 integer, p2 integer, c1 varchar, c2 bigint,
  constraint pk primary key(p1 desc))'CONSISTENCY'='eventual';
```

- 创建表格后，通过以下语句修改表属性为最终一致。

```
ALTER TABLE dt SET 'CONSISTENCY'='eventual';
```

创建表格后，通过以下语句修改表属性为强一致。

```
ALTER TABLE dt2 SET 'CONSISTENCY'='strong';
```

数据写入

Lindorm多可用区版本与单可用区版本的使用方式完全一样。连接Lindorm宽表引擎并写入数据，具体操作请参见通过Lindorm-cli连接并使用宽表引擎。

数据导入

- 使用API方式将数据导入Lindorm多可用区版本与单可用区方法相同，使用控制台上提供的连接地址即可。
- 使用bulkload方式导入Lindorm多可用区版本，需要在两个可用区都导入一份数据，相关使用咨询方法请参见技术支持。

7.4.10. 审计日志

7.4.10.1. 开通审计日志

云原生多模数据库Lindorm基于阿里云日志服务SLS，推出审计日志功能，通过审计日志可以精准分析指定时间段的所有数据操作，同时可以按照指定字段进行过滤筛选。本文介绍通过云原生多模数据库Lindorm控制台开通审计日志功能。

前提条件

Lindorm宽表引擎小版本为2.2.18及以上，关于升级小版本的操作请参见升级小版本。

注意事项

- Lindorm单节点和LindormServerless版本不支持审计日志功能。
- 开通审计日志后，系统默认在Lindorm实例所属的地域中创建Project和专属Logstore。
  - Project的名称格式为*aliyun-product-阿里云账号ID-地域ID*。
  - 专属Logstore的名称为*lindorm_audit_log*。
- 专属Logstore不支持写数据、修改索引、删除索引、修改属性等操作，但在查询、统计、告警等功能上没有限制。
- 不支持在日志服务端（例如日志服务控制台、API、SDK等）修改专属Logstore的数据存储时间，您可以通过云原生多模数据库Lindorm控制台修改。
- 日志服务SLS处于可用状态，否则全量日志分析功能无法使用。

费用说明

审计日志的收费根据审计日志的存储空间和保存时长按量收取费用，目前Lindorm的审计日志功能不额外计费，相关收费的详细内容请参见日志服务详细价格 。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**宽表引擎 > 审计日志**。
5. 打开**当前状态**开关。
6. 在**审计日志**对话框，单击**确定**。

7.4.10.2. 查询审计日志

您可以通过审计日志查看一段时间内的请求操作记录，还可以使用各种过滤条件对查询到的记录进行筛选。

前提条件

已开通云原生多模数据库Lindorm的审计日志功能，具体操作请参见开通审计日志。

查看审计日志

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**宽表引擎 > 审计日志**，查看Lindorm宽表的审计日志详情。

> **说明** 默认显示最近15分钟内的日志详情。

使用过滤条件筛选日志

您可以通过不同的过滤条件筛选日志，精确定位目标记录。

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**宽表引擎 > 审计日志**。
5. 单击日志中不同字段进行过滤，字段说明如下：



字段名	说明
instance_id	Lindorm实例ID。
namespace	Lindorm宽表的命名空间。
user	Lindorm实例的用户名。
table	Lindorm宽表的表名。
ip	存储宽表数据的服务器IP地址。
operation	操作类型，包括DML和DDL。 ○ DML：scan/get/batchGet/exist/put/checkAndPut/delete/checkAndDelete/checkAndMutation/batchPut/append/increment/select/aggregate/upsert ○ DDL：createTable/disableTable/dropTable/truncateTable/createNamespace/createUser/createIndex/deleteIndex/buildIndex
param_details	对数据进行操作的具体信息，包括请求参数等。
request_time	请求发起时间。
took	请求消耗的时间，仅针对DML操作。

> **说明** 更多审计日志的查询方法请参见查询和分析日志。

查询指定时间段的日志

通过指定的时间段，您可以定位指定时间范围内产生的日志。

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**宽表引擎 > 审计日志**。
5. 单击页面右上角的 15分钟（相对） ，并单击不同页签选择需要指定的时间段，时间页签说明如下表。



页签	说明
相对	选择一个相对于当前时间点的时间段。将鼠标悬浮于任意选项上时，可在时间详情区域查看该选项对应的时间段。
整点时间	选择一个分钟级以上的整点时间段。将鼠标悬浮于任意选项上时，可在时间详情区域查看该选项对应的时间段。
自定义	自定义一个时间段，输入自定义时间后需单击确定按钮使之生效。

7.5. 数据生态

7.5.1. 数据导入与迁移

Lindorm支持从云数据库RDS、MySQL、不同HBase版本（包括云数据库HBase和自建HBase集群）导入数据，并进行实时的增量同步，也支持将数据迁移到不同版本的HBase中。数据的导入和迁移，都是基于LTS服务实现。

RDS/MySQL历史数据导入

将RDS/MySQL的历史数据导入到Lindorm，可以使用LTS，请参见RDS全量导入。

RDS增量数据同步

通过LTS数据同步服务可以将RDS增量数据实时同步到Lindorm中，使用说明步骤如下：

1. 开通并登录LTS。
2. 在LTS中添加Lindorm集群，具体操作方法参见添加HBase数据源。
3. RDS增量导入。

云HBase标准版或开源HBase迁移HBase增强版（Lindorm宽表）

使用LTS数据同步服务可以将云数据库HBase标准版或开源HBase的历史和增量数据迁移到HBase增强版（Lindorm宽表）。

- 数据导入（历史数据迁移）
  i. 开通并登录LTS。
  ii. 在LTS中添加HBase集群和HBase增强版集群，具体操作方法请参见添加HBase数据源。
  iii. 添加历史数据迁移任务。
- 增量数据同步

i. 开通并登录LTS。
ii. 在LTS中添加HBase集群和HBase增强版集群，具体操作方法请参见添加HBase数据源。
iii. 添加实时数据同步任务。

HBase增强版历史和增量数据迁移至云数据库HBase标准版

使用LTS数据同步服务可以将HBase增强版历史和增量数据迁移到云数据库HBase标准版或开源HBase，请参见LTS使用说明。

- 数据导入（历史数据迁移）
  i. 开通并登录LTS。
  ii. 在LTS中添加HBase集群和HBase增强版集群，具体操作方法参见添加HBase数据源。
  iii. 添加历史数据迁移任务。
- 增量数据同步
  i. 开通并登录LTS。
  ii. 在LTS中添加HBase集群和HBase增强版集群，具体操作方法参见添加HBase数据源。
  iii. 添加实时数据同步任务。

7.5.2. 使用DataWorks或者DataX导入数据

云原生多模数据库Lindorm提供LTS服务能够支持各种Lindorm版本之间相互迁移和实时同步，并且支持同步RDS、Loghub的实时数据到Lindorm。DataX是阿里巴巴集团内被广泛使用的离线数据同步工具，实现MySQL、Oracle、SqlServer、PostgreSQL、HDFS、Hive、ADS、Lindorm、TableStore(OTS)、MaxCompute(ODPS)、DRDS等各种异构数据源之间高效的数据同步功能。

前提条件

导入数据前，请根据DataX所部署的ECS需要添加白名单，才能访问Lindorm实例。同时，如果ECS与HBase增强版不在一个VPC网络下，则需要使用公网地址访问。

使用DataX进行数据同步

使用DataX进行数据同步有两种方法：

- 使用阿里云DataWorks的数据集成服务配置DataX任务（推荐使用该方法）。
- 使用开源DataX配置同步任务。

使用DataWorks配置DataX

1. 创建工作空间，具体操作请参见创建工作空间。
2. 创建资源组，资源组类型和说明如下表。

资源组类型	配置文档	特点	注意事项
独享资源组	独享资源组模式	独享资源组的机器由DataWorks自动买出，运维完全托管于系统，您无需担心机器服务和可用性等问题。	独享资源不支持跨地域使用。例如，华东2（上海）地域的独享资源，只能给华东2（上海）地域的工作空间使用（无法绑定其他区域的VPC），并且独享资源组不能夸Vswtich访问Lindorm集群。
自定义资源组	新增和使用自定义数据集成资源组	仅DataWorks企业版及以上版本支持自定义资源组。自定义资源组的ECS机器由用户自己买出，用户可以将ECS买在Lindorm的VPC内，从而用内网访问Lindorm，否则只能用外网访问。	自定义资源组的机器完全可控、可登录访问，但是需要自行安装/运维/升级DataX版本（配置文档中有安装方法）。
默认资源组	无	默认资源组机器无法从内网访问Lindorm所在VPC，只能通过公网访问Lindorm。	公网访问Lindorm会在DataWorks产生额外费用。

3. 配置网络。

- 独享资源组网络配置
  a. 将独享资源组与Lindorm所在VPC绑定，详情请参见独享资源组模式。
  b. 在VPC控制台中找到独立资源组所绑定的VPC和Vswtich所在的IPv4网段。将IPv4添加至Lindorm白名单，具体操作请参见设置白名单。
- 自定义资源组网络配置
  自定义资源组的机器都是用户自己购买，因此能看到每台ECS的具体IP，将这些IP添加至Lindorm白名单，具体操作请参见设置白名单。
- 默认资源组网络配置
  默认资源组机器的IP段详见IP网段，将区域对应的IP添加至Lindorm白名单，具体操作请参见设置白名单。

4. 创建同步任务，具体操作请参见创建同步任务。
5. 修改插件配置。
   - 如果使用Lindorm SQL方式访问，请参考Lindorm Reader和Lindorm Writer配置文档中TableService模型的配置。
   - 如果使用HBase兼容方式访问，请参考Lindorm Reader和Lindorm Writer配置文档中WideColumn模型的配置。

   > **说明** 上述配项中的lindorm.client.seedserver参数为Lindorm控制台**数据库连接**页面的HBase兼容地址，连接地址格式为host:30020。

   ```
   "configuration": {
     "lindorm.client.namespace" : "namespace",
     "lindorm.client.seedserver" : "host:30020",
     "lindorm.client.username" : "root",
     "lindorm.client.password" : "root"
   }
   ```

   > **说明**
   > - lindorm.client.namespace：访问的Lindorm表所在的namespace。
   > - Lindorm.client.seedserver：用户控制台上提供的Java API访问地址，用户可以参见连接集群获得。
   > - Lindorm.client.username和password：Lindorm增强版中用户自己创建的用户名和密码（默认均为root），用户必须保证提供的用户有读写Lindorm增强版中表的权限（默认提供的root用户已经具有读写所有表的权限）。关于用户和ACL，请参见连接集群。

6. 配置任务资源组。

使用开源DataX同步任务（仅支持使用HBase API访问的表）

1. 下载DataX安装包。

   > **说明** 如果已经有DataX安装包，需要下载最新版本的DataX安装包，同时需要导入JAR包。仅需要下载1.x版本（选择最新版本）的aliLindorm-connector-1.x jar文件，将下载的JAR包导入 `DataX/plugin/writer/hbase11xwriter/libs` 目录中，如果需要用DataX读取Lindorm增强版的数据，则需将此JAR包也导入 `DataX/plugin/reader/hbase11xreader/libs` 目录中。

2. 解压DataX安装包。
3. 编辑配置文件。在DataX中，读取Lindorm的插件为hbase11xreader，此插件的具体配置请参见hbase11xreader。写入Lindorm的插件为hbase11xwriter，此插件的具体配置请参见hbase11xwriter。读写Lindorm的Lindormconfig部分配置如下，其他部分与官方配置保持一致。

   ```
   "hbaseConfig": {
     "hbase.client.connection.impl" : "com.alibaba.hbase.client.AliHBaseUEConnection",
     "hbase.client.endpoint" : "host:30020",
     "hbase.client.username" : "root",
     "hbase.client.password" : "root"
   }
   ```

> 说明
> - hbase.client.connection.impl：固定配置不需要修改。
> - hbase.client.endpoint：Lindorm控制台**数据库连接**页面的HBase兼容地址。
> - hbase.client.username和password：Lindorm增强版中用户自己创建的用户名和密码（默认均为root），用户必须保证提供的用户有读写Lindorm增强版中表的权限（默认提供的root用户已经具有读写所有表的权限）。关于用户和ACL，请参见连接集群。

4. 启动DataX开始迁移数据，DataX的具体使用方式请参见官方文档。

7.5.3. 通过Spark访问Lindorm

本文主要介绍如何通过Spark访问Lindorm宽表。

访问准备

Lindorm宽表支持从Spark访问，您需要添加alihbase-connector的依赖，具体步骤如下：

1. 确认Spark环境中依赖的开源HBase客户端版本以及安装目录，可以通过yarn logs -applicationId xxx查看具体执行日志中加载的版本和路径，hbase-client-xxx.jar。
2. 确认添加alihbase-connector插件的版本，版本对应表请参见通过Maven方式升级HBase客户端。
3. 下载对应版本的alihbase-connector，下载地址请参见通过导入JAR文件升级HBase客户端。
4. 将alihbase-connector移动至开源HBase客户端目录。

添加Lindorm访问配置

- 方式一：通过配置文件添加访问配置。
  在配置文件 `hbase-site.xml` 中增加下列配置项：

```
<configuration>
      <!--
    集群的连接地址，在控制台页面的数据库连接界面获得（注意公网地址和VPC内网地址）。
    -->
    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>ld-xxxx-proxy-hbaseue.hbaseue.xxx.rds.aliyuncs.com:30020</value>
    </property>
    <!--
   设置用户名密码，默认root:root，可根据实际情况调整。
    -->
    <property>
        <name>hbase.client.username</name>
        <value>root</value>
    </property>
    <property>
        <name>hbase.client.password</name>
        <value>root</value>
    </property>
    <!--
    如果您直接依赖了阿里云HBase客户端，则无需配置connection.impl参数，如果您依赖了alihbase-connector，则需要配置此参数。
    -->
    <!--property>
        <name>hbase.client.connection.impl</name>
       <value>org.apache.hadoop.hbase.client.AliHBaseUEClusterConnection</value>
    </property-->
</configuration>
```

- 方式二：通过代码在Configuration中添加参数。

```
// 新建一个Configuration
Configuration conf = HBaseConfiguration.create();
// 集群的连接地址，在控制台页面的数据库连接界面获得（注意公网地址和VPC内网地址）。
conf.set("hbase.zookeeper.quorum", "ld-xxxx-proxy-hbaseue.hbaseue.xxx.rds.aliyuncs.com:30020");
// 设置用户名密码，默认root:root，可根据实际情况调整。
conf.set("hbase.client.username", "root")
conf.set("hbase.client.password", "root")
// 如果您直接依赖了阿里云HBase客户端，则无需配置connection.impl参数，如果您依赖了alihbase-connector，则需要配置此参数。
//conf.set("hbase.client.connection.impl", AliHBaseUEClusterConnection.class.getName());
```

Spark访问示例

添加HBaseue访问配置。

```
test(" test the spark sql count result") {
//添加HBaseue访问配置
var conf = HBaseConfiguration.create
conf.set("hbase.zookeeper.quorum", "ld-xxxx-proxy-hbaseue.hbaseue.xxx.rds.aliyuncs.com:30020")
conf.set("hbase.client.username", "test_user")
conf.set("hbase.client.password", "password")
//创建表
val hbaseTableName = "testTable"
val cf = "f"
          val column1 = cf + ":a"
val column2 = cf + ":b"
var rowsCount: Int = -1
var namespace = "spark_test"
val admin = ConnectionFactory.createConnection(conf).getAdmin()
val tableName = TableName.valueOf(namespace, hbaseTableName)
val htd = new HTableDescriptor(tableName)
htd.addFamily(new HColumnDescriptor(cf))
admin.createTable(htd)
//插入测试数据
val rng = new Random()
val k: Array[Byte] = new Array[Byte](3)
val famAndQf = KeyValue.parseColumn(Bytes.toBytes(column))
val puts = new util.ArrayList[Put]()
var i = 0
for (b1 <- ('a' to 'z')) {
    for (b2 <- ('a' to 'z')) {
      for (b3 <- ('a' to 'z')) {
          if(i < 10) {
             k(0) = b1.toByte
             k(1) = b2.toByte
             k(2) = b3.toByte
             val put = new Put(k)
             put.addColumn(famAndQf(0), famAndQf(1), ("value_" + b1 + b2 + b3).getBytes())
             puts.add(put)
             i = i + 1
           }
        }
       }
 }
 val conn = ConnectionFactory.createConnection(conf)
 val table = conn.getTable(tableName)
 table.put(puts)


 //创建spark表
 val sparkTableName = "spark_hbase"
 val createCmd = s"""CREATE TABLE ${sparkTableName} USING org.apache.hadoop.hbase.spark
                       |    OPTIONS ('catalog'=
                       |    '{"table":{"namespace":"$${hbaseTableName}",                    "name":"${hbaseT
ableName}"},"rowkey":"rowkey",
                       |    "columns":{
                       |    "col0":{"cf":"rowkey", "col":"rowkey", "type":"string"},
                       |    "col1":{"cf":"cf1", "col":"a", "type":"string"},
                       |    "col2":{"cf":"cf1", "col":"b", "type":"String"}}}'
                       |    )""".stripMargin

 println(" createCmd: \n" + createCmd + " rows : " + rowsCount)
 sparkSession.sql(createCmd)

 //执行count sql
 val result = sparkSession.sql("select count(*) from " + sparkTableName)
 val sparkCounts = result.collect().apply(0).getLong(0)
 println(" sparkCounts : " + sparkCounts)
```

7.5.4. 通过Hive访问Lindorm

使用HBase API的用户，支持通过Hive访问Lindorm，具体步骤如下：

1. 参考 通过导入JAR文件升级HBase客户端，将hive/lib中hbase-client jar替换为阿里云发布的版本，注意不要删除hive-hbase-handler-{version}.jar，这是Hive访问HBase的逻辑代码jar包。
2. 如果之前在hive/.hivercc中或者在启动Hive时使用--auxpath参数指定了hbase依赖的，需要把加载的jar包换成新的alihbase开头的jar。

连接前的准备

- 获取连接地址
- 获取用户名密码
- 将访问Lindorm的Hive机器IP加入白名单
  所有访问Lindorm的Hive机器的IP，必须加入Lindorm集群的白名单中，否则无法访问，添加白名单请参见设置白名单。

在Hive中配置连接参数

在Hive中配置连接Lindorm的参数有两种参数，一种是直接配置在hive-site.xml文件中。在这个文件中加入如下配置项：

```
<configuration>
    <!--
   集群的连接地址，在控制台页面的数据库连接界面获得（注意公网地址和VPC内网地址）
   -->
   <property>
       <name>hbase.zookeeper.quorum</name>
       <value>ld-xxxx-proxy-hbaseue.hbaseue.xxx.rds.aliyuncs.com:30020</value>
   </property>
   <!--
   设置用户名密码，默认root:root，可根据实际情况调整
    -->
    <property>
       <name>hbase.client.username</name>
       <value>root</value>
    </property>
    <property>
       <name>hbase.client.password</name>
       <value>root</value>
    </property>
</configuration>
```

另一种方式是在Hive的Client中直接用命令的方式配置：

```
set hbase.zookeeper.quorum=ld-xxxx-proxy-hbaseue.hbaseue.xxx.rds.aliyuncs.com:30020
set hbase.client.username=root
set hbase.client.password=root
```

当完成这些配置后，就可以在Hive中，像访问HBase一样，自由地访问Lindorm。

Hive简单使用指南

使用Hive访问Lindorm的方式与访问一致，Lindorm作为Hive外表，简单使用指南可以参考hive访问HBase，如下：

- 进入hive cli命令行

```
[root@emr-header-2 hive-conf]# hive

Logging initialized using configuration in file:/etc/ecm/hive-conf-2.3.3-1.0.1/hive-log4j2.properties Async: true
Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
hive>
    > 
```

云栖社区 yq.aliyun.com

- 创建HBase表

```
1. CREATE TABLE hive_hbase_table(key int, value string)
2. STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
3. WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:val")
4. TBLPROPERTIES ("hbase.table.name" = "hive_hbase_table", "hbase.mapred.output.outputtable" = "hive_hbase
_table");
```

- Hive中向hbase插入数据

```
1. insert into hive_hbase_table values(212,'bab');
```

```
hive> insert into hive_hbase_table values(212,'bab');
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
Query ID = root_20181014173030_a0e99198-9aa5-4d29-b011-dc7b36365a20
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1536221485395_0084, Tracking URL = http://emr-header-1.cluster-74778:20888/proxy/application_1536221485395_0084/
Kill Command = /usr/lib/hadoop-current/bin/hadoop job  -kill job_1536221485395_0084
Hadoop job information for Stage-3: number of mappers: 1; number of reducers: 0
2018-10-14 17:30:40,833 Stage-3 map = 0%,  reduce = 0%
2018-10-14 17:30:47,252 Stage-3 map = 100%,  reduce = 0%, Cumulative CPU 3.66 sec
MapReduce Total cumulative CPU time: 3 seconds 660 msec
Ended Job = job_1536221485395_0084
MapReduce Jobs Launched:
Stage-Stage-3: Map: 1   Cumulative CPU: 3.66 sec   HDFS Read: 11867 HDFS Write: 0 SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 660 msec
OK
Time taken: 17.385 seconds
```

云栖社区 yq.aliyun.com

- 查看云HBase表,hbase表已创建，数据也已写入。

```
[root@iZbp16ku9i9clejitib6dzZ ~]# hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/apps/t-apsara-hbase-1.4.6.3/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/apps/t-emr-hadoop-2.7.2.2/share/hadoop/common/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.4.6.3, r89ac288a5add370c07548ec3ce25f6e1f3210d23, Fri Jul  6 14:13:46 CST 2018

hbase(main):001:0> list
TABLE
A_TABLE
BAKE
BASE_TABLE
B_IDX
B_IDX1
B_IDX2
DEFAULT.TEST_IDX
MY_TABLE
PROD_METRICS
SYSTEM.MUTEX
SYSTEM:CATALOG
SYSTEM:FUNCTION
SYSTEM:SEQUENCE
SYSTEM:STATS
[redacted]
hive_hbase_table
[redacted]
tv
18 row(s) in 0.2110 seconds
```

云栖社区 yq.aliyun.com

- 在HBase中写入数据，并在Hive中查看。

```
hbase(main):005:0> put 'hive_hbase_table','132','cf1:val','acb'
0 row(s) in 0.0430 seconds
```

云栖社区 yq.aliyun.com

- 在Hive中查看：

```
hive> select * from hive_hbase_table;
OK
132     acb
212     bab
Time taken: 0.273 seconds, Fetched: 2 row(s)
```

云栖社区 yq.aliyun.com

- Hive删除表，HBase表也删除。

```
hive>
    >
    >
    > drop table hive_hbase_table;
OK
Time taken: 6.307 seconds
```

云栖社区 yq.aliyun.com

- 查看hbase表，报错不存在表。

```
hbase(main):008:0* scan 'hive_hbase_table'
ROW                                                    COLUMN+CELL

ERROR: Unknown table hive_hbase_table!
```

云栖社区 yq.aliyun.com

- 如果HBase表已存在，可在Hive中HBase外表进行关联，外部表在删除时不影响HBase已创建表。
- 云hbase中创建hbase表，并put测试数据。

```
hbase(main):020:0* create 'hbase_table','f'
0 row(s) in 1.3010 seconds

=> Hbase::Table - hbase_table
hbase(main):021:0> put 'hbase_table','1122','f:col1','hello'
0 row(s) in 0.0190 seconds

hbase(main):022:0> put 'hbase_table','1122','f:col2','hbase'
0 row(s) in 0.0110 seconds
```

云栖社区 yq.aliyun.com

- Hive中创建HBase外部关联表，并查看数据。

```
hive> create external table hbase_table(key int, col1 string, col2 string)
    > STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
    > WITH SERDEPROPERTIES ("hbase.columns.mapping" = "f:col1,f:col2")
    > TBLPROPERTIES("hbase.table.name" = "hbase_table", "hbase.mapred.output.outputtable" = "hbase_table");
OK
Time taken: 0.129 seconds
hive> select * from hbase_table;
OK
1122    hello   hbase
Time taken: 0.181 seconds, Fetched: 1 row(s)
hive>
```

云栖社区 yq.aliyun.com

- 删除Hive表不影响HBase已存在表。

```
hive> drop table hbase_table;
OK
Time taken: 0.102 seconds
hive>
```

云栖社区 yq.aliyun.com

Hive更多操作HBase步骤，请参见更多操作

> **注意** 目前暂不支持Hive直接读取Lindorm的底层文件。如果是采用关联/创建外表的方式读写是完全兼容的。

7.5.5. 通过Flink访问Lindorm

使用Flink访问云原生多模数据库Lindorm宽表分为社区版Flink和阿里云实时计算Flink。这两种方式访问Lindorm宽表的使用方法不一样，本文介绍通过Flink访问Lindorm宽表的注意事项和相关文档。

背景信息

您可以将云原生多模数据库Lindorm宽表作为Flink中的维表或者结果表，通过Flink SQL或者Flink DataStream访问Lindorm宽表。

注意事项

- 使用Lindorm宽表作为Flink的维表或者结果表时，创建表的DDL语句需要使用连接地址。通过Flink访问Lindorm宽表使用的是Lindorm宽表引擎的**HBase Java API访问地址**。如果使用新创建的用户，请确保该用户有访问Flink表的读写权限，赋予权限的具体操作请参见为指定用户赋予权限。
- 如果您使用社区版Flink访问Lindorm宽表，仅支持访问HBase like表，也就是说，您通过lindorm-cli创建的SQL表不能作为社区版Flink维表或者结果表，只能通过hbaseue Shell创建HBase like表提供给社区版Flink进行访问。但阿里云实时计算Flink无此限制。
- 如果您使用社区版Flink访问Lindorm宽表，需要修改Flink connector配置中的Maven依赖。
  - 如果您使用Flink SQL访问Lindorm宽表，请查看并核对Flink connector配置中的HBase依赖版本，具体内容请参见HBase依赖版本。

- 如果您使用Flink DataStream访问Lindorm宽表，请确定依赖的hbase-client版本，修改*pom.xml*文件并打包，版本对应关系请参见hbase-client同alihbase-connector版本对应关系。

  > **说明**
  > - 您也可以直接下载*pom.xml*文件包，下载地址请单击pom文件包。
  > - 打包命令：`mvn clean package -X -e -Dcheckstyle.skip` 。

- 如果您定制开发了Flink或者hbase-client，请修改Flink connector配置或者hbase-client版本并打包，修改后如果出现其他依赖异常（例如NoSuchMethod或者NoSuchClass等）请提交工单。
- 如果您使用阿里云实时计算Flink并采用Flink SQL方式访问Lindorm宽表，那么可以直接使用Lindorm Connector。

创建维表

- 如果您使用社区版Flink访问Lindorm宽表，具体操作请参见创建维表。
- 如果您使用阿里云实时计算Flink访问Lindorm宽表，具体操作请参见创建Lindorm维表。

创建结果表

- 如果您使用社区版Flink访问Lindorm宽表，具体操作请参见创建结果表。
- 如果您使用阿里云实时计算Flink访问Lindorm宽表，具体操作请参见创建Lindorm结果表。

7.5.6. 通过Grafana访问Lindorm

Grafana是指标数据图形化展示工具，可以从多个数据源采集数据并以图表展示。本文介绍如何通过Grafana访问Lindorm中的数据，可以利用Grafana丰富易用的可视化工具更好地监控和分析来自Lindorm的数据。

前提条件

- 已安装5.x-7.x版本的Grafana，请单击下载Grafana。
- 确保ECS实例与Lindorm实例属于同一专有网络。
- 确保ECS实例已开通公网访问。如果您未分配公网IP地址，可以申请弹性公网IP，具体操作，请参见申请弹性公网IP。
- 将ECS实例的内网IP地址添加至Lindorm实例的白名单中。具体操作，请参见设置白名单。
- 该数据源只适用于TableService结构的时序数据（time-series data）。

安装Grafana插件

1. 进入Grafana插件安装目录。

   ```
   cd *
   ```

   > **说明** 如果在Ubuntu系统中，需要在*/var/lib/grafana/plugins/*中安装插件。

2. 执行以下命令下载插件。

   ```
   wget https://github.com/aliyun/aliyun-apsaradb-hbase-demo/archive/refs/heads/master.zip
   ```

3. 执行以下命令解压插件。

   ```
   unzip master.zip
   ```

4. 在已部署Grafana的机器中，打开*grafana.in*配置文件。
   - macOS系统的文件路径：*/usr/local/etc/grafana/grafana.ini*。
   - Linux系统的文件路径：*/etc/grafana/grafana.ini*。
5. 在*grafana.in*配置文件中的plugins项设置allow_loading_unsigned_plugins参数。

   ```
   allow_loading_unsigned_plugins = lindorm_lql_datasource
   ```

6. 重启Grafana服务。

   ```
   service grafana-server restart
   ```

配置数据源

1. 登录Grafana服务。
2. 在左侧菜单栏，选择  >Data Sources。
3. 在Data Sources页签，单击Add data source。
4. 在Add data source页面，单击Alibaba Lindorm对应的Select按钮。
5. 配置数据源参数，包括通讯地址和端口。

> **说明** 通讯地址和端口使用Lindorm宽表SQL地址及端口，可以通过云原生多模数据库Lindorm控制台获取，具体操作请参见访问实例。

添加仪表盘

通过Grafana查询数据包括两种方式。

- 通过Query Configurator方式查询数据
- 通过Query Editor方式查询数据

> **说明** 通过Query Configurator方式查询数据更易于使用但存在一定的限制，通过Query Editor方式查询数据需要熟悉Lindorm SQL语言。

通过Query Configurator方式查询数据

Query Configurator是查询数据最简单的方法。先输入keyspace和table name，然后选择columns。如果keyspace和table name为正确，数据源将自动建议columns名称。参数说明如下。

参数名称	说明
Time Column	表示存储timestamp value的Column，用于回答何时问题。
Value Column	表示存储需要显示的值的Column，可以是数值、温度或其他属性。
ID Column	唯一标识数据来源的Column，例如sensor_id、shop_id或其他数据来源。

> **说明**
> 全表扫描时需要使用ALLOW FILTERING选项，但是建议避免使用该选项。

通过Query Editor方式查询数据

Query Editor是查询数据较强大的方法。需要启用Query Editor，请单击  。

Query Editor可以用Lindorm SQL的语法执行。更多语法信息，请参见Lindorm宽表SQL语法手册。

```
SELECT id, value, created_at FROM test.test WHERE id IN (99051fe9-6a9c-46c2-b949-38ef78858dd1, 99051fe9-6a9c
-46c2-b949-38ef78858dd0) AND created_at > $__timeFrom and created_at < $__timeTo
```

SELECT表达式参数说明如下：

参数名称	说明
id	表达式第一个属性，表示唯一标识数据。
Value	表达式第二个属性，表示需要显示的值。
Timestamp	表示值的时间戳。

> **说明** 通过添加 `_timeTo` 占位符可以实现按时间过滤数据。如果没有添加占位符获取到的数据为整个时间段的。

操作示例

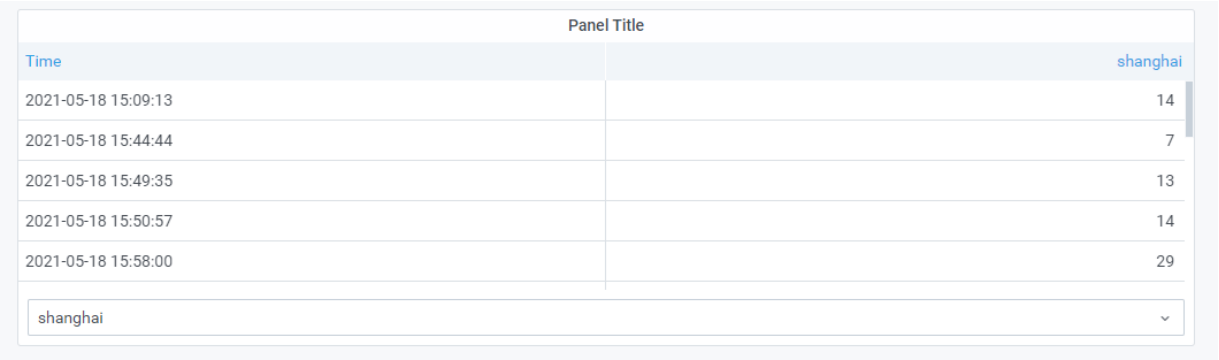
假如想绘制一张包含各地气温随时间变化曲线的图表。Lindorm中的数据如下，包括气温temp，测量时间local_time，城市city和索引id。

```
create table log.temp
(
    local_time timestamp,
    temp int,
    id int,
    city text,
    primary key ((id))
)
```

通过Query Editor方式查询，Lindorm SQL语句如下所示。

```
SELECT city, temp , local_time FROM log.temp WHERE local_time > $__timeFrom and local_time < $__timeTo  ALLOW FILTERING
```

查询到的数据如下图所示。



在Visualization中选择Graph，并调整时间段，得到的图表如下所示。



7.5.7. 通过Redash访问Lindorm

Redash是一款开源的BI工具，提供了基于Web的数据库查询和数据可视化功能，本文介绍如何使用Redash连接Lindorm。

前提条件

已安装Redash，具体信息，请参见Redash官网。

安装插件

- 如果用拉取Docker镜像方式部署Redash，则需要将redash_server，redash_scheduler，redash_adhoc_worker，redash_scheduled_worker四个容器中的*/app*目录作为 `$REDASH_SETUP_FOLDER` ，具体操作如下：
  i. 下载*Lindorm.png*和*lindorm.py*文件，下载链接，请参见redash。

ii. 将*Lindorm.png*文件复制到*REDASH_SETUP_FOLDER/client/app/assets/images/db-logos/*和*REDASH_SETUP_FOLDER/client/dist/images/db-logos/*目录下。

iii. 将*lindorm.py*文件复制到*$REDASH_SETUP_FOLDER/redash/query_runner/*目录下。

iv. 在文件*$REDASH_SETUP_FOLDER/redash/settings/__init__.py*中按如下方式增加一行。

```
default_query_runners = [
  'redash.query_runner.athena',
  'redash.query_runner.big_query',
  ........
  'redash.query_runner.uptycs',
  'redash.query_runner.lindorm',      # Add this line
]
```

v. 重启Redash。

- 如果用源码的方式部署Redash，则需要将 `$REDASH_SETUP_FOLDER` 作为代码的根目录。

配置数据源

1. 登录Redash。
2. 单击页面右上角的图标。
3. 在**Data Sources**页签，单击**New data source**。
4. 在**New data source**页面，单击Lindorm对应的图标。
5. 配置Lindorm的连接地址、端口、用户名和密码。

   > **说明** 请使用CQL访问地址和端口。

6. 单击**Create**。

   您可以通过**Test Connection**检测连接状态。

操作示例

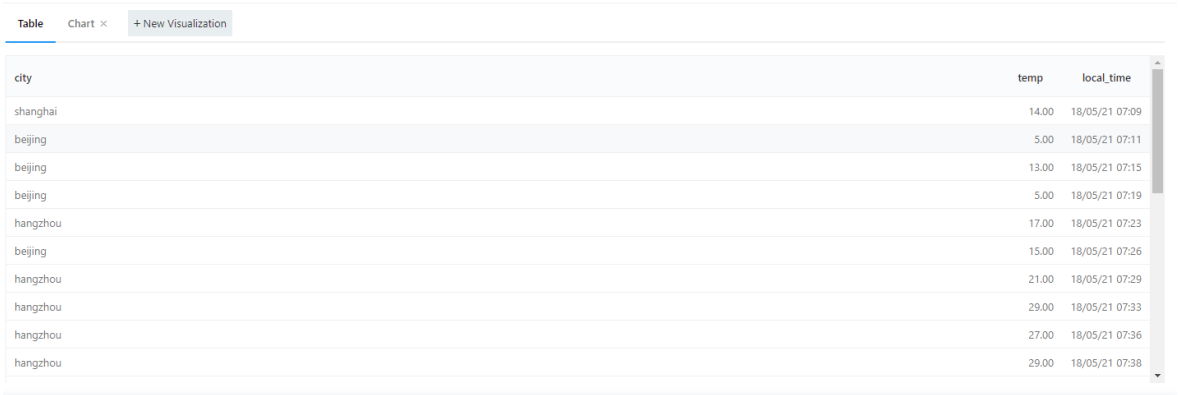
如果您想绘制一张包含各地气温随时间变化的曲线图表，数据库按照如下方式设计，包含气温（temp）、测试时间（local_time）、城市（city）、索引（id）这几个字段：

```
CREATE TABLE log.temp
(
    local_time timestamp,
    temp int,
    id int,
    city text,
    primary key ((id))
)
```

1. 在查询编辑器（Query Editor）页面，按如下方式编写CQL语句。

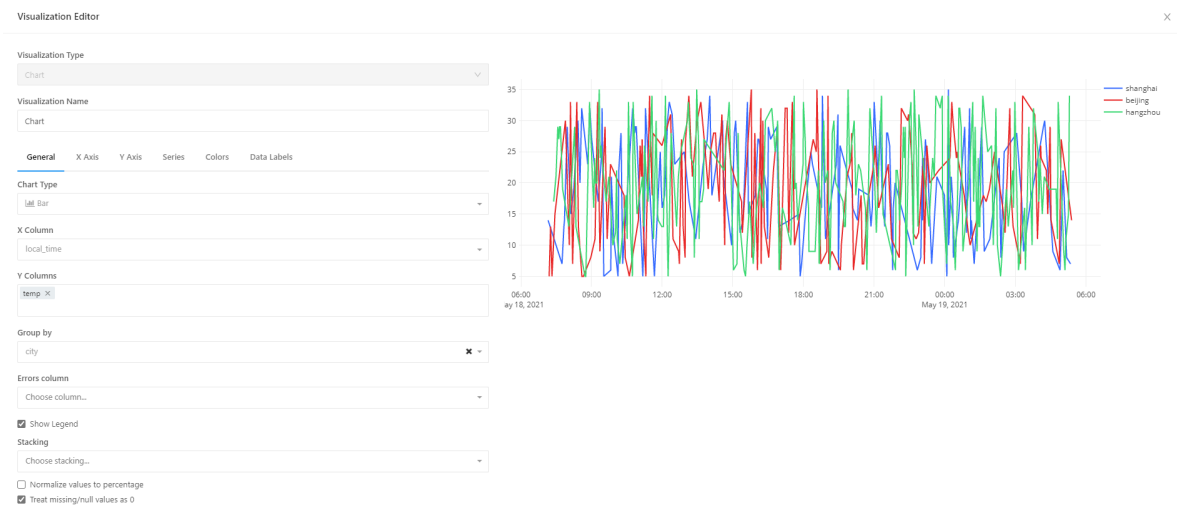
   ```
   SELECT city, temp, local_time FROM temp
   ```

2. 查询的数据如下。



3. 单击**New Visualization**。

   X Column选择`local_time`，**Y Columns**选择`temp`，**Group by**选择`City`，绘制的图标如下。



7.5.8. 通过DataWorks将MongoDB数据迁移至Lindorm宽表

DataWorks（大数据开发治理平台）是阿里云重要的PaaS（Platform-as-a-Service）平台产品，支持多种计算和存储引擎服务。本文介绍通过DataWorks将MongoDB的离线数据迁移至Lindorm宽表。

背景信息

有关大数据开发治理平台DataWorks，具体请参见什么是DataWorks。

注意事项

将MongoDB的离线数据迁移至Lindorm宽表时，需要将MongoDB嵌套的JSON数据展开成水平结构，并且不涉及数据的变换。

> **说明** 如果迁移过程中需要对某些数据进行处理，例如对MongoDB数据中主键进行MD5散列，需要按照如下步骤操作：
>
> 1. 通过DataWorks将MongoDB数据迁移至MaxCompute（ODPS）。
> 2. 在MaxCompute（ODPS）上执行SQL处理数据。
> 3. 通过DataWorks将MaxCompute数据迁移至Lindorm宽表。

准备工作

将MongoDB的离线数据迁移至Lindorm宽表需要完成以下准备工作。

- MongoDB的数据准备如下：

```
{
   "id" : ObjectId("624573dd7c0e2eea4cc8****"),
   "title" : "MongoDB教程",
   "description" : "MongoDB是一个NoSQL数据库",
   "by" : "菜鸟教程",
   "url" : "http://www.runoob.com",
   "map" : {
        "a" : "mapa",
        "b" : "mapb"
    },
   "likes" : 100
}
```

- Lindorm宽表的Schema数据准备如下：

```
CREATE TABLE t1 (
  title varchar not null primary key,
  desc varchar,
  by varchar,
  url varchar,
  a varchar,
  b varchar,
  likes int);
```

- 使用大数据开发治理平台DataWorks的数据集成服务配置DataX任务，具体操作请参见使用DataWorks配置DataX。

操作步骤

1. 在DataWorks控制台配置MongoDB数据源，具体操作请参见配置MongoDB数据源。
2. 通过脚本模式配置离线数据同步任务，具体请参见通过脚本模式配置离线同步任务。
   i. 创建业务流程。
      a. 登录DataWorks控制台。
      b. 在左侧导航栏，单击**工作空间列表**。
      c. 选择工作空间所在地域后，单击相应工作空间后的**进入数据开发**。
      d. 在**数据开发**页面，鼠标悬停至图标，单击**业务流程**。
      e. 在**新建业务流程**对话框中，输入**业务名称**和**描述**。

         注意　业务名称是由大小写字母、中文、数字、下划线（_）以及小数点（.）中的一种或多种组成，且不能超过128个字符。

      f. 单击**新建**。
   ii. 创建离线同步节点。
      a. 展开业务流程，右键单击**数据集成**。
      b. 单击**新建 > 离线同步**。
      c. 在**新建节点**对话框中，输入**节点名称**，并选择**目标文件夹**。

         注意　节点名称是由大小写字母、中文、数字、下划线（_）以及小数点（.）中的一种或多种组成，且不能超过128个字符。

      d. 单击**提交**。
   iii. 配置离线同步任务的读写端。

a. 成功创建离线同步节点后，单击工具栏中的**转换脚本**。



b. 在**提示**对话框中，单击**确认**，进入脚本模式进行开发。

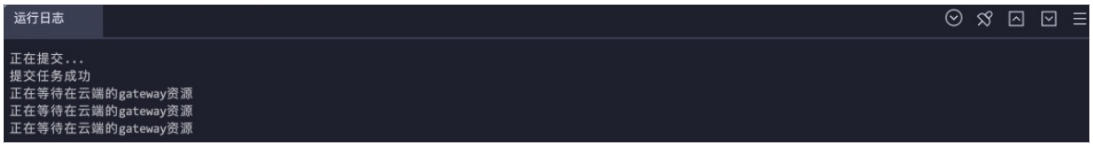
c. 脚本模式中已生成了基本的读取端配置，此时您可以继续手动配置离线同步任务的读取和写入端数据源，以及需要同步的表信息等。示例如下：

> **说明**
> - MongoDB Reader的配置参数说明具体请参见MongoDB Reader。
> - Lindorm Writer的配置参数说明具体请参见Lindorm Writer。

```
{
    "type": "job",
    "version": "2.0",
    "steps": [
        {
            "stepType": "mongodb",
            "parameter": {
                "datasource": "test_mongo",    //MongoDB数据源名称。
                "column": [
                    {
                        "name": "title",
                        "type": "string"
                    },
                    {
                        "name": "description",
                        "type": "string"
                    },
                    {
                        "name": "by",
                        "type": "string"
                    },
                    {
                        "name": "url",
                        "type": "string"
                    },
                    {
                        "name": "map.a",
                        "type": "document.string"
                    },
                    {
                        "name": "map.b",
                        "type": "document.string"
                    },
                    {
                        "name": "likes",
                        "type": "int"
                    }
                ],
                "collectionName": "testdatax"
            },
            "name": "Reader",
            "category": "reader"
        },
```

```
        },
        {
            "stepType": "lindorm",
            "parameter": {
                "configuration":  {
                    "lindorm.client.seedserver": "ld-xxxx-proxy-lindorm.lindorm.rds.aliyuncs.com
:30020",
                    "lindorm.client.username": "root",
                    "lindorm.client.namespace": "test",
                    "lindorm.client.password": "root"
                },
                "nullMode": "skip",
                "datasource": "",
                "writeMode": "api",
                "envType": 1,
                "columns": [
                    "title",
                    "desc",
                    "by",
                    "url",
                    "a",
                    "b",
                    "likes"
                ],
                "dynamicColumn": "false",
                "table": "t1",
                "encoding": "utf8"
            },
            "name": "Writer",
            "category": "writer"
        }
    ],
    "setting": {
        "executeMode": null,
        "errorLimit": {
            "record": ""
        },
        "speed": {
            "concurrent": 2,
            "throttle": false
        }
    },
    "order": {
        "hops": [
            {
                "from": "Reader",
                "to": "Writer"
            }
        ]
    }
}
```

d. 完成离线同步节点的配置后，请保存并单击脚本模式左上角的图标，在**运行日志**页签中显示数据迁移过程。



7.6. 性能白皮书

7.6.1. 测试环境

本次测试对开源自建HBase与云Lindorm进行了多个场景的性能对比。

环境配置

- 网络类型为VPC网络，并保证客户端与服务端处于同一服务可用区。
- 所有测试均在华东2（上海）地域的可用区B完成。
- 开源自建HBase使用社区1.4.9版本，为了对比方便，基于E-MapReduce提供的HBase服务替代（EMR版本：EMR-3.20.0）。
- Lindorm的版本为1.5.1.2。

名称	内容
Core节点配置	16C32G（ecs.c5.4xlarge）
Core节点数量	3
Core节点单节点磁盘大小	3.2TB
Core节点磁盘类型	高效云盘
Master节点配置	4c8g

Lindorm集群所用配置

名称	内容
Core节点配置	16C32G
Core节点数量	3
存储大小	4800GB
存储类型	标准型云存储

压力端配置

名称	内容
压力节点配置	16C32G（ecs.c5.4xlarge）
压力节点数量	1

7.6.2. 性能测试工具-Shell

AHBench是由阿里云Lindorm团队研发的benchmark测试套件，支持一键测试Lindorm和HBase等集群性能。

介绍

该测试套件集成了YCSB（Yahoo! Cloud Serving Benchmark）工具，YCSB提供了测试集合、测试流程控制、结果汇聚等功能。使用该测试套件，您可以通过简单配置，一键执行性能测试。

准备工作

- 在压测客户端部署在ECS，您需要确保云原生多模数据库Lindorm实例和ECS实例满足以下条件，以保证网络的连通性。ECS实例信息的查看方法请参见查看实例信息。
  - 所在地域相同，并建议所在可用区相同（以减少网络延时）。

- 网络类型相同。

  > **说明**
  > - 专有网络具有更高的安全性，建议使用专有网络。
  > - 如果网络类型为**专有网络**，需确保使用相同的专有网络ID。

- 将压测客户端IP地址添加至云原生多模数据库Lindorm实例的白名单中，添加方法请参见设置白名单。
- 下载测试套件AHBench，并上传到压测客户端并解压。

注意事项

- 测试压力过程中可能会使被测系统负载过高，请勿在生产环境中运行该测试工具。
- 该测试套件使用"ahbenchtest-read"与"ahbenchtest-write" 两个表作为测试用表。测试过程中可能会删除再创建这两张表。请确保这两张表可以安全地被删除。
- 确保有足够测试集群的存储空间。
- ECS为虚拟运行环境，同一机型下的性能测试结果可能存在5%～10%的波动，处于预期范围之内。

运行环境

请确保压测客户端运行环境满足：

- Linux系统
- JDK 1.8 +
- Python 2.7
- 建议客户端CPU配置为独享16Core以上

配置集群地址

在*AHBench/conf/hbase-site.xml*中配置需要测试的Lindorm或HBase集群地址。

访问Lindorm可参考通过HBase Java API连接并使用宽表引擎。并在*hbase-site.xml*中增加配置以下参数。

```
<property>
  <name>hbase.client.connection.impl</name>
  <value>org.apache.hadoop.hbase.client.AliHBaseUEClusterConnection</value>
</property>
```

配置环境变量

在AHBench/conf/ahbench-env.properties路径下配置工具运行的环境变量。

1. 执行以下命令打开*ahbench-env.properties*文件。

   ```
   vi AHBench/conf/ahbench-env.properties
   ```

2. 配置JDK的安装地址（例如：`JAVA_HOME=/usr/java/jdk1.8.0/`）。如果Java已经在系统PATH中，可以跳过该配置。
3. 配置被测HBase集群的版本，如果为1.x版本则填1，如果为2.x版本则填2。

   ```
   HBASE_VERSION=2
   ```

配置测试相关参数（可选）

在AHBench/conf/ahbench-settings.properties路径下配置测试相关的参数，如压缩、编码、线程数、数据量、字段大小等，默认不需要修改，如有需求可根据场景进行定制修改。

> **说明** 部分参数仅在特定Lindorm版本支持，如ZSTD压缩和INDEX编码仅在云Lindorm支持，你可以通过配置ZSTD和INDEX获得更好的性能。

```
# 配置被测表的压缩算法，可选项有：
# NONE LZO ZSTD SNAPPY GZ LZ4 ZSTD 等
# 注意部分被测系统可能不支持指定的压缩算法
# 云Lindorm推荐使用ZSTD
ahbench.table.compression=SNAPPY
# 配置被测表的编码算法，可选项有：
# NONE DIFF INDEX
# 注意部分被测系统可能不支持指定的编码算法
# 云Lindorm推荐使用INDEX
ahbench.table.encoding=DIFF
```

启动测试

- 快速测试集
  测试数据量1000万，整体运行时间大约40分钟（视被测系统可能有变化），至少需要总存储空间20 GB。

  ```
  cd AHBench
  ./fast_test
  ```

- 完整测试集
  测试数据量20亿，整体运行时间大约25小时（视被测系统可能有变化），至少需要总存储空间2 TB。

  ```
  cd AHBench
  ./full_test
  ```

  若要重复进行该测试，可以通过跳过数据导入阶段（上一次测试已成功运行），减少运行时间。跳过导入阶段，整体测试运行时间大约3.5小时（视被测系统可能有变化）。

  ```
  cd AHBench
  ./full_test --skipload
  ```

测试结果分析

测试完成后，会在当前目录生成CSV文件。CSV（Comma-Separated Values）文件全称为逗号分隔值文件。可以将测试结果复制到Excel、Numbers等数据分析软件中做进一步对比分析。

1. 查看CSV文件的名称。

   ```
   ls -ltr
   ```

2. 查看指定CSV文件的内容。

   ```
   cat full_throughput.csv
   ```

CSV文件内容如图所示。

```
hadoop@nake-ahbench-for-c5 ~/AHBench $ ls -ltr
total 52
drwxr-xr-x 14 hadoop hadoop 4096 Sep  5 10:23 workloads/
-rw-r--r--  1 hadoop hadoop 1421 Sep  5 10:23 README
-rw-r--r--  1 hadoop hadoop  558 Sep  5 10:23 LICENSE
-rwxr-xr-x  1 hadoop hadoop  741 Sep  5 10:23 full_test*
-rwxr-xr-x  1 hadoop hadoop  963 Sep  5 10:23 fast_test*
drwxr-xr-x  5 hadoop hadoop 4096 Sep  5 10:23 tools/
drwxr-xr-x  2 hadoop hadoop 4096 Sep  5 10:31 conf/
drwxrwxr-x  2 hadoop hadoop 4096 Sep  5 10:37 tmp/
drwxr-xr-x  2 hadoop hadoop 4096 Sep  5 10:43 bin/
drwxr-xr-x  2 hadoop hadoop 4096 Sep  5 10:43 suite/
-rw-rw-r--  1 hadoop hadoop  437 Sep  6 11:16 full_throughput.csv
-rw-rw-r--  1 hadoop hadoop  408 Sep  6 11:16 full_spike_latency.csv
drwxrwxr-x  2 hadoop hadoop 4096 Sep  6 11:43 logs/
hadoop@nake-ahbench-for-c5 ~/AHBench $ cat full_throughput.csv
TestName,OperationType,Throughput(rows/s),AverageLatency(us),P95Latency(us),P99Latency(us),P999Latency(us)
full_throughput_single_read,READ,68901.0,2898.706728054881,9255.0,46559.0,108863.0
full_throughput_scan,SCAN,1247157.0,4004.8719123340397,18639.0,47167.0,71103.0
full_throughput_single_write,INSERT,195834.0,1016.75785602,1344.0,3009.0,13295.0
full_throughput_batch_write,BATCH,782735.0,12722.031288233438,20623.0,87743.0,116095.0
hadoop@nake-ahbench-for-c5 ~/AHBench $ cat full_spike_latency.csv
TestName,OperationType,Throughput(rows/s),AverageLatency(us),P95Latency(us),P99Latency(us),P999Latency(us)
full_spike_single_read,READ,4996.0,656.0502980620339,974.0,2335.0,51775.0
full_spike_scan,SCAN,249864.0,1390.0287474438796,2061.0,8035.0,16655.0
full_spike_single_write,INSERT,49971.0,582.7455317893222,664.0,1112.0,5943.0
full_spike_batch_write,BATCH,199871.0,2798.396296975711,2945.0,14951.0,96319.0
```

常见问题

如果测试中遇到错误退出，请检查如下事项。

- JAVA_HOME是否正确设置，是否安装了Python运行环境。
- 被测集群地址是否填写正确。
- 被测集群HBase版本是否填写正确。
- 被测集群是否支持测试所指定的压缩算法。
- 被测集群状态是否正常服务。

7.6.3. 性能测试工具-图形工具

AHBench-jenkins是由阿里云HBase团队研发的benchmark测试套件。本文介绍AHBench-jenkins测试套件的使用和注意事项。

介绍

该测试套件集成了YCSB（Yahoo! Cloud Serving Benchmark）工具，YCSB提供了测试集合、测试流程控制、结果汇聚等功能。使用该测试套件，您可以通过简单配置，一键执行性能测试。

准备工作

- 在压测客户端部署在ECS，您需要确保云原生多模数据库Lindorm实例和ECS实例满足以下条件，以保证网络的连通性。ECS实例信息的查看方法请参见查看实例信息。
  - 所在地域相同，并建议所在可用区相同（以减少网络延时）。
  - 网络类型相同。

  > **说明**
  > - 专有网络具有更高的安全性，建议使用专有网络。
  > - 如果网络类型为**专有网络**，需确保使用相同的专有网络ID。

- 将压测客户端IP地址添加至云原生多模数据库Lindorm实例的白名单中，添加方法请参见设置白名单。
- 下载测试套件AHBench-jenkins，并上传到压测客户端并解压。

注意事项

- 测试压力过程中可能会使得被测实例负载过高，请勿在生产环境中运行该测试套件。
- 该测试套件同一时间只能运行一个测试集，测试集内部测试任务串行执行。请确保没有并行执行不同测试集的测试任务。

- 该测试套件使用"ahbenchtest-read"与"ahbenchtest-write" 两个表作为测试用表。测试过程中可能会删除再创建这两张表。请确保这两张表可以安全地被删除。
- 确保测试实例有足够的存储空间。
- ECS为虚拟运行环境，同一机型下的性能测试结果可能存在5%～10%的波动，处于预期范围之内。

运行环境

请确保压测客户端运行环境满足以下条件：

- Linux系统
- JDK 1.8+
- Python 2.7
- 建议客户端CPU配置为独享16Core以上

启动服务

启动AHBench-jenkins服务后，您可以通过在浏览器中输入目标地址（例如：HTTP://IP:PORT） 来访问服务。

```
# 通过如下命令可以设置该套件的端口，默认为8080
cd AHBench-jenkins
./start.sh 8080
```

选择测试集模板

测试集模板包括快速测试集（FastTest）和完整测试集（FullTest）。

- 快速测试集：测试数据量1000万，整体运行时间大约2小时（视被测实例规格可能有变化），至少需要总存储空间20 GB。
- 完整测试集：测试数据量20亿，整体运行时间大约25小时（视被测实例规格可能有变化），至少需要总存储空间2 TB。

配置测试任务

选择测试模板后，您可以单击左侧菜单栏**Build with Parameters**来配置测试任务。

配置实例地址

您可以通过cluster_addr参数配置实例地址。配置方式和路径如下表所示。

配置方式	cluster_addr参数说明	获取路径
HBase标准版	HBase标准版实例的ZK链接地址。	**控制台页面 > 数据库连接 > 连接信息**（注意区分专有网络和外网地址）。
HBase增强版	HBase增强版实例的Java API访问地址。	
Lindorm	Lindorm实例宽表引擎的HBase Java API访问地址。	**控制台页面 > 数据库连接 > 宽表引擎**（注意区分专有网络和外网地址）。

专有配置说明（可选）

不同配置方式中的参数说明如下表所示。

配置方式	参数说明
HBase标准版	cluster_version：HBase标准版实例版本。 如果为1.x版本则填1。如果为2.x版本则填2。默认值为2。
HBase增强版	• username：访问HBase增强版实例的数据库用户，请先对该账号进行授权。 • password：访问HBase增强版的数据库密码，请先对该账号进行授权。
Lindorm	• username：访问Lindorm实例的数据库用户，请先对该账号进行授权。 • password：访问Lindorm实例的数据库密码，请先对该账号进行授权。

通有配置说明（可选）

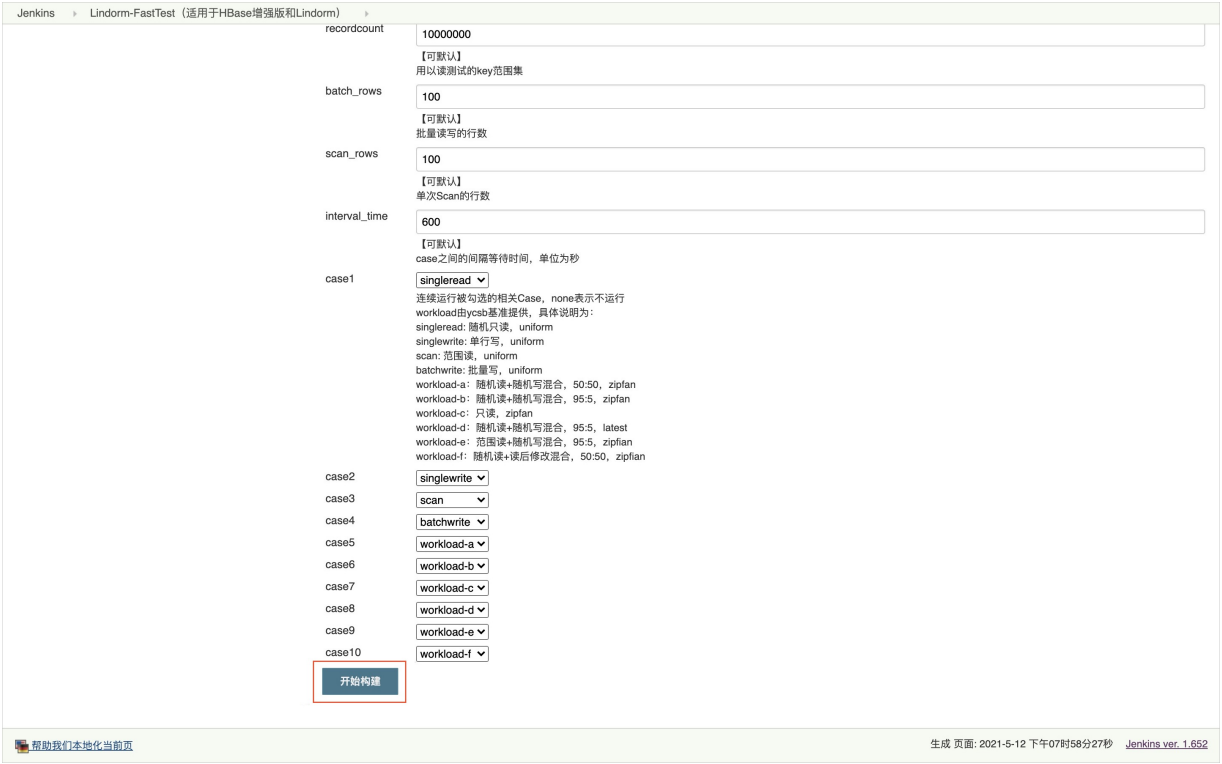
以下配置测试相关的参数，如压缩、编码、线程数、数据量、字段大小、测试用例等。默认不需要修改，如有需求可根据场景进行定制修改。

参数名称	参数说明
test_name	测试名称，只用作显示。
table_name	测试用表名称，用于测试过程中读写数据。
row_len	每行数据的主键长度。
col_num	每行数据中列的个数。
col_len	每个列的长度。
re_create_table	表示是否要重建新表。
compression	表的压缩算法。支持的压缩算法包括NONE、LZO、SNAPPY、GZ、LZ4和ZSTD。 **说明** ZSTD仅在HBase增强版和Lindorm中支持。
encoding	表的编码算法。支持的编码算法包括NONE、DIFF和INDEX。 **说明** INDEX仅在HBase增强版和Lindorm中支持。
regions	预分区数目。
pre_load	是否要预加载数据，只有预加载写入的数据才能进行读相关的测试。
load_params	预加载的相关参数，并发线程、限速（每秒目标操作数）和加载行数。
load_wait	预加载后的等待时间。
warm_up	是否在测试前进行预热读。
warm_up_params	预热的相关参数，并发线程、运行时间（秒）和读取范围。
threads	并发测试线程数。
exec_time	每个测试用例的运行时间。
recordcount	用以读测试的Key范围集。
batch_rows	批量读写的行数。
scan_rows	单次scan的行数。
interval_time	测试用例的间隔等待时间。
case1-case10	顺序运行的测试用例，可以根据需要选择要运行的用例。 • none：不运行该测试用例。 • singleread：随机只读。 • uniformsinglewrite：单行写。 • uniformscan：范围读，uniform。 • batchwrite：批量写，uniform。 • workload-a：随机读+随机写混合，50:50，zipfan。 • workload-b：随机读+随机写混合，95:5，zipfan。 • workload-c：只读，zipfan。 • workload-d：随机读+随机写混合，95:5，latest。 • workload-e：范围读+随机写混合，95:5，zipfian。 • workload-f：随机读+读后修改混合，50:50，zipfian。

> 说明
>
> - 部分参数仅在特定HBase版本支持，例如ZSTD压缩算法和INDEX编码算法仅在HBase增强版和Lindorm中支持，您可以通过配置ZSTD和INDEX获得更好的性能。
> - 若要重复进行该测试，您可以通过取消re_create_table和pre_load来跳过重新建表和数据导入阶段（上一次测试已成功运行）。跳过导入阶段，整体测试运行时间大约3.5小时（视被测实例规格可能有变化）。

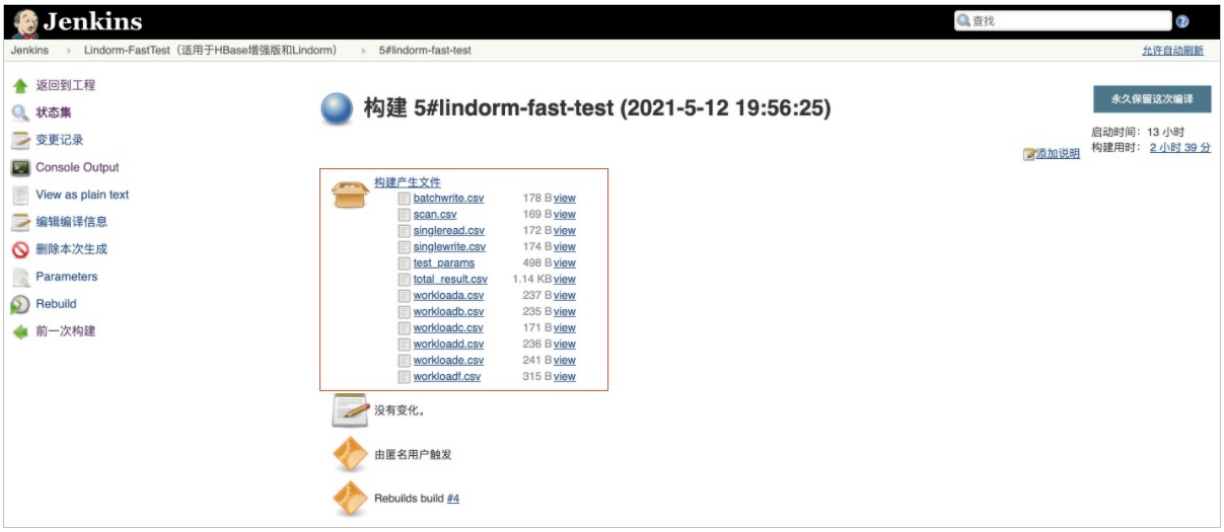
启动测试

单击**开始构建**按钮，如下图所示，提交测试任务到执行队列中。



测试结果分析

测试用例运行完毕后，会生成CSV文件，如下图所示，您可以通过浏览器进行查看和下载。



CSV（Comma-Separated Values）文件全称为逗号分隔值文件。您可以将数据结果复制到Excel或者Numbers等数据分析软件中做进一步对比分析。CSV文件结果如图所示。

```
TestName,OperationType,Throughput(rows/s),AverageLatency(us),P95Latency(us),P99Latency(us),P999Latency(us)
batchwrite,BATCH,101437.0,125750.6753771419,210431.0,254463.0,314111.0
scan,SCAN,1655724.0,7704.259769699419,21295.0,58655.0,85887.0
singleread,READ,75501.0,1689.6162362271912,2044.0,2669.0,22415.0
singlewrite,INSERT,42512.0,3001.485613855078,4483.0,9263.0,50271.0
workloada,READ,26010.0,1996.9924856520854,2789.0,4135.0,27535.0
workloada,UPDATE,26011.0,2907.7582520171654,3963.0,5891.0,31071.0
workloadb,READ,59728.0,1989.3114753677578,2771.0,3699.0,19631.0
workloadb,UPDATE,3150.0,2770.810318072335,4025.0,5311.0,22783.0
workloadc,READ,67388.0,1892.5555709970954,2555.0,3433.0,17103.0
workloadd,READ,61513.0,1926.9758923129186,2717.0,3705.0,15391.0
workloadd,INSERT,3237.0,2781.8499603085943,4055.0,5623.0,18815.0
workloade,INSERT,188.0,4851.740461394097,13071.0,48319.0,109183.0
workloade,SCAN,349277.0,26626.44060447983,68415.0,417535.0,463615.0
workloadf,READ,28806.0,2688.7845312080804,4383.0,10071.0,55967.0
workloadf,UPDATE,14402.0,3481.2656232319496,5551.0,8895.0,55455.0
workloadf,READ-MODIFY-WRITE,14402.0,6398.633906954003,9551.0,24639.0,66623.0
```

常见问题

如果测试中遇到错误退出，请检查如下事项：

- Java和Python是否安装，版本是否正确。
- 被测实例地址以及专用配置是否填写正确。
- 被测实例是否支持测试所指定的压缩算法。
- 被测实例状态是否正常服务。

7.6.4. 测试方法

本次测试包括吞吐量对比测试与毛刺率对比测试。
吞吐量测试使用相同的线程数分别对社区版HBase和云原生多模数据库Lindorm进行对比测试，对比其吞吐量。毛刺率测试使用相同的吞吐压力分别对社区版HBase和云原生多模数据库Lindorm进行对比测试，对比其毛刺率。压缩率测试使用相同的数据写入社区版HBase和云原生多模数据库Lindorm进行对比测试，对比其压缩比。

建表

在社区版HBase和云原生多模数据库Lindorm集群中分别建表。所有case的表构造都相同，根据ycsb数据进行预分区，建表时预分200个分区。

> **说明** 建表语句请使用HBase Shell执行，使用方法请参见通过HBaseue Shell连接并使用宽表引擎。

- 云原生多模数据库Lindorm使用独有的INDEX编码（当指定编码为DIFF时会自动升级为INDEX编码算法）和ZSTD压缩，建表语句为：

```
create 'test', {NAME => 'f', DATA_BLOCK_ENCODING => 'DIFF', COMPRESSION => 'ZSTD'}, {SPLITS => (1..199).ma
p{|i| "user#{(i * ((2**63-1)/199)).to_s.rjust(19, "0")}"} }
```

- 社区版HBase使用官方推荐的DIFF编码和SNAPPY压缩，建表语句为：

```
create 'test', {NAME => 'f', DATA_BLOCK_ENCODING => 'DIFF', COMPRESSION => 'SNAPPY'}, {SPLITS => (1..199).
map{|i| "user#{(i * ((2**63-1)/199)).to_s.rjust(19, "0")}"} }
```

数据准备

对于单行读和范围读场景，需要对读取的数据进行数据准备。

单表准备20亿行基础数据，每行数据20列，value为20byte。

ycsb的配置文件为：

```
recordcount=2000000000
operationcount=150000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=false
fieldcount=20
fieldlength=20
readproportion=1.0
updateproportion=0.0
scanproportion=0
insertproportion=0
requestdistribution=uniform
```

启动命令为：

```
bin/ycsb load hbase10 -P <workload> -p table=test -threads 200 -p columnfamily=f -s
```

吞吐量对比测试

吞吐量测试为在固定线程数条件下，对比社区版HBase与云原生多模数据库Lindorm本的吞吐能力差异。 一共有4个测试场景，每组测试场景均独立进行，测试场景之间没有前后依赖关系。

- 单行读场景
  20亿行基础数据，每行数据20列，value为20byte，查询区间为1000万行。 上述数据准备完成后，触发其major_compaction并等待完成。 先进行预热测试20分钟，再进行20分钟的正式测试。
  ycsb的workload配置如下：

  ```
  recordcount=10000000
  operationcount=2000000000
  workload=com.yahoo.ycsb.workloads.CoreWorkload
  readallfields=false
  fieldcount=1
  fieldlength=20
  readproportion=1.0
  updateproportion=0.0
  scanproportion=0
  insertproportion=0
  requestdistribution=uniform
  ```

  启动命令为：

  ```
  bin/ycsb run hbase10 -P <workload> -p table=test -threads 200 -p columnfamily=f -p maxexecutiontime=1200
  ```

- 范围读场景
  20亿行基础数据，每行数据20列，value为20byte。查询区间为1000万行，每次范围读取50行。 上述数据准备完成后，触发其major_compaction并等待完成。 先进行预热测试20分钟，再进行20分钟的正式测试。
  ycsb的workload配置如下：

```
recordcount=10000000
operationcount=2000000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=false
fieldcount=1
fieldlength=20
readproportion=0.0
updateproportion=0.0
scanproportion=1.0
insertproportion=0
requestdistribution=uniform
maxscanlength=50
Lindorm.usepagefilter=false
```

启动命令为：

```
bin/ycsb run hbase10 -P <workload> -p table=test -threads 100 -p columnfamily=f -p maxexecutiontime=1200
```

- 单行写场景
  每次插入1列，value为20byte。执行20分钟测试。
  ycsb的workload配置如下：

```
recordcount=2000000000
operationcount=100000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=false
fieldcount=1
fieldlength=20
readproportion=0.0
updateproportion=0.0
scanproportion=0
insertproportion=1.0
requestdistribution=uniform
```

启动命令为：

```
bin/ycsb run hbase10 -P <workload> -p table=test -threads 200 -p columnfamily=f -p maxexecutiontime=1200
```

- 批量写场景
  每次插入1列，value为20byte，每次batch写入100行。执行20分钟测试：

```
recordcount=2000000000
operationcount=10000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
fieldcount=1
fieldlength=20
cyclickey=true
readallfields=false
readproportion=0
updateproportion=0
scanproportion=0
insertproportion=0.0
batchproportion=1.0
batchsize=100
requestdistribution=uniform
```

启动命令为：

```
bin/ycsb run hbase10 -P <workload> -p table=test -threads 100  -p columnfamily=f -p maxexecutiontime=1200
```

毛刺率对比测试

毛刺率对比测试为在固定OPS（Operation per second）的条件下，对比社区版HBase与云原生多模数据库Lindorm本的毛刺差别。

- 单行读场景
  20亿行基础数据，每行数据20列，value为20byte，查询区间为1000万行。 OPS限制为5000 上述数据准备完成后，触发其major_compaction并等待完成。 先进行预热测试20分钟，再进行20分钟的正式测试。
  ycsb的workload配置如下：

```
recordcount=10000000
operationcount=2000000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=false
fieldcount=1
fieldlength=20
readproportion=1.0
updateproportion=0.0
scanproportion=0
insertproportion=0
requestdistribution=uniform
```

  启动命令为：

```
bin/ycsb run hbase10 -P <workload> -p table=test -threads 200 -p columnfamily=f -p maxexecutiontime=1200 -
p target=5000
```

- 范围读场景
  20亿行基础数据，每行数据20列，value为20byte，查询区间为1000万行，每次范围读取50行。 OPS限制为5000 上述数据准备完成后，触发其major_compaction并等待完成。 先进行预热测试20分钟，再进行20分钟的正式测试。
  ycsb的workload配置如下：

```
recordcount=10000000
operationcount=2000000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=false
fieldcount=1
fieldlength=20
readproportion=0.0
updateproportion=0.0
scanproportion=1.0
insertproportion=0
requestdistribution=uniform
maxscanlength=50
Lindorm.usepagefilter=false
```

  启动命令为：

```
bin/ycsb run hbase10 -P <workload> -p table=test -threads 100 -p columnfamily=f -p maxexecutiontime=1200 -
p target=5000
```

- 单行写场景
  每次插入1列，value为20byte。执行20分钟测试 OPS限制为50000。
  ycsb的workload配置如下：

```
recordcount=2000000000
operationcount=100000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=false
fieldcount=1
fieldlength=20
readproportion=0.0
updateproportion=0.0
scanproportion=0
insertproportion=1.0
requestdistribution=uniform
```

启动命令为：

```
bin/ycsb run hbase10 -P <workload> -p table=testwrite -threads 200 -p columnfamily=f -p maxexecutiontime=1
200 -p target=50000
```

- 批量写场景
  每次插入1列，value为20byte，每次batch写入100行。执行20分钟测试： ops限制为2000。

```
recordcount=2000000000
operationcount=10000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
fieldcount=1
fieldlength=20
cyclickey=true
readallfields=false
readproportion=0
updateproportion=0
scanproportion=0
insertproportion=0.0
batchproportion=1.0
batchsize=100
requestdistribution=uniform
```

启动命令为：

```
bin/ycsb run hbase10 -P <workload> -p table=testwrite -threads 100 -p columnfamily=f -p maxexecutiontime=1
200 -p target=2000
```

压缩率对比测试

以下所有压缩率对比测试均使用同一测试流程。 通过ycsb写入500万行测试数据，手动触发flush与major_compaction并等待完成，统计其表大小。

每行列个数	每列value大小
1	10
1	100
20	10
20	20

ycsb的workload配置如下：

```
recordcount=5000000
operationcount=150000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=false
fieldcount=<每行列个数>
fieldlength=<每列value大小>
readproportion=1.0
requestdistribution=uniform
```

写入数据命令为：

```
bin/ycsb load hbase10 -P <workload> -p table=test -threads 200 -p columnfamily=f -s
```

7.6.5. 测试结果分析

本文介绍云原生多模数据库Lindorm与开源HBase的吞吐量对比、毛刺率对比和压缩率对比的测试结果。

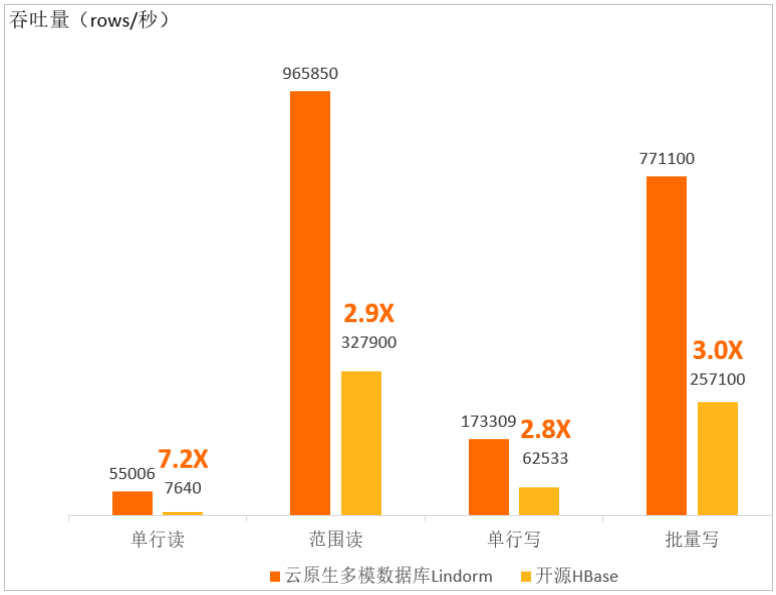
前提条件

基于以下环境配置，性能测试工具和测试方法分析本文的测试结果。

- 环境配置的详情请参见测试环境。
- 性能测试工具的使用请参见性能测试工具。
- 测试方法请参见测试方法。

吞吐量对比测试结果

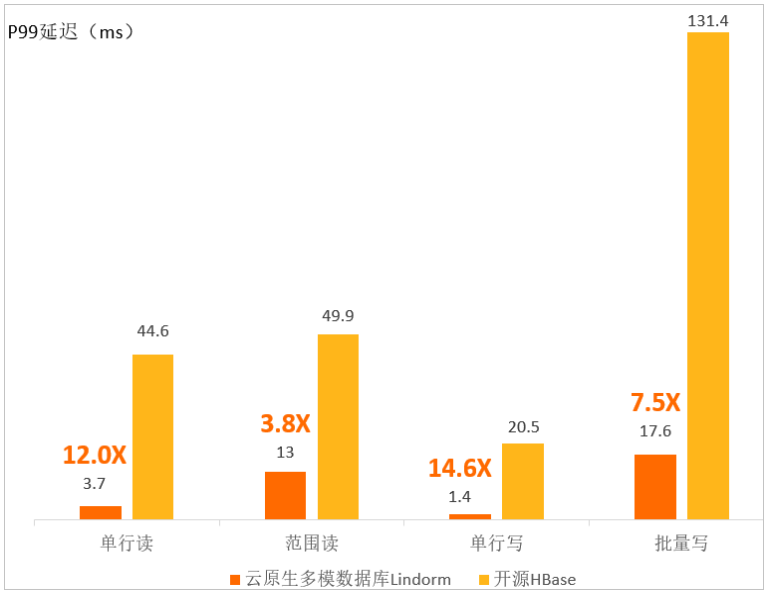
相同规格下，20亿行数据量的吞吐量对比如下：



> **说明** 图中数值仅供参考，具体以实际测试结果为准。

毛刺率对比测试结果

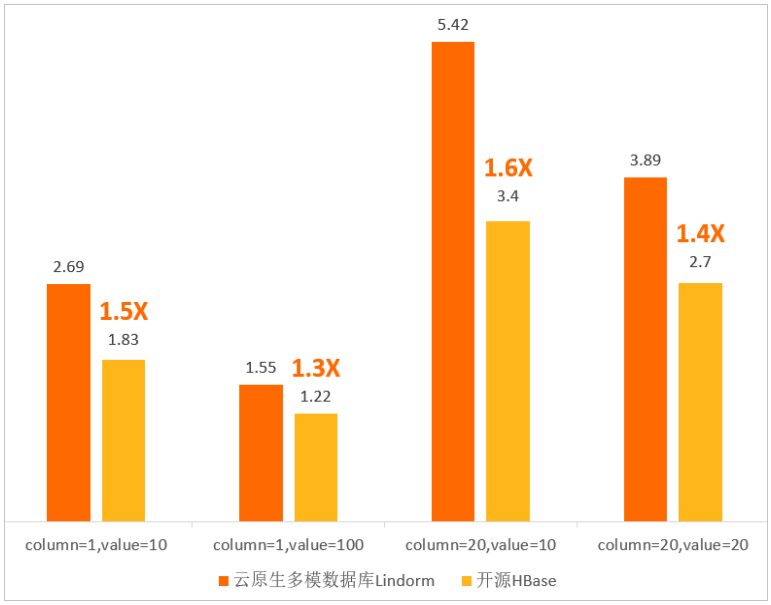
相同规格下，20亿行数据量的毛刺率对比如下：



> **说明**
> - P99延迟表示99%请求的响应时间小于该值。
> - 图中数值仅供参考，具体以实际测试结果为准。

压缩率对比测试结果

不同场景中随机数据的压缩率对比如下：



7.7. 集群管理系统

7.7.1. 登录集群管理系统

通过集群管理系统可以查看集群的空间使用情况，还可以对集群中的数据进行管理。本文介绍如何通过云原生多模数据库Lindorm控制台登录集群管理系统。

前提条件

- 已创建云原生多模数据库Lindorm实例，具体操作请参见创建实例。

- 已将访问集群管理系统的IP地址添加至Lindorm实例的白名单中，具体操作请参见设置白名单。

操作步骤

1. 登录Lindorm管理控制台。
2. 在**实例列表**页面中单击目标实例ID。
3. 在左侧导航栏中，选择**宽表引擎 > 集群管理**。
4. 在**UI访问**区域中单击**ClusterManager**进入集群管理系统。

> **说明** 如果您是第一次访问集群管理系统，请单击**UI访问**区域中右上方的**重置UI访问密码**，设置集群管理系统的访问用户名和密码。设置规则如下：
>
> - 长度为2~30位。
> - 需包含大写字母、小写字母、特殊字符和数字中的一种或者多种。
> - 允许的特殊字符包括下划线（_）和短划线（-）。

集群管理系统介绍

集群管理系统包括信息面板、数据管理和数据查询。详情参见下表。

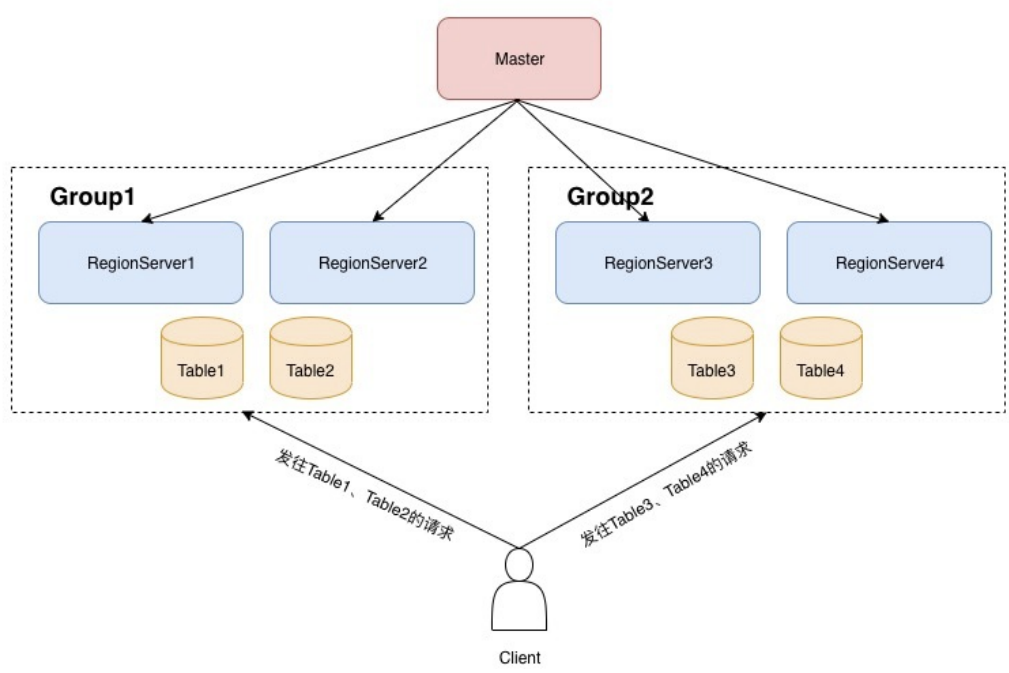
主页面	子页面	页面描述
信息面板	集群概览	查看集群的基本信息、当前的存储空间和当前的分组概览信息。
	节点详情	查看集群分组中节点的基本信息以及当前节点的分片详情。
数据管理	Namespace管理	查看集群中所有的命名空间，并对命名空间进行管理。
	表管理	查看集群中所有的表和表属性。单击表名可以查看表的详细信息。
	用户管理	查看集群中的用户和用户拥有的权限，并对用户进行管理。
	ACL管理	可以管理用户所拥有的权限，赋予和回收指定用户的一项或多项权限。
数据查询	不涉及	使用SQL语句查询Lindorm宽表的数据。

7.7.2. 分组管理

云原生多模数据库Lindorm支持的分组功能可以用来解决多租户隔离问题，通过划分RegionServer到不同的分组，每个分组上存储不同的表，从而达到资源隔离的目的。本文介绍云原生多模数据库Lindorm通过集群管理系统管理分组。

背景信息

当多个用户或者多个业务同时使用一个Lindorm集群时，往往存在资源争抢的问题，一些重要的在线业务读写操作可能会被离线业务批量读写所影响，可以使用分组功能解决这一问题。例如，如下图所示，创建Group1并把RegionServer1和RegionServer2划分到Group1中，创建Group2并把RegionServer3和RegionServer4划分到Group2中。同时把Table1和Table2也移动到Group1分组，Table1和Table2的所有Region都只会分配到Group1中的RegionServer1和RegionServer2机器上。同理，Table3和Table4的所有Region都只会分配到Group2中的RegionServer3和RegionServer4机器上。所以您发送到Table1和Table2的请求，只会由RegionServer1和RegionServer2服务响应，而发送到Table3和Table4的请求，只会由RegionServer3和RegionServer4服务响应，从而达到资源隔离的目的。



前提条件

已登录目标实例的集群管理系统，具体操作请参见登录集群管理系统。

查看分组信息

在集群管理系统的左侧导航栏，选择**信息面板**，在**集群分组**区域中可以看到当前集群所有的分组信息。如果您没有创建过分组，系统会默认创建一个分组，该分组名称为**default**，所有的RegionServer和表都会归属到这个分组中。



创建分组

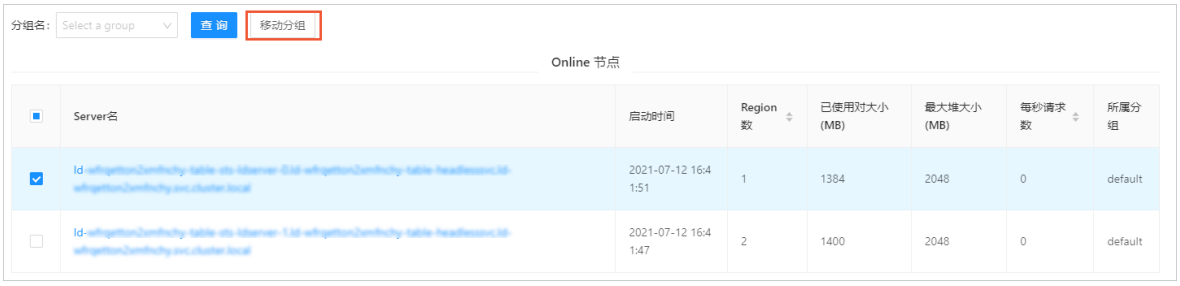
1. 在集群管理系统的左侧导航栏，选择**信息面板**。
2. 在**集群分组**区域中，单击右方的**更多操作 > 新建分组**。
3. 在**新建分组**对话框中输入**分组名**。
4. 单击**确定**。

> **说明** 创建分组后，分组中的Server数量和Table数量都为0，需要您将Server和Table移动到该分组中。

移动RegionServer的分组

默认状态下，所有的RegionServer都属于default分组，您需要将RegionServer移动到对应的分组中才能使用。

1. 在集群管理系统的左侧导航栏，选择**信息面板**。
2. 在**集群分组**区域中，勾选目标RegionServer。
3. 单击**移动分组**。



4. 在**移动RegionServer分组**对话框中选择target group。
5. 单击**确定**。

> **说明**
> - 如果您将Table移动到一个没有RegionServer的分组中，那么该表的Region会由于没有服务器可以上线而无法访问。
> - 建议每个分组至少有两台RegionServer机器，当一台RegionServer机器无法运行时，表的Region可以分配到同一分组的另一台RegionServer机器上。如果只有一台RegionServer机器，当这台RegionServer机器无法运行时，这个分组所有的表都无法被访问。
> - 在移动RegionServer的分组时，这个RegionServer上正在访问的Region会被立即重新分配到分组的其他RegionServer机器上。

删除分组

> **注意** 只有分组中的所有Server和Table都被移出（也就是Server数和Table数为0），分组才能被删除。

1. 在集群管理系统的左侧导航栏，选择**信息面板**。
2. 在**集群分组**区域，单击目标分组中**Action**列的**删除**。
3. 在对话框中单击**Yes**。

设置Namesapce所属分组

如果您没有设置Namespace所属分组，在这个Namesapce中新建的表默认都会属于default分组。如果您不想每次在新建表后都手动将表移动到对应的分组中，可以通过设置Namespace所属分组将该Namespace下所创建的新表自动加入至对应的分组中。

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > Namespace管理**。
2. 在**namespace**页面，单击目标Namespace中**Action**列的**绑定group**。
3. 在**bind group**对话框中选择**group name**。
4. 单击**确定**。

> **说明**
> - 修改Namespace分组的所属分组信息不会影响该Namespace中已有的表，已有表的分组不会被改变。
> - 如果Namespace有所属分组，新建的表会自动属于该分组。您也可以手动将这个表移动到另一个分组中。

移动表的分组

请勿将表移动到没有服务器的分组，否则会造成表无法被访问。

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > 表管理**。

2. 在**用户表**页面，勾选目标表名，单击**移动分组**。
3. 在**移动表分组**对话框中输入target group。
4. 单击**确定**。

7.7.3. Namespace管理

云原生多模数据库Lindorm中的Namespace是表的一个逻辑分组。不同的Namespace中的表允许重名。Lindorm会默认提供一个名称为default的Namespace，如果您没有指定使用Namespace，则会默认使用default Namespace。本文介绍通过集群管理系统管理Namespace。

前提条件

已登录目标实例的集群管理系统，具体操作请参见登录集群管理系统。

创建Namespace

1. 在集群管理系统的左侧导航栏中选择**数据管理 > Namespace管理**。
2. 在**namespace**页面，选择**更多操作 > 新建namespace**。
3. 在**Create namespace**对话框中输入**Namespace name**。
4. 单击**确定**。

删除Namespace

> 注意　删除Namespace前请先将Namespace中所有的表都删除，否则删除操作会失败。

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > Namespace管理**。
2. 在**namespace**页面，单击目标Namespace中**Action**列的**删除**。
3. 在弹出对话框中单击**删除**。

设置Namespace所属分组

设置Namespace所属分组可以将该Namespace下所创建的新表自动加入至对应的分组中，具体操作请参见设置Namesapce所属分组。

7.7.4. 表管理

通过集群管理系统可以查看云原生多模数据库Lindorm实例中所有表的详细信息和所有Region信息，本文介绍如何通过集群管理系统管理表。

前提条件

已登录目标实例的集群管理系统，具体操作请参见登录集群管理系统。

查询表的详细信息

1. 在集群管理系统的左侧导航栏中选择**数据管理 > 表管理**。
2. 在**用户表**区域，可以根据表的分组名、Namespace、表类型、表状态或者表名过滤目标表。
3. 单击**查询**。
4. 单击目标表的表名，可以查看表属性、Table Region等详情信息。



移动表分组

请勿将表移动到没有服务器的分组，否则会造成表无法被访问。具体操作请参见移动表的分组。

7.7.5. 用户和权限管理

7.7.5.1. 管理用户

云原生多模数据库Lindorm支持用户认证功能。用户的认证只需要填写用户名和密码，用户的密码在服务器端非明文存储，并且在认证过程中不会明文传输密码，即使在验证过程中密文被拦截，用户认证的内容不可重复使用，无法被伪造。本文介绍通过集群管理系统管理用户。

前提条件

已登录目标实例的集群管理系统，具体操作请参见登录集群管理系统。

创建用户

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > 用户管理**。
2. 在**用户列表**页面，选择**更多操作 > create user**。
3. 在**新建用户**对话框中输入**用户名**、**密码**和**确认密码**。
4. 单击**确定**。

> 说明
> - 创建用户后无法查看该用户的密码，如果您忘记密码，只能通过修改密码的方式来重置密码。
> - 新建的用户没有任何权限，需要在ACL管理权限页面赋予权限后，才能正常访问。具体操作请参见管理ACL权限。

修改用户密码

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > 用户管理**。
2. 在**用户列表**页面，单击目标用户**Action**列的**修改密码**。
3. 在**修改密码**对话框中输入**新密码**和**确认密码**。
4. 单击**确定**。

删除用户

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > 用户管理**。
2. 在**用户列表**页面，单击目标用户**Action**列的**删除**。
3. 在对话框中单击**Yes**。

7.7.5.2. 管理ACL权限

本文介绍通过云原生多模数据库Lindorm实例的集群管理系统来管理用户的ACL权限。

权限层级

目前对Lindorm实例的操作有三个权限层级，由高到低为Global，Namespace和Table，这三者是相互覆盖的关系。

例如：如果为user1用户赋予Global的读写权限，那么user1用户拥有Lindorm宽表的所有Namespace中所有Table的读写权限。如果为user2用户赋予Namespace1的读写权限，那么user2用户拥有Namespace1中所有Table的读写权限（包括Namespace1中新建的Table）。

权限分类

操作权限分类和操作语法说明如下表。

分类	描述	操作语法
WRITE	写Lindorm表相关的操作。	PUT、BATCH、DELETE、INCREMENT、APPEND、CHECKANDMUTATE

分类	描述	操作语法
READ	读Lindorm表相关的操作。	GET、SCAN、EXIST
	读取Lindorm表descriptor，namespace列表等相关操作。	GETTABLEDESCRIPTOR、LISTTABLES、LISTNAMESPACEDESCRIPTORS
ADMIN	不会涉及表删除和表数据删除的DDL操作。	CREATETABLE、ENABLETABLE、DISABLETABLE
	Namespace相关DDL操作。	CREATENAMESPACE
TRASH	防止表误删和表数据被清空操作。	TRUNCATETABLE、DELETETABLE
SYSTEM	执行运维操作。使用BDS迁移和同步Lindorm时需要此权限。	COMPACT、FLUSH

为指定用户赋予权限

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > ACL管理**。
2. 在**Global权限**区域中，选择**更多操作 > grant privileges**。
3. **grant global privilege**对话框中，从**user name**列表中选择目标用户名。
4. 在**grant privileges**选项中，选中为该用户赋予的权限。
5. 单击**确定**。

为指定Namespace赋予权限

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > ACL管理**。
2. 在**Namespace权限**区域中，选择**更多操作 > grant privileges**。
3. **grant namespace privilege**对话框中，从**user name**列表中选择目标用户名。从**grant namespace**列表中选择该用户中的Namespace。
4. 在**grant privileges**选项中，选中为指定的Namespace赋予的权限。
5. 单击**确定**。

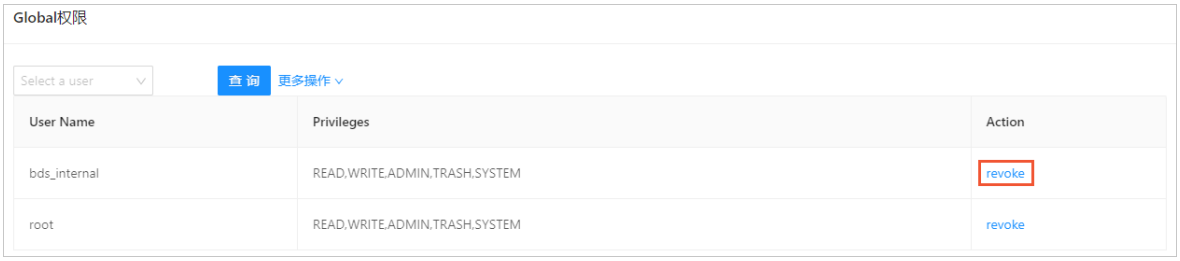
为指定Table赋予权限

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > ACL管理**。
2. 在**Table权限**区域中，选择**更多操作 > grant privileges**。
3. **grant table privilege**对话框中，从**user name**列表中选择目标用户名。从**grant namespace**列表中选择该用户中的Namespace。从**grant table**列表中选择该Namespace中指定的表。
4. 在**grant privileges**选项中，选中为目标表格赋予的权限。
5. 单击**确定**。

收回权限

集群管理系统的ACL权限管理中，可以收回指定用户、指定Namespace或者指定Table的权限。每个用户可能有多个层级的权限。以回收指定用户的权限为例，具体操作如下：

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > ACL管理**。
2. 在**Global权限**区域中，单击目标用户**Action**列的**revoke**。



3. 在**revoke privilege**对话框中选中需要回收的权限。

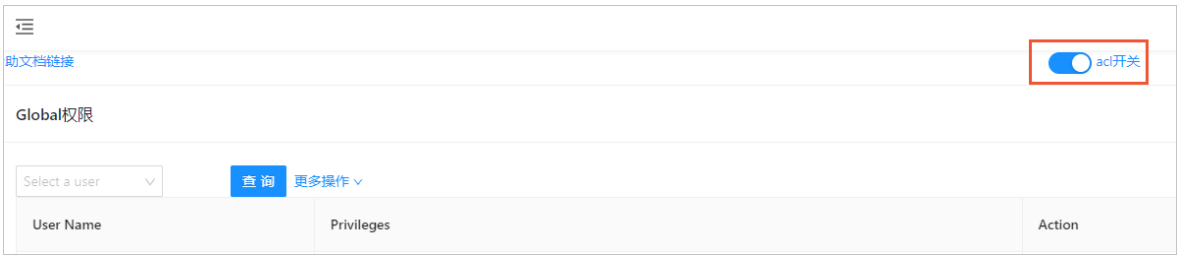
> **说明** revoke privileges对话框中会列出目标用户在层级（Global，Namespace和Table）中的所有权限。

4. 单击**确定**。

打开或关闭ACL功能

> **说明** 如果您不需要ACL功能来进行访问限制，可以将ACL功能关闭。ACL功能关闭后所有访问（包括API访问，SQL访问和非Java访问）都不需要用户名和密码。ACL功能可以动态地打开和关闭，无需重启集群。如果ACL是关闭状态，动态打开后没有提供用户名和密码的客户端在重连时，会因为无法认证而报错。提供了用户名和密码的客户端在重连时会正常认证。

1. 在集群管理系统中，左侧导航栏中选择**数据管理 > ACL管理**。
2. 在页面右上角，打开或关闭**acl开关**。



3. 在对话框中单击**yes**。

7.7.6. 数据查询

Lindorm宽表引擎在集群管理系统中提供了简单的SQL查询入口，您可以使用SQL语法来查询Lindorm宽表数据。本节介绍通过集群管理系统查询表的数据。

使用限制

- 集群管理系统只支持 `SELECT` 语句。
- 为确保数据安全，本系统每次查询最多返回100条数据。
- 对varbinary类型的字段进行条件查询时，必须使用HexCode编码的字符串作为Value。
- `ROW` 字段对应Lindorm的RowKey，`ROW` 和 `qualifier` 都是varbinary类型，`qualifier` 如果不属于family，需要指定family。
- `ROW` 和 `COL` 是SQL保留字段，查询时需要加反引号（`）。`qualifier` 指定family时也需要加反引号（`）。

查询表数据

1. 在集群管理系统的左侧导航栏，选择**数据查询**。
2. 在**SQL方式查询**区域中，选择**Namespace**。
3. 在右侧**Namespace下的表**选择目标表名。
4. 在编辑器中输入SQL语句。
5. 单击**执行**。



> **说明** 执行操作可以通过键盘完成。Windows系统：CTRL+ENTER。macOS系统：COMMAND+RETURN。

获取HexCode编码字符

对varbinary类型的字段进行条件查询时，必须使用HexCode编码的字符串作为Value。例如：查询rowkey为r1的数据，SQL查询语句中的 `WHERE` 应该写成 `WHERE rowkey='7321'`（r1的HexCode编码字符为7321）。为了方便您获取HexCode编码字符，集群管理系统的数据查询页面提供了转换工具。

1. 在集群管理系统的左侧导航栏，选择**数据查询**。
2. 在**SQL方式查询**区域中，选择**Namespace**。
3. 在**Binary-Hex转换说明**中输入Binary编码字符串。
4. 在SQL编辑器中最上面查看对应的HexCode编码字符。
5. 使用HexCode编码字符编写SQL语句。



6. （可选）勾选**HexString**可以查看HexCode编码字符的查询结果。
7. 单击**执行**。

7.8. 开发指南

7.8.1. Lindorm宽表SQL语法手册

7.8.1.1. 名词解释

本文介绍Lindorm宽表SQL语法的常见术语。

名词	对应Lindorm中的含义	描述
Schema	NameSpace	Schema逻辑包含很多表，类似数据库的概念。
Table	Table	表，构建表时需要定义表的schema结构，包括表的列名和列类型。
PRIMARY KEY	PRIMARY KEY	主键，用于标识一行数据，可以唯一确定一行数据的分布位置。
HASH Table	HASH Table	通过设置表的USE_HASH属性，使该表的主键具备随机打散能力，预防热点请求，但该属性对于顺序scan不支持。
NUMREGIONS	NUMREGIONS	Lindorm中表示数据分区数，通过设置表的属性进行配置，建表预分区中设置的属性之一。
STARTKEY和ENDKEY	STARTKEY和ENDKEY	Lindorm中表示数据分区范围，通过设置表的属性进行配置，建表预分区中设置的属性之一，可以不指定。
Mutability属性	Mutability属性	Lindorm中构建索引时需要设置的属性，包括四种：immutable、immutable_rows、mutable_latest、mutable_all。
动态列	动态列	Lindorm允许用户不预先定义列的情况下插入列。

7.8.1.2. 数据类型

Lindorm宽表SQL是一种类型化语言，支持丰富的数据类型集。本文介绍Lindorm宽表SQL支持的数据类型。

使用限制

- 单个主键列最大长度2 KB。
- 所有主键列长度之和不超过30 KB。
- 单个非主键列最大长度不超过2 MB。

基础数据类型

数据类型	对应的Java类型	描述
BOOLEAN	java.lang.Boolean	长度为1字节。定长精确数值类型。值域取决于长度，以及是否为无符号。0表示false，1表示true。
TINYINT	java.lang.Byte	长度为1字节。定长精确数值类型。值域取决于长度，以及是否为无符号。
SMALLINT	java.lang.Short	长度为2字节。定长精确数值类型。值域取决于长度，以及是否为无符号。
INTEGER	java.lang.Integer	长度为4字节。定长精确数值类型。值域取决于长度，以及是否为无符号。
BIGINT	java.lang.Long	长度为8字节。定长精确数值类型。值域取决于长度，以及是否为无符号。
FLOAT	java.lang.Float	长度为4字节。定长非精确数值类型。值域和精度取决于长度、precision和scale，以及是否为无符号。
DOUBLE	java.lang.Double	长度为8字节。定长非精确数值类型。值域和精度取决于长度、precision和scale，以及是否为无符号。走sql拼写方式按照科学计数法进行拼写double数据
DECIMAL(precision，scale)	java.lang.BigDecimal	变长二进制类型。decimal为十进制，消耗的字节数随精度增加而增加，通常用于存储金额等高精度数据，对于精度要求不高的场景（如监控），可以使用float或double。定义类型时需要指定precision和scale。 • precision：最大一共有多少位，值域是[1，38]。 • scale：小数点后最大有多少位，值域是[0，precision]。
VARCHAR / CHAR(N)	java.lang.String	字符串，支持中文。 • VARCHAR 变长字符串类型。 • CHAR 定长字符串类型，需要指定长度N。
BINARY(N)	byte[]	定长二进制类型，N表示字节数。写入数据时，包括两种情况。 • 若给出的值不足N字节，系统会自动在末尾将缺的字节用0补充。 • 若超出N字节，则自动截断。
VARBINARY	byte[]	变长二进制类型，作为主键时只能是最后一列主键。
DATE	java.sql.Types#DATE	日期时间（仅存储日期，不存储时间，在时区转换过程中容易出现日期不对的现象，不推荐使用）。
TIME	java.sql.Types#TIME	时间类型。
TIMESTAMP	java.sql.Types#TIMESTAMP	时间戳。

不支持的数据类型

数组等复杂类型。

7.8.1.3. 函数说明

7.8.1.3.1. Count函数

Count函数用于计算给定列的非空值，可以对查询返回的行进行计数。

示例

```
select count(*) from test;
select count(*) from test where c1 = 1;
select count(c1) from test where c1 = 1;
```

7.8.1.3.2. Sum函数

Sum函数用于求和，计算指定列的查询返回值之和。

示例

```
select sum(c1) from test;
```

7.8.1.3.3. Avg函数

Avg函数用于计算指定列查询返回值的平均值。

示例

```
//计算test表格中c1列的平均值。
SELECT avg(c1) FROM test;
```

7.8.1.3.4. Max函数和Min函数

Max函数和Min函数用于计算指定列的查询返回的最大值和最小值。

示例

```
SELECT max(c1) FROM test;
SELECT min(c1) FROM test;
```

7.8.1.3.5. As函数

As是将请求的结果以别名方式命名。

示例

```
select count(*) as countRow from test;
select c1 as testC from test where c1 = '2';
```

7.8.1.4. 使用限制说明

Lindorm宽表SQL语法包括DDL（Data Definition Language）数据定义语言、DML（Data Manipulation Language）数据操作语言和DCL（Data Control Language）数据控制语言三种。本文介绍Lindorm宽表SQL语法的使用限制和关键字使用说明。

使用限制

- 不支持关系运算符：相加、减等。
- 不支持事务功能。
- 不支持HAVING、UDF、VIEW等操作。
- 不支持子查询和嵌套查询等复杂操作。

- 目前只支持Aggregation相关的 SUM, AVG,MIN,MAX这几个函数，更多的函数支持补充中
- 有限支持ORDER BY，参见文档Lindorm SQL Order By介绍
- 有限支持GROUP BY、DISTINCT、JOIN等功能。由于Lindorm使用的是全内存计算，如果数据量过大，会导致集群Full GC或者OOM。使用前请评估数据量级和集群配置是否能满足。
- 表名和列名区分大小写。

> **说明** Lindorm宽表SQL详细的使用功能请参见Lindorm宽表SQL语法手册。

关键字使用说明

如果Lindorm宽表SQL语法中使用如下关键字，请添加反引号（`）进行转义，关键字不区分大小写。

ADD ADMIN_USER ADMIN_TRASH ALL ALLOW ALTER ANALYZE AND ANY ARRAY AS ASC ASYNC AT BERNOULLI BETWEEN BUILD BY CALL CALLED CASCADE CASE CAST CATALOGS COLUMN COLUMNS COMMENT COMMIT COMMITTED CONSTRAINT CREATE SEARCH SEARCH_QUERY INDEX INCLUDE CROSS CUBE CURRENT CURRENT_DATE CURRENT_ROLE CURRENT_TIME CURRENT_TIMESTAMP CURRENT_USER DATA DATABASE DATABASES DATE DAY DEALLOCATE DEFINER DELETE DESC DESCRIBE DETERMINISTIC DISTINCT DISTRIBUTED DROP DUPLICATE ELSE END ESCAPE EXCEPT EXCLUDING EXECUTE EXISTS EXPLAIN EXTRACT EXTERNAL FALSE FILTER FILTERING FIRST FOLLOWING FOR FORCE FORMAT FROM FULL FUNCTION FUNCTIONS GLOBAL GRANT GRANTED GRANTS GRAPHVIZ GROUP GROUPING HAVING HOUR IF IDENTIFIED IGNORE IN INCLUDING INNER INPUT INSERT INTERSECT INTERVAL INTO INVOKER IO IS ISOLATION JSON JOIN KEY LANGUAGE LAST LATERAL LEFT LEVEL LIKE LIMIT LOCALTIME LOCALTIMESTAMP LOGICAL MAP MINUTE MONTH NAME NAMESPACE NAMESPACES NATURAL NFC NFD NFKC NFKD NO NONE NORMAL_USER NORMALIZE NOT NULL NULLIF NULLS OLDPASSWORD ON ONLY ONLINE OFFLINE OFFSET OPTION OR ORDER ORDINARILY OUTER OUTPUT OVER PARTITION PARTITIONS PASSWORD POSITION PRECEDING PREPARE PRIMARY PRIVILEGES PROPERTIES RANGE READ RECURSIVE RENAME REPEATABLE REPLACE RESET RESPECT RESTRICT RETURN RETURNS REVOKE RIGHT ROLE ROLES ROLLBACK ROLLUP ROW ROWS SCHEMA SCHEMAS SECOND SECURITY SELECT SERIALIZABLE SESSION SET SETS SHOW SOME SQL START STATS SUBSTRING SYSTEM TABLE TABLES TABLESAMPLE TEXT THEN TIME TIMESTAMP TO TRASH TRANSACTION TRUE TRY_CAST TRUNCATE TYPE UESCAPE UNBOUNDED UNCOMMITTED UNION UNNEST UPDATE UPSERT USE USER USERS USING VALIDATE VALUES VERBOSE VIEW WHEN WHERE WITH WORK WRITE YEAR ZONE

7.8.1.5. DDL

7.8.1.5.1. 概览

本节介绍Lindorm宽表SQL语法的DDL语句。

语句	描述	相关文档
SHOW	查看SCHEMA或表信息。	SHOW
USE	使用某个SCHEMA。	USE
CREATE SCHEMA	创建SCHEMA。	CREATE DATABASE
DROP SCHEMA	删除SCHEMA。	DROP DATABASE
CREATE TABLE	创建Lindorm宽表。	CREATE TABLE
ALTER TABLE	增加表的列或修改表属性。	ALTER TABLE
DROP TABLE	删除Lindorm宽表。	DROP TABLE
CREATE INDEX	创建表的二级索引。	CREATE INDEX
ALTER INDEX	修改表的二级索引属性。	ALTER INDEX
SHOW INDEX	查看表的二级索引。	SHOW INDEX
DROP INDEX	删除表的二级索引。	DROP INDEX
CREATE SEARCH INDEX	创建表的搜索索引。	CREATE SEARCH INDEX

语句	描述	相关文档
DROP SEARCH INDEX	删除表的搜索索引。	DROP SEARCH INDEX
ALTER SEARCH INDEX	修改表的搜索索引，包括新增索引列，删除索引列和修改搜索索引属性。	ALTER SEARCH INDEX
SHOW SEARCH INDEX	查看表的搜索索引。	SHOW SEARCH INDEX

7.8.1.5.2. DESCRIBE/SHOW/USE

可以展示DataBase（Schema）、表（Table）和索引（Index），以及使用具体的schema。

示例

- 以下语句可以展示所有的database。

  ```
  show schemas;
  ```

  2.2.16版本后默认以下述语句：

  ```
  show databases;
  ```

- 在对应的Schema下，以下语句可以展示所有Table，默认Default Schema下的表格。

  ```
  show tables;
  ```

- 2.2.16版本后查看表的详细信息，包括列信息，列类型。

  ```
  show columns in/from 表名;
  ```

- 以下语句可以展示test表下的idx1索引。

  ```
  show index from test;
  ```

- 以下语句也可以展示test表详细的信息，包括列信息，列类型。

  ```
  describe test;
  ```

- 当我们需要使用某个特定的schema的时候，可以使用以下语句。

  ```
  use schema;
  ```

7.8.1.5.3. CREATE DATABASE

创建Database。

语法

```
create_database_statement ::=  CREATE DATABASE [ IF NOT EXISTS ] database_name
```

注意：lindorm 2.2.16版本前，默认以database的别名schema为主，默认支持等价的语法create schema [if not exists] schema_name

示例

```
create schema test;
create database test;
```

7.8.1.5.4. DROP DATABASE

删除Lindorm Database。

语法

```
drop_database_statement ::=  DROP DATABASE [ IF EXISTS ] database_name
```

注意：lindorm 2.2.16版本前，默认以database的别名schema为主，默认支持等价的语法drop schema [if exists] schema_name

示例

```
drop schema if exists test;
drop database if exists test;
```

7.8.1.5.5. CREATE TABLE

创建Lindorm表结构。

语法

```
create_table_statement ::=  CREATE TABLE [ IF NOT EXISTS ] table_name
                            '('
                                column_definition
                                ( ',' column_definition )*
                                 ',' [constraint pk]  PRIMARY KEY '(' primary_key ')'
                            ')' WITH [ table_options ]
column_definition      ::=  column_name lql_type
primary_key            ::=  column_name [ ',' column_name (ASC|DESC)]
```

> **说明** 创建表语句支持IF NOT EXISTS，其中主键由某个或某几个列组合而成。

支持的Table_Option列表

选项	类型	描述
COMPRESSION	String	表的压缩算法。默认不设置，可选的压缩算法分别是： • SNAPPY • ZSTD • LZ4
TTL	Int	数据有效期，单位为秒，默认不过期。
MUTABILITY	String	索引相关，表示对主表的写入模式进行分类，可选值： • IMMUTABLE • IMMUTABLE_ROWS • MUTABLE_LATEST（默认值） • MUTABLE_ALL **说明** 具体含义请参见高性能原生二级索引。
CONSISTENCY	String	表的一致性。默认主表和索引表最终一致，包括以下两种级别： • eventual：表示在某个时间点后主表和索引表最终保持一致。 • strong：表示强一致性。在任意时刻，所有表中的数据是同步一致的。
NUMREGIONS	Int	预先设置建表时的Region数。
STARTKEY和ENDKEY	String	预先设置建表的Region分区起止Key。

选项	类型	描述
DYNAMIC_COLUMNS	String	是否开启动态列。取值： • True：是。 • False：否，默认值。 说明 动态列只支持Varbinary类型。

说明 Lindorm 2.2.16版本后设置表属性支持多种关键字，Lindorm 2.2.16版本前设置表属性无需添加 WITH 关键字，设置表属性需要在属性关键字KEY上添加单引号（'），属性VALUE上基于类型进行设置，如果属性VALUE类型是字符串则添加单引号（'）。但是自Lindorm 2.2.16版本，不但支持Lindorm 2.2.16版本前语法规则，而且还支持表属性前加关键字 WITH，所有属性的KEY两边无需添加单引号（'），属性VALUE两边添加单引号（'）。以下示例都可以执行。

示例

- 通过以下两种方式创建宽表。
  - 方式一

```
CREATE TABLE dt (
  p1 integer, p2 binary(3), c1 varchar, c2 decimal(22, 2),
  PRIMARY KEY(p1, p2 desc));
```

  - 方式二

```
CREATE TABLE dt (
  p1 integer, p2 binary(3), c1 varchar, c2 decimal(22, 2),
  constraint pk PRIMARY KEY(p1, p2 desc));
```

- 通过以下两种方式创建宽表并指定表的压缩算法和数据有效期。
  - 方式一

```
CREATE TABLE dt (
  p1 integer, p2 integer, c1 varchar, c2 bigint,
  constraint pk PRIMARY KEY(p1 desc)) WITH (COMPRESSION='ZSTD', TTL='2592000');
```

  - 方式二

```
CREATE TABLE dt (
  p1 integer, p2 integer, c1 varchar, c2 bigint,  constraint pk PRIMARY KEY(p1 desc)) 'COMPRESSION'='ZSTD', 'TTL'='2592000';
```

7.8.1.5.6. ALTER TABLE

修改表的相关属性，包括增加列和修改表属性（table_options）。

语法

```
alter_table_statement   ::=  ALTER TABLE table_name alter_table_instruction
alter_table_instruction ::=  ADD column_name lql_type ( ',' column_name lql_type )*
                           | table_options
```

说明 关于Table_Options选项，请参见支持的Table_Option列表。

示例

```
//增加单列
ALTER TABLE test ADD c4 VARCHAR;
//批量增加多列
ALTER TABLE test ADD c3 INTEGER, c4 INTEGER, c5 VARCHAR;
//开启动态列，动态列只支持Varbinary类型。
ALTER TABLE test SET 'DYNAMIC_COLUMNS' = 'true';
```

7.8.1.5.7. TRUNCATE TABLE

通过TRUNCATE TABLE语法可以清空表内的数据。云原生多模数据库Lindorm 2.2.16版本后为了防止误清空表的数据，必须先执行OFFLINE TABLE后才能执行TRUNCATE TABLE。

语法

```
truncate_table_statement ::= TRUNCATE TABLE table_name
```

> **说明** table_name：目标表的名称。

示例

```
TRUNCATE TABLE dt;
TRUNCATE TABLE my_schema.dt;
```

7.8.1.5.8. DROP TABLE

删除表。lindorm-2.2.16版本后为了防止误删表，必须先执行offline table 后才能执行drop table

语法

```
drop_table_statement ::=  DROP TABLE [ IF EXISTS ] table_name
```

示例

```
use my_schema;
DROP TABLE dt;
DROP TABLE IF EXISTS dt;
DROP TABLE my_schema.dt;
```

7.8.1.5.9. CREATE INDEX

创建二级索引。
Lindorm提供了主键索引功能，即按照Primary Key的定义进行排序的索引。Lindorm SELECT可基于此主键索引高效地执行整行查询等操作。但若需要使用主键（Primary Key）之外的列进行查询，可以通过Lindorm提供的二级索引来满足上述需求。

语法

```
create_index_statement ::=  CREATE INDEX [ index_name ]
                               ON table_name '(' index_identifier ')'
                            [INCLUDE include_identifier]
                            [ASYNC]
                               [ index_options ]
index_identifier       ::=  '('column_name1 [desc],...,column_namen [desc]')'
include_identifier   ::= '('column_name1,...,column_namen ')'
```

支持的Index_Options列表

名称	类型	描述
COMPRESSION	String	索引表的压缩算法，支持的压缩算法包括： • SNAPPY • ZSTD • LZ4
INDEX_COVERED_TYPE	String	表示索引的冗余方式，支持如下方式： • COVERED_ALL_COLUMNS_IN_SCHEMA表示冗余所有列。 • COVERED_DYNAMIC_COLUMNS表示冗余动态列。 • 如果没有该选项，可以使用Include表示冗余Includes的列。
STARTKEY和ENDKEY	String	表示索引表的起止Key。
NUMREGIONS	String	表示索引表的预分区数。

> **说明**
> - 索引表支持是否冗余列，冗余列可以帮助加速查询，避免回查主表。如果只是冗余某几个列可以使用Include。Include（c1,c2）表示冗余c1和c2列。
> - 默认表的索引构建方式是同步的，但是如果添加Async表示异步构建，需要执行BUILD INDEX命令。
> - 索引表数目不超过5个，过多索引表会增加成本和写入RT。

示例

```
//异步建索引，执行后返回，后台不对历史数据建索引，如需构建，使用BUILD INDEX命令。
create index idx1 on test(c4 desc) include(c5,c6) async 'COMPRESSION'='ZSTD';
//同步建索引，执行后阻塞，对历史数据建完索引后返回。
create index idx1 on test(c4 desc) include(c5,c6)  'COMPRESSION'='ZSTD';
//冗余所有列。
create index idx1 on test(c4 desc)
'INDEX_COVERED_TYPE'='COVERED_ALL_COLUMNS_IN_SCHEMA';
//冗余动态列。
create index idx1 on test(c4 desc) 'INDEX_COVERED_TYPE'='COVERED_DYNAMIC_COLUMNS';
//如果要预分区，则指定STARTKEY, ENDKEY和NUMREGIONS三个属性。
//NUMREGIONS：分区个数。

create index idx1 on test(c4 desc) include(c5,c6)  'NUMREGIONS' ='32';
```

7.8.1.5.10. ALTER INDEX

修改索引状态。

语法

```
alter_index_statement ::=  ALTER INDEX [ index_name ]
                                ON table_name index_state
```

Index_State支持如下状态：

- DISABLE：禁用索引，再次启用前，需要重新改为BUILDING状态，再重新执行BUILD INDEX操作。
- BUILDING：重新对历史数据构建索引，索引构建完成后，索引状态会自动切换为ACTIVE。
- ACTIVE：将索引状态切换为有效状态。

注意 Lindorm 2.2.16版本以前索引支持如下状态：

- DISABLED：禁用索引，再次启用前，需要进行REBUILD操作。
- REBUILD：重新对历史数据构建索引，索引构建完成后，索引状态会自动切换为USABLE。
- UNUSABLE：禁用索引，再次启用前无需进行REBUILD操作，因为禁用期间主表数据会同步应用到索引表。
- USABLE：启用索引。

创建索引后，如果数据库中有存量历史数据需要执行 `BUILD INDEX` 操作，具体操作请参见BUILD INDEX。

示例

```
ALTER INDEX idx1 ON test DISABLED;
ALTER INDEX idx1 ON test ACTIVE;
ALTER INDEX idx1 ON test BUILDING;
```

Lindorm 2.2.16版本前，修改索引状态的示例语句为：

```
ALTER INDEX idx1 ON test DISABLE;
ALTER INDEX idx1 ON test REBUILD;
ALTER INDEX idx1 ON test UNUSABLE;
ALTER INDEX idx1 ON test USABLE;
```

7.8.1.5.11. BUILD INDEX

Lindorm 2.2.16版本后，通过 `BUILD INDEX` 语句可以单独构建二级索引。

语法

Lindorm 2.2.16版本后，创建完索引，如果有存量历史数据需要执行 `BUILD INDEX` 操作，请参考如下语法进行操作。该操作默认为异步构建索引。

```
build_index_statement ::=  BUILD INDEX [ index_name ]
                                ON table_name
```

示例

```
BUILD INDEX idx ON test;
```

7.8.1.5.12. SHOW INDEX

索引结构展示。

语法

```
showindex_statement ::= show index from table_name;
```

示例

```
show index from test;
```

7.8.1.5.13. DROP INDEX

删除索引。删除索引表之前请使用ALTER INDEX方法将索引表设置成DISABLED状态。

语法

```
dropindex_statement ::= drop index [if exists] index_name on table_name;
```

示例

```
drop index idx1 on test;
```

7.8.1.5.14. OFFLINE TABLE

Lindorm 2.2.16版本后，通过 `OFFLINE TABLE` 语句可以下线Lindorm宽表，在执行删除表操作之前需要执行 `OFFLINE TABLE` 操作。

语法

```
OFFLINE TABLE table_name
```

示例

```
OFFLINE TABLE test;
```

7.8.1.5.15. ONLINE TABLE

Lindorm 2.2.16版本后，通过 `ONLINE TABLE` 语句上线Lindorm宽表。

语法

```
ONLINE TABLE table_name;
```

示例

```
ONLINE TABLE test;
```

7.8.1.5.16. CREATE SEARCH INDEX

云原生多模数据库Lindorm支持通过 `CREATE SEARCH INDEX` 创建搜索索引。

使用说明

- 搜索索引主要适用于多维查询场景，支持任意列的排序或分页查询。
- 一张表只能生成一个搜索索引，可以在原有搜索索引基础上增加新的索引列。
- 如果历史数据中有新增的列，需要进行一次全量索引构建，支持星号（*）创建所有列的搜索索引。
- 索引列支持的数据类型包括：TINYINT、SMALLINT、INTEGER、INT、BIGINT、LONG、FLOAT、DOUBLE、VARCHAR、BINARY、VARBINARY、BOOLEAN。

语法

```
CREATE SEARCH INDEX [ IF NOT EXISTS ] index_name
        ON table_name '(' index_identifier ')'
        [ASYNC]
        [ WITH '(' index_options ')' ]
index_identifier:
(column_name'(' column_options ')' ',' ... column_name'(' column_options ')')* | "*"
column_options:
option_name '=' option_value (',' option_name '='option_value)
index_options:
option_name '=' option_value (',' option_name '='option_value)
```

参数

column_options参数的说明。

参数	类型	描述

参数	类型	描述
indexed	String	是否需要搜索索引。 • true：默认值，需要搜索索引。 • false：不需要搜索索引。
rowStored	String	是否存储原始数据。 • true：存储原始数据。 • false：默认值，不存储原始数据。
columnStored	String	是否设置为列存储，用来加速排序分析。 • true：默认值，设置列存储。 • false：不设置列存储。
type	String	分词场景下，分词字段需要设置type参数为text，其它情况默认与宽表数据类型保持一致。 **说明** 需要同时与analyzer分词器使用。
analyzer	String	分词器列表。取值： • standard • english • ik • whitespace • comma **说明** • 需要同时与type参数使用。 • 当列在宽表中类型为数据类型（例如：INT）时，需要用standard分词器。

index_options参数的说明。

参数	类型	描述
indexState	String	搜索索引状态，取值为： • BUILDING：默认值。 • ACTIVE：即为USABLE，属于启用搜索索引状态。在没有历史数据的场景下，设置为ACTIVE。
numShards	String	指定分片数，默认是搜索节点个数的两倍。

示例

1. 创建SCHEMA和表。如果SCHEMA和表已存在，可以跳过该步骤。

```
CREATE SCHEMA test;
USE test;
CREATE TABLE IF NOT EXISTS tbn (p1 varchar, p2 varchar, col1 bigint, col2 varchar, col3 varchar, constra
int primary key (p1, p2 desc));
```

2. 根据不同场景创建搜索索引。
   - 通过指定索引列属性的方式，系统默认会执行同步构建搜索索引，对历史数据执行索引构建，执行过程所需的时间较长。

     ```
     CREATE SEARCH INDEX IF NOT EXISTS idx on tbn(col1, col2, col3);
     ```

   - 创建搜索索引时如果不需要执行同步构建搜索索引，可以在创建搜索索引语句后增加 `ASYNC` 。后续如果需要执行同步

构建搜索索引，请参见步骤3。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on tbn(col1, col2, col3) ASYNC;
```

- 创建所有列的搜索索引，不指定具体列属性，均为默认值。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on tbn("*");
```

- 创建所有列的搜索索引，指定具体列属性。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on tbn("*",col1(type=text,analyzer=ik,indexed=true));
```

- 指定所有列共同属性。如果索引列有col1，col2，col3，索引列的属性都为type=text，analyzer=ik，indexed=true。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on tbn("*"(type=text,analyzer=ik,indexed=true));
```

- 指定列，不指定列属性。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on tbn(col1, col2, col3);
```

- 指定列，并指定部分列属性，同时指定搜索索引状态。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on tbn(col1, col2(type=text,analyzer=ik), col3) WITH  (indexStat
e=ACTIVE, numShards=4);
```

> **说明** 当设置 `column_options` 时，type和analyzer属性需要同时设置。

3. 如果使用 `ASYNC` 方式异步构建搜索索引，并且需要历史数据也被检索，请执行以下语句，默认同步构建搜索索引。执行后，可以在LTS页面左侧导航栏选择**Lindorm Search > 全量同步**，查看REBUILD进度，具体操作请参见开通并登录LTS。

```
ALTER SEARCH INDEX IF EXISTS idx ON tbn REBUILD;
```

7.8.1.5.17. DROP SEARCH INDEX

云原生多模数据库Lindorm支持通过 `DROP SEARCH INDEX` 删除表的搜索索引。

前提条件

删除搜索索引前，需要先修改搜索索引属性为 `DISABLED` ，具体操作请参见ALTER SEARCH INDEX。

语法

```
DROP SEARCH INDEX [IF EXISTS] index_name ON table_name
```

示例

1. 创建SCHEMA和表。如果SCHEMA和表已存在，可以跳过该步骤。

```
CREATE SCHEMA test;
USE test;
CREATE TABLE IF NOT EXISTS tbn (p1 varchar, p2 varchar, col1 bigint, col2 varchar, col3 varchar, constra
int primary key (p1, p2 desc));
```

2. 创建搜索索引。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on tbn(col1, col2, col3);
```

3. 删除表的搜索索引。删除前需要修改表的搜索索引属性为禁用。

```
ALTER SEARCH INDEX IF EXISTS idx ON tbn DISABLED;
DROP SEARCH INDEX IF EXISTS idx ON tbn;
```

7.8.1.5.18. ALTER SEARCH INDEX

云原生多模数据库Lindorm支持通过 `ALTER SEARCH INDEX` 新增索引列，删除索引列，修改搜索索引属性。

使用说明

- 执行新增索引列或删除索引列操作后，还需要执行 `REBUILD` 操作。
- 索引列支持的数据类型包括：TINYINT、SMALLINT、INTEGER、INT、BIGINT、LONG、FLOAT、DOUBLE、VARCHAR、BINARY、VARBINARY、BOOLEAN。

新增索引列

语法

```
ALTER SEARCH INDEX [IF EXISTS] index_name ON table_name ADD COLUMNS '(' index_identifier ')'
index_identifier:
(column_name['(' column_options ')'] ',' ... column_name['(' column_options ')'])*
column_options:
option_name '=' option_value (',' option_name '=' option_value)
```

参数

column_options参数的说明。

参数	类型	描述
indexed	String	是否需要索引。 • true：默认值，需要索引。 • false：不需要索引。
rowStored	String	是否存储原始数据。 • true：存储原始数据。 • false：默认值，不存储。
columnStored	String	是否设置为列存储，用来加速排序分析。 • true：默认值，设置列存储。 • false：不设置列存储。
type	String	分词场景下，分词字段需要设置type参数为text，其它情况默认与宽表数据类型保持一致。 **说明** 需要同时与analyzer分词器使用。
analyzer	String	分词场景下需要使用分词器。取值： • standard • english • ik • whitespace • comma **说明** • 需要同时与type参数使用。 • 当列在宽表中类型为数据类型（例如：INT）时，需要用standard分词器。

示例

1. 创建SCHEMA和表。如果SCHEMA和表已存在，可以跳过该步骤。

```
CREATE SCHEMA test;
USE test;
CREATE TABLE IF NOT EXISTS tbn (p1 varchar, p2 varchar, col1 bigint, col2 varchar, col3 varchar, constra
int primary key (p1, p2 desc));
```

2. 新增一个或多个索引列。
   - 不指定具体列属性。

     ```
     ALTER SEARCH INDEX IF EXISTS idx ON tbn ADD COLUMNS(col1,col2,col3);
     ```

   - 指定具体列属性。

     ```
     ALTER SEARCH INDEX IF EXISTS idx ON tbn ADD COLUMNS(col1,col2,col3(type=text,analyzer=ik));
     ```

3. 有历史数据的场景，需要执行 `REBUILD` 操作，默认同步执行。执行后，可以在LTS页面左侧导航栏选择**Lindorm Search > 全量同步**，查看REBUILD进度，具体操作请参见开通并登录LTS。

   ```
   ALTER SEARCH INDEX IF EXISTS idx ON tbn REBUILD;
   ```

删除索引列

语法

```
ALTER SEARCH INDEX [IF EXISTS] index_name ON table_name DROP COLUMNS '(' index_identifier ')'
index_identifier:
(column_name ',' ... column_name)*
```

示例

- 删除多个索引列，不指定具体列属性。

  ```
  ALTER SEARCH INDEX IF EXISTS idx ON tbn DROP COLUMNS(col2,col3);
  ```

- 删除后，需要重新构建索引，即执行 `REBUILD` 操作，默认同步执行。执行后，可以在LTS页面左侧导航栏选择**Lindorm Search > 全量同步**，查看REBUILD进度，具体操作请参见开通并登录LTS。

  ```
  ALTER SEARCH INDEX IF EXISTS idx ON tbn REBUILD;
  ```

修改搜索索引属性

语法

```
ALTER SEARCH INDEX [IF EXISTS] index_name ON table_name index_state_definition
                                [ASYNC]
index_state_definition:
REBUILD | DISABLED | UNUSABLE | USABLE
```

参数

index_state_definition参数的说明。

参数	类型	描述
DISABLED	String	禁用搜索索引，再次启用前，需要进行 `REBUILD` 操作。
REBUILD	String	重新对历史数据构建搜索索引，搜索索引构建完成后，搜索索引状态会自动变为USABLE。
UNUSABLE	String	禁用搜索索引，再次启用前无需进行 `REBUILD` 操作，因为禁用期间主表数据会同步应用到索引表。
USABLE	String	启用搜索索引。

示例

- 禁用搜索索引。

```
ALTER SEARCH INDEX IF EXISTS idx ON tbn DISABLED;
```

- 重新对历史数据构建搜索索引，默认同步执行构建，也可以通过 `ASYNC` 参数指定异步执行构建。执行后，可以在LTS页面左侧导航栏选择**Lindorm Search > 全量同步**，查看REBUILD进度，具体操作请参见开通并登录LTS。

```
ALTER SEARCH INDEX IF EXISTS idx ON tbn REBUILD;
ALTER SEARCH INDEX IF EXISTS idx ON tbn REBUILD ASYNC;
```

7.8.1.5.19. SHOW SEARCH INDEX

云原生多模数据库Lindorm支持通过 `SHOW SEARCH INDEX` 查看表的搜索索引。

语法

```
SHOW SEARCH INDEX FROM table_name
```

示例

1. 创建SCHEMA和表。如果SCHEMA和表已存在，可以跳过该步骤。

```
CREATE SCHEMA test;
USE test;
CREATE TABLE IF NOT EXISTS tbn (p1 varchar, p2 varchar, col1 bigint, col2 varchar, col3 varchar, constra
int primary key (p1, p2 desc));
```

2. 创建搜索索引。

```
CREATE SEARCH INDEX IF NOT EXISTS idx on tbn(col1, col2, col3);
```

3. 查看表的搜索索引。

```
SHOW SEARCH INDEX FROM tbn;
```

7.8.1.6. DML

7.8.1.6.1. SELECT

通过SELECT语句可以查询一个或多个表格中的行数据。

> **说明** 与RDBMS不同，Lindorm的SELECT语句仅支持扁平化的查询。

语法

```
SELECT ( select_clause | '*' )
    FROM table_name
    [force index(index_name)]
    [ WHERE where_clause ]
    [ ORDER BY ordering_clause ]
    [ LIMIT integer ] | [LIMIT integer, integer]
    [`ALLOW FILTERING`]
select_clause    ::=  selector [ AS identifier ] ( ',' selector [ AS identifier ] )
selector         ::=  column_name
                      | term
                      | function_name '(' [ selector ( ',' selector )* ] ')'
                      | COUNT '(' '*' ')'
where_clause     ::=  relation ( AND|OR relation )*
relation         ::=  column_name operator term
                      '(' column_name ( ',' column_name )* ')' operator tuple_literal
operator         ::=  '=' | '<' | '>' | '<=' | '>=' | '!=' | IN | IS NOT? NULL
ordering_clause  ::=  column_name [ ASC | DESC ] ( ',' column_name [ ASC | DESC ] )*
```

参数说明

SELECT语法中部分参数说明如下：

- 执行SELECT语句时如果您需要强制选择某条索引，请在查询语句后使用 `force index(index_name)` 。
- `LIMIT` 后只跟一个数字，表示LIMIT限制。如果跟两个由英文逗号（,）分隔的数字，表示OFFSET和LIMIT条数。
- 在SELECT语句末尾添加 `ALLOW FILTERING` ，可以强制执行低效查询操作。当执行SELECT语句被Lindorm宽表引擎识别为低效查询时，为了避免这类查询语句给Lindorm带来性能稳定性风险，Lindorm宽表引擎默认不允许执行该类查询，同时会抛出异常。例如Lindorm数据库中存在一张表dt，执行 `SELECT * FROM dt WHERE nonPK=100;` 查询语句，会返回以下异常报错。如果查询语句为低效查询语句，解决方法请参见常见问题。

```
DoNotRetryIOException: Detect inefficient query: SELECT * FROM dt WHERE nonPK=100 supportEmptyResult true.
This query may be a full table scan and thus may have unpredictable performance.
```

示例

```
//简单查询操作
SELECT * FROM dt;             --选择全部列
SELECT c1,c2 FROM dt;                    --选择部分列
SELECT * FROM dt LIMIT 10;
SELECT * FROM dt WHERE pk = 10;
SELECT * FROM dt WHERE pk > 10 LIMIT 5, 20;           --跳过5行，最多返回20行
SELECT * FROM dt WHERE pk > 10 LIMIT 20 OFFSET 5;      --等价于limit 5,20
SELECT c1 FROM dt WHERE pk > 10;
SELECT c1 AS xx FROM dt WHERE pk > 10;
//聚合操作
SELECT count(*) FROM dt;
SELECT count(c1) FROM dt;
SELECT sum(c2) FROM dt;
SELECT sum(c2), AVG(c2),MIN(c2),MAX(c2) FROM dt;
//where条件中is null等用法
SELECT c1, c2 FROM dt WHERE (c1>1 OR (c2>0 AND c7 is null))
AND c3 <>1.0 AND c6 is not null;
//别名用法
SELECT count(*) AS countRow FROM dt;
SELECT c1 AS testC FROM t1 WHERE c1 = '2';
//in用法
SELECT * FROM dt WHERE dt in ('a', 'b');
//强制执行低效查询语句
SELECT * FROM dt WHERE nonPK=100 ALLOW FILTERING;
```

常见问题

- Q：什么情况下，查询语句被认为是低效查询语句？
  A：如果SELECT查询的条件同时满足下列三个条件，则该查询被认为是低效查询。
  - `WHERE` 语句中没有指定表的第一个主键的上下界。
  - `WHERE` 语句中没有指定索引表中第一个索引列的上下界。
  - `WHERE` 语句中存在非第一个主键（或非第一个索引列）的条件。
- Q：如果查询语句为低效查询语句，应该怎么解决？
  A：解决方法有以下四种。
  - 在查询语句末尾增加 `ALLOW FILTERING` ，如下语句，表示跳过该检查，强制执行低效查询操作。

    ```
    SELECT * FROM dt WHERE nonPK=100 ALLOW FILTERING;
    ```

  - `WHERE` 语句中增加表的主键条件。
  - 修改表的主键设计，避免大查询，具体详情请参见主键设计的问题和示例。
  - 使用高性能原生二级索引，具体详情请参见高性能原生二级索引。

7.8.1.6.2. UPSERT

更新数据或插入数据。Upsert是Insert与Update的结合体，表示行存在时执行Update，不存在时执行Insert。执行Upsert时必须要指定完全的Primary key的相关列信息。Upsert语法支持带时间戳的插入和批量写入。

语法

```
upsert_statement ::=  UPSERT INTO table_name
                      '('columns| _l_ts_')'
                      VALUES '(' values ')'
                      [ ON DUPLICATE KEY update_column = update_value |IGNORE ]?
columns ::=  column_name, columns
values  ::=  term, values,'(' values ')'
update_column ::= column_name
update_value ::= term
```

> **说明**
>
> - 支持批量写入，指定需要写入的相关列，可以在Values关键字中确定几行数据。
> - 支持upsert on duplicate key操作，类似check and put操作，要求upsert指定的value值可以确定一行。如果ON DUPLICATE KEY跟随的是指定的需要更新的列和value（暂时无法做到基于原有key存在性判断进行对应value更新），如果存在则抛出异常。 如果是ON DUPLICATE KEY IGNORE表示忽略对应的确定的一行是否存在的检查，直接进行插入。
> - Lindorm SQL支持标准JDBC的访问方式，包括正常statement以及preparestatemt方式，但是对于标准JDBC在preparestatemt场景下的标准batch插入数据的方式仅在lindorm 2.2.16版本以后支持。

示例

```
upsert into dt (p1,p2,c1,c2) values(10, 20, 30, 40);
upsert into dt (p1,p2,c2) values(10, 20, 40);   -- 仅写入一行的部分列
upsert into dt (p1,p2,c1) values(10, 20, 30);   -- 再写入同一行的其他列
-- 等价于upsert into dt (p1,p2,c2) values(10, 20, 5)，忽略on duplicate key ignore操作
upsert into dt (p1,p2,c2) values(10, 20, 5) on duplicate key ignore;
-- 条件更新，对数据源行p1=10，p2=20,c2=5，进行c1 = 30的列的数据插入。
upsert into dt (p1,p2,c2) values(10, 20, 5) on duplicate key update c1 = 30;
-- 插入的行，时间戳是111232
 UPSERT INTO dt (p1,p2,p3,_l_ts_) values('1','2','3',111232);
-- 批量写入
upsert into dt (p1,p2,c1,c2) values(1,2,3,4), (2,3,4,5), (3,4,5,6);
--- upsert on duplicate key ignore，如果1 2 3 4 5 确定的行不存在则insert
upsert into dt (p1,p2,p3,c1,c2) values (1,2,3,4,5) on duplicate key ignore;
--- upsert on duplicate key ，如果1 2 3 4 5 确定的行不存在则不insert，抛出异常，存在则update c2 为6
upsert into dt (p1,p2,p3,c1,c2) values (1,2,3,4,5) on duplicate key update c2 = 6;
```

7.8.1.6.3. DELETE

删除数据，可以删除表中的一行数据（不可删除多行），可指定过滤条件。
DELETE语法是高危操作，为了确保数据安全，Lindorm将DELETE的语义限制为只能使用完备的主键删除一行数据。

语法

```
delete_statement ::=  DELETE FROM table_name
                      WHERE where_clause
```

> **说明** Where_Clause必须可以唯一限定一行数据。

语法限制

- DELETE的WHERE子句中，必须给出能够唯一定位一行数据的完备主键条件。
- 一条DELETE语句只能删除一行数据，不支持一条语句删除多行。

- 删除操作并非是先读取再删除，而是直接写入删除标记。所以无论被删除的行是否存在，删除都会成功。
- 可以指定删除的时间戳，带时间戳的delete会删除指定行中，小于等于指定时间戳的列。

示例

```
delete from dt where p1 = 10 and p2 = 20;
-- 以下语句是非法的
delete from dt where p1 = 10; -- dt表有两个主键列，p1=10无法唯一确定一行
delete from dt where c1 = 30; -- 必须给出主键列的条件
```

7.8.1.7. DCL

7.8.1.7.1. 概述

DCL包含用户角色管理语法和权限管理语法。

用户角色管理语法

Lindorm宽表SQL语法使用database user来表示用户角色。定义如下：

```
user_name ::=  identifier | string
```

权限管理语法

Lindorm宽表SQL语法的权限表示对于某个用户具备某个schema或者某个table的相关权限，如果权限操作没有指定具体的schema或者table。那么默认表示的是在全局级别（global level）。权限包括如下：

权限	描述
READ	表示某个表或者schema下所有表具备读的权限。
WRITE	表示某个表或者schema下所有表具备写的权限。
ADMIN	表示某个表或者schema下所有表具备ADMIN的权限。
TRASH	表示某个表或者schema下所有表具备删除的权限。
SYSTEM	表示某个表或者schema下所有表具备系统管理的权限（一般不开放）。
NORMAL_USER	权限价值等于：READ+WRITE。
ADMIN_USER	权限价值等于：READ+WRITE+ADMIN。
ADMIN_TRASH	权限价值等于：READ+WRITE+ADMIN+TRASH。
ALL	权限价值等于：READ+WRITE + ADMIN + TRASH + SYSTEM 。

7.8.1.7.2. SHOW USERS

展示所有用户。

示例

```
show users;
```

7.8.1.7.3. CREATE USER

创建用户并设置密码。

语法

```
create_user_statement ::=  CREATE USER user_name identified by '''password''';
password ::= string
```

参数

参数	说明
user_name	指定用户名称。
password	表示用户登录的密码。

示例

创建用户并指定相关的密码。

```
create user test_user identified by 'testuserpassword';
```

7.8.1.7.4. ALTER USER

修改用户密码。

语法

```
alter_user_statement ::=  ALTER USER user_name SET password='newpassord'
            | WHERE oldpassword='oldpassword';
```

示例

修改用户的密码信息。

```
 alter user test_user set password='fda.@' where oldpassword='t12a@Hi';
```

7.8.1.7.5. DROP USER

删除用户。

语法

```
drop_user_statement ::=  DROP USER user_name
```

参数

user_name表示指定删除的用户名称。

示例

```
DROP USER test_user;
```

7.8.1.7.6. GRANT PERMISSION

为用户授予指定权限。

语法

```
grant_permission_statement ::=  GRANT permissions ON [GLOBAL|SCHEMA schema| TABLE table] resource TO user_na
me
permission                 ::=  ALL | READ | WRITE | ADMIN | TRASH
resource                   ::=  schema_name
                              | table_name
```

> **说明** 权限范围：GLOBAL > SCHEMA > TABLE

参数

参数	说明
user_name	指定用户名称。
permissions	• `ALL` 指定具有全部的权限，包括下面所属。 • `READ` 指定具有“读“的权限。 • `WRITE` 指定具有“写“的权限。 • `ADMIN` 指定具有“管理员“的权限。 • `TRASH` 指定具有“删除“的权限。
resource	指定被授予权限的资源，如 `ALL` 、 `schema` 、 `TABLE` 。
schema_name	指定Schema名称。
table_name	指定表名称。

示例

```
GRANT READ TO User1;

GRANT WRITE ON TABLE Table1 TO User2;

GRANT ADMIN ON TABLE my_schema.my_table TO User2;

GRANT TRASH ON SCHEMA my_schema TO User3;
```

7.8.1.7.7. REVOKE PERMISSION

删除某个用户指定的权限。

语法

```
revoke_permission_statement ::=  REVOKE permissions ON [GLOBAL | SCHEMA | TABLE] resource FROM user_name
resource ::= schema_name | table_name
```

参数

参数名	说明
user_name	指定用户名称。

参数名	说明
permissions	• `ALL` 指定具有所有的权限，包括下述的。 • `READ` 指定具有“读“的权限。 • `WRITE` 指定具有“写“的权限。 • `ADMIN` 指定具有“管理员“的权限。 • `TRASH` 指定具有“删除“的权限。
resource	指定被授予权限的资源，如 `ALL` 、 `schema` 、 `TABLE` 。

说明 权限范围：GLOBAL > SCHEMA > TABLE

示例

```
REVOKE READ FROM User1;

REVOKE WRITE ON TABLE Table1 FROM User2;

REVOKE ADMIN ON TABLE my_schema.my_table FROM User2;

REVOKE TRASH ON SCHEMA my_schema FROM User3;
```

7.8.1.8. EXPLAIN

使用EXPLAIN查看Select语句的执行计划操作。在EXPLAIN语法的帮助下，您可以查看表中添加索引的位置，以便通过索引查找行来更快地执行语句。

语法

```
explain select select_options;
```

示例

```
explain select c1,c2, c3 from test where c1>1 and c2<3;
```

7.8.2. Lindorm SQL ORDER BY介绍

本文介绍在数据量较大的结果集场景中Lindorm SQL ORDER BY的使用解决方案以及最佳案例。

使用场景

常见Lindorm SQL的ORDER BY语法使用场景如下：

- 在数据量较小（10万内的数据量）的结果集场景中，使用ORDER BY可以在内存中计算，没有任何使用限制。这种场景需要开启在内存中计算，具体开启的方法请提交工单。
- 在数据量较大的结果集场景中，使用ORDER BY需要依赖Lindorm本身对数据的存储规则，所以存在一定的限制。使用ORDER BY过程中不对数据做额外的排序，如果有需要排序的列，可以将此列设置为单等值前缀列。

注意 开启在内存中计算，如果结果集较大可能会导致客户端停止响应。

解决方案

一般情况下，创建表格后只能进行有限的排序，不支持任意列排序。如果有需要排序的列，可以将此列设置为单等值前缀列。例如以下代码。

```
CREATE TABLE test(p1,p2,p3,c1,c2,c3, primary key(p1,p2,p3))   // 其中p1,p2,p3为联合主键
SELECT * FROM test WHERE p1=? ORDER BY p2                     // 该SQL支持ORDER BY
SELECT * FROM test WHERE p1=? and p2=? ORDER BY p3            // 该SQL支持ORDER BY
SELECT * FROM test WHERE p1=? and c1=? ORDER BY p2            // 该SQL支持ORDER BY，p2列已设置为单等值前缀列
SELECT * FROM test ORDER BY p1                                // 该SQL支持ORDER BY，但是不建议这么用
SELECT * FROM test WHERE p1=? ORDER BY p3                     // 该SQL不支持ORDER BY，要求给定所有的前缀
SELECT * FROM test WHERE p1<? ORDER BY p2                     // 该SQL不支持ORDER BY，要求等值前缀
SELECT * FROM test WHERE p1=? and p2>? ORDER BY p3            // 该SQL不支持ORDER BY，要求等值前缀
SELECT * FROM test WHERE p1=? and p2=? and p3=? ORDER BY c1   // 该SQL不支持ORDER BY，c1为非主键列
SELECT * FROM test WHERE p1=a and p1=b ORDER BY p2            // 该SQL不支持ORDER BY，要求单值前缀，而p1给了好几个值
```

在数据量较大的结果集场景中，如果您必须使用ORDER BY排序，提供以下解决方案。

- 修改表的主键。
- 为需要排序的列创建二级索引或者搜索索引。例如上述代码中的 `SELECT * FROM test WHERE p1=? ORDER BY p3` ，通过创建二级索引语句 `CREATE INDEX idx ON test(p1, p3)` 可以支持排序。

示例

在数据量较大的结果集场景中，如果您必须使用ORDER BY排序，可以参考如下示例。

- 由于二级索引与主表数据没有存储在同一张表中，所以在创建二级索引使用ORDER BY排序时，您需要完全指定二级索引列。代码如下：

```
//需要排序c1列
SELECT * FROM test WHERE p1=? ORDER BY c1
//解决方案：创建二级索引实现c1列的排序
CREATE INDEX idx ON test(p1, c1)
```

- 二级索引列排序方式与需要的常用排序方式保持一致，性能最佳。代码如下：

```
//需要排序c1列
SELECT * FROM test WHERE p1=? ORDER BY c1 desc
//解决方案：创建二级索引实现c1列的排序
CREATE INDEX idx ON test(p1, c1 desc)
```

- ORDER BY多列场景中，需要保证多列连续，并且在表中的顺序与ORDER BY的顺序保持一致，性能最佳。代码如下：

```
//主表结构
CREATE TABLE test (p1 asc,p2 asc,p3,c1,c2)
//需要排序c1和c2索引列
SELECT * FROM table WHERE p1 = ? ORDER BY c1 desc, c2 asc
//创建二级索引实现排序时要保证c1、c2索引列与ORDER BY排序方式相同
CREATE INDEX idx ON test(p1, c1 desc, c2 asc)
```

7.8.3. 主键设计的问题和示例

Lindorm主键设计在数据分区和数据查询中很重要，本文介绍设计主键前需要考虑的一些问题以及设计示例。

问题考虑

- Q：主键是唯一的吗？
  A：相同的主键在Lindorm中被认为是同一条数据的多个版本，查询时默认返回最新版本的数据，所以通常主键都需要保证唯一，除非用到多版本特性。
  最佳设计示例：主键可以是一列，也可以是多列的组合。每个主键表示一条记录。
  - `[userid]` ：表示主键只有一列，每个用户只有一条记录。
  - `[userid][orderid]` ：表示主键为两列的组合，每个用户有多条记录。
- Q：满足哪种查询场景？
  A：主键的设计限制了数据的查询方式，一条 `SELECT` 查询语句，Lindorm服务器端会编译为两种查询方式。

- 根据完整的主键查询（get方式），例如 `SELECT * FROM table WHERE userid='abc' AND orderid=123` 。

  > **说明** get方式需要知道所有的主键列，即组成主键所有字段的值都是确定的。

- 根据主键的范围查询（scan方式），例如 `SELECT * FROM table WHERE userid='abc' AND 123<orderid<456` 。

  > **说明** scan方式需要指定第一列主键的范围，否则Lindorm默认会拒绝低效的全表扫描查询，具体详情请参见ALLOW FILTERING介绍。

最佳设计示例：在有限的查询方式下如何实现复杂查询？以下方法可以帮您实现。

- 再新建一张表作为索引表。
- 查询条件给定非主键列范围，服务端会使用Filter过滤不需要的数据。
- 使用二级索引。
- 使用 `ORDER BY` 方法实现倒序（将新数据排在前面），例如 `SELECT * FROM table WHERE userid='abc' AND 123<orderid<456 ORDER BY orderid DESC` 。

  > **说明** 由于表字段原始顺序的倒序性能比正序性能差，如果大部分数据是倒序场景，可以体现在主键设计上，主键设计为 `[userid][orderid DESC]` 。

- Q：数据足够分散，会存在堆积的热点现象吗？
  A：散列的目的是将数据分散到不同的分区，不至于产生热点使某一台服务器终止，其他服务器空闲，充分发挥分布式和并发的优势。
  最佳设计示例：
  - 设计md5散列算法，主键设计为 `[md5(userid).subStr(0,4)][userId][orderid]` 。
  - 设计反转，主键设计为 `[reverse(userid)][orderid]` 。
  - 设计取模，主键设计为 `[bucket][timestamp][hostname][log-event]; long bucket = timestamp % numBuckets` 。
  - 增加随机数，主键设计为 `[userId][orderid][random(100)]` 。
- Q：主键可以再精简点吗？
  A：精简的主键列可以减少数据量，提高数据查询和数据写入效率。
  最佳设计示例：
  - 使用Long或Int代替String，例如 `'2015122410' => Long(2015122410)` 。
  - 使用编码代替名称，例如 `'淘宝'=> 'tb'` 。
- Q：使用scan方式会查询出不需要的数据吗？
  A：不会。Lindorm宽表SQL支持多种数据类型，每种数据类型单独编码，具体支持的数据类型请参见数据类型。

常见设计示例

- 日志类、时间序列数据。列举出三个场景设计主键。
  - 查询某台机器某个指标某段时间内的数据，主键设计为 `[hostname][log-event][timestamp]` 。
  - 查询某台机器某个指标最新的几条数据，主键设计为 `[hostname][log-event][timestamp DESC]` 。
  - 查询的数据存在只有时间一个维度或某一个维度数据量巨大的情况，主键设计为 `long bucket = timestamp % numBuckets; [bucket][timestamp][hostname][log-event]` 。
- 交易类数据。列举出四个场景设计主键。
  - 查询某个卖家某段时间内的交易记录，主键设计为 `[seller_id][timestamp][order_number]` 。
  - 查询某个买家某段时间内的交易记录，主键设计为 `[buyer_id][timestamp][order_number]` 。
  - 根据订单号查询，主键设计为 `[order_number]` 。
  - 查询中同时满足三张表，一张买家维度表主键设计为 `[buyer_id][timestamp][order_number]` 。一张卖家维度表主键设计为 `[seller_id][timestamp][order_number]` 。一张订单索引表主键设计为 `[order_number]` 。

7.8.4. 统计表行数的方法

本文介绍如何统计云原生多模数据库Lindorm宽表的总行数。

注意事项

- 云原生多模数据库Lindorm是基于LSM-Tree存储结构的NoSQL数据库，如果您要精确查询Lindorm宽表的行数，需要扫描全表，表越大，耗时越长，所以不建议频繁在Lindorm宽表中进行COUNT操作。如果您确实需要获取表的行数，可以通过以下方式统计表行数。
  - 通过HBase Shell工具统计表行数
  - 使用HBase RowCounter统计表行数
  - 通过Lindorm SQL统计表行数
- 如果您要粗略查询Lindorm宽表的行数，可以通过Lindorm集群管理系统的**表概览**页面进行查看，详情请参见通过集群管理系统粗略查看表行数。

通过HBase Shell工具统计表行数

已通过HBase Shell连接Lindorm宽表引擎，具体操作请参见通过HBaseue Shell连接并使用宽表引擎。

通过HBase Shell工具使用COUNT命令可以精确的统计HBase表行数。执行COUNT命令的原理是将表的数据分批次全部扫描出来进行统计，所以建议在相同内网的ECS客户端上执行COUNT命令。如果通过公网执行COUNT命令，网络使用率会较大，导致统计效率降低。根据表结构的不同，扫描的速度会有所差别，使用COUNT命令扫描全表的速度可以小于10万行每秒。执行以下语句统计table表的总行数。

```
COUNT 'table'
```

执行结果如下：

```
Current count: 1000, row: user000000000000000000000000000000000000236698081956413
Current count: 2000, row: user000000000000000000000000000000000000467631147824420
Current count: 3000, row: user000000000000000000000000000000000000694638584618397
Current count: 4000, row: user000000000000000000000000000000000000921348905034511
Current count: 5000, row: user000000000000000000000000000000000001153174794400148
Current count: 6000, row: user000000000000000000000000000000000001384424654494732
Current count: 7000, row: user000000000000000000000000000000000001616145341907025
Current count: 8000, row: user000000000000000000000000000000000001851296181770033
Current count: 9000, row: user000000000000000000000000000000000002080082525016669
Current count: 10000, row: user000000000000000000000000000000000002308643211802137
```

使用HBase RowCounter统计表行数

已通过HBase Shell连接Lindorm宽表引擎，具体操作请参见通过HBaseue Shell连接并使用宽表引擎。

RowCounter会在本地启动一个伪分布式的MR任务来进行COUNT操作。默认情况下这种方式是按照单个线程来执行的，统计速率与HBase Shell执行COUNT命令的统计速率差不多。如果您想提高统计速度，可以通过指定 `Dmapreduce.local.map.tasks.maximum=线程数` 来进行多线程并发执行，需要注意以下两点：

- 线程数小于等于表Region的个数。
- 线程数增多可能造成集群负载过高从而影响线上业务，请根据业务需求设置。

在HBase Shell中执行以下代码统计Lindorm宽表的行数。

- 统计目标表table的总行数。

  ```
  ./alihbase-2.0.18/bin/hbase org.apache.hadoop.hbase.mapreduce.RowCounter  "table"
  ```

- 统计目标表table的总行数，16个线程并发执行。

  ```
  ./alihbase-2.0.18/bin/hbase org.apache.hadoop.hbase.mapreduce.RowCounter -Dmapreduce.local.map.tasks.maximum=16 "table"
  ```

- 统计NameSpace为ns中的目标表table的总行数。

  ```
  ./alihbase-2.0.18/bin/hbase org.apache.hadoop.hbase.mapreduce.RowCounter  "ns:table"
  ```

执行结果保存在Log目录中的*hbase.log*文件。

```
2022-01-18 18:13:54,435 INFO  [LocalJobRunner Map Task Executor #0] mapred.LocalJobRunner(591): map
2022-01-18 18:13:54,435 INFO  [LocalJobRunner Map Task Executor #0] mapred.Task(1158): Task 'attempt_local1438979686_0001_m_000063_0' done.
2022-01-18 18:13:54,435 INFO  [LocalJobRunner Map Task Executor #0] mapred.LocalJobRunner(249): Finishing task: attempt_local1438979686_0001_m_000063_0
2022-01-18 18:13:54,435 INFO  [Thread-27] mapred.LocalJobRunner(456): map task executor complete.
2022-01-18 18:13:55,186 INFO  [main] mapreduce.Job(1378): Job job_local1438979686_0001 completed successfully
2022-01-18 18:13:55,250 INFO  [main] mapreduce.Job(1385): Counters: 21
        File System Counters
                FILE: Number of bytes read=1900842053
                FILE: Number of bytes written=1932117632
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=0
                HDFS: Number of bytes written=0
                HDFS: Number of read operations=0
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=0
        Map-Reduce Framework
                Map input records=15110895
                Map output records=0
                Input split bytes=10796
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=4133
                Total committed heap usage (bytes)=3892314112
        org.apache.hadoop.hbase.mapreduce.RowCounter$RowCounterMapper$Counters
                ROWS=15110895
        File Input Format Counters
                Bytes Read=0
        File Output Format Counters
                Bytes Written=0
```

通过Lindorm SQL统计表行数

已通过Lindorm-cli连接宽表引擎，具体操作请参见通过Lindorm-cli连接并使用宽表引擎。

通过Lindorm SQL统计表行数的速度比使用HBase RowCounter统计表行数的速度快，是因为Lindorm自动将COUNT的统计逻辑分布式地下发给每一个Lindorm进程，相当于多线程进行统计，而HBase Shell是单线程进行统计。但是COUNT的统计过程仍然是扫描全表，同时Lindorm SQL中的执行语句默认超时时间为120秒，如果120秒内仍然无法统计出结果，该语句会超时报错。使用Lindorm SQL统计表行数的速度大概可以达到每台每秒数十万行，由于COUNT是分布式进行，集群的机器越多越快。

执行以下代码统计table表行数。

```
SELECT COUNT(*) FROM table;
```

执行结果如下：

```
+-----------+
|  COUNT(*)|
+-----------+
|    16000 |
+-----------+
1 rows in set (201 ms)
```

> **说明** 统计行数需要扫描全表，请慎重操作，如果数据量超过百万，建议使用搜索索引加速：通过搜索索引查询宽表数据

通过集群管理系统粗略查看表行数

已进入Lindorm集群管理系统，具体操作请参见登录集群管理系统。

Lindorm集群管理系统中，在**表概览**页面可以粗略查看表行数。该估计值是通过将数据文件的行数元信息累加计算出来的。如果用户存在更新，删除等操作，同一行数据可能分布在多个文件当中。同时，文件的行数元信息是在生成文件时统计的，如果用户使用了TTL功能，文件中的部分数据可能已经过期了，从而导致该信息不准。因此，该值仅仅只是一个估计值。但如果用户的表没有更新，没有删除和TTL过期，显示的行数是完全准确的，可以在历史数据迁移完成时用于估计数据完整性。Lindorm集群管理系统中，单击左侧导航栏**数据管理 > 表管理**，在**用户表**区域下，找到目标表，在EstimateRowCount列中查看目标表的粗略行数。



说明

- 如果在Lindorm集群管理系统的**表概览**页面没有**EstimateRowCount**列，说明Lindorm集群管理系统版本较低，请提交工单处理，提交工单的链接请单击提交工单。
- 如果表中确实有数据但是预估行数显示为0，那么可能Lindorm宽表引擎的小版本较低，请升级Lindorm宽表引擎小版本，具体操作请参见升级小版本。

8.时序引擎

8.1. 引擎简介

云原生多模数据库Lindorm时序引擎是一款**高性能**、**低成本**、**稳定可靠**的在线时序数据库引擎服务，提供高效读写、高压缩比存储、时序数据聚合计算等能力。

核心能力

- **高性能**：时序引擎支持高写入吞吐，通过自研的时序引擎，实现高并发的时序数据写入。
- **数据压缩**：时序引擎内置数据压缩能力，结合自研时序压缩算法以及通用块压缩，压缩率最高可达10:1。
- **时序索引**：时序引擎内置时序索引能力，支持百亿级时间线多维数据高性能查询。
- **时序计算**：时序引擎提供丰富的时序聚合计算能力，聚合引擎支持10+核心聚合算子，20+填充策略，10+插值算法。
- **弹性伸缩**：时序引擎采用分布式架构，支持在线弹性伸缩，以适应任何规模的数据存储与处理需求。

云原生多模数据库Lindorm更多特性请参见功能特性。

使用场景

云原生多模数据库Lindorm时序引擎广泛应用于**物联网（IoT）**、**工业互联网（IIoT）**、**应用性能监控（APM）**等行业场景。支持秒级写入千万级时序数据点，提供高压缩比低成本存储、预降采样、插值、多维聚合计算、可视化查询结果等功能，满足海量时序数据存储与处理。

- 物联网行业
- 汽车行业
- 工业物联网行业

访问方式

您可以使用SQL方式连接并访问云原生多模数据库Lindorm时序引擎，包括以下三种方式：

- 如果您使用Java语言访问时序引擎，具体请参见通过JDBC Driver连接并访问Lindorm时序引擎。
- 如果您使用其他语言访问时序引擎，具体请参见SQL接口应用指南。
- 如果您是DBA或者数据分析人员，具体请参见通过Lindorm-cli连接访问Lindorm时序引擎。

8.2. 产品功能

本文介绍时序引擎的实用功能。

时序数据高效读写

Lindorm时序引擎提供高效的并发读写，支持每秒百万数据点的数据读取及千万数据点的写入能力。

- **数据写入**
  支持以下方式进行数据写入：
  - SQL的INSERT语句 (推荐使用JDBC)。
  - 兼容设备通过MQTT协议写入数据。
  - 兼容InfluxDB的写入Line Protocol。
  - 兼容OpenTSDB协议的写入API。
  - 支持 Prometheus 以 Remote Storage 的形式将监控数据写入。
- **数据查询**
  支持SQL进行数据查询，兼容基于OpenTSDB协议的查询API等方式进行数据查询。用户也可以通过产品控制台的数据查询功能进行数据分组、降采样、空间聚合等可视化数据查询展现。

数据管理

- **数据库管理**
  时序数据的存储以Database为粒度进行隔离。支持通过SQL对Database的时间分区间隔、冷热分界线、数据过期时间等属性进行设置。
  详见 数据库管理 。
- **用户和权限管理**

支持对时序数据的访问进行用户认证以及存储权限鉴定。用户可以通过SQL或控制台对时序引擎内的用户以及用户权限进行管理。
详见 用户及权限管理。

高效时序存储技术

- 时序数据压缩
  Lindorm时序引擎使用高效的数据压缩技术，将单个数据点的平均使用存储空间降为1~2个字节，可以降低90%存储使用空间，同时加快数据写入的速度。
- 冷数据归档能力
  Lindorm时序引擎支持将长期存储的冷数据归档到低成本存储介质，从而将存储成本降低70%以上。

时序数据计算能力

Lindorm时序引擎提供专业全面的时序数据计算函数，支持降采样、数据插值和空间聚合计算，能满足各种复杂的业务数据查询场景。

监控运维

Lindorm时序引擎提供实例运维系统，用户可以实时掌握实例的运行情况、性能指标和存储空间使用情况，并通过设置报警通道，及时发现资源瓶颈。

数据和实例安全

Lindorm时序引擎提供以下方案保证用户数据和实例的安全：

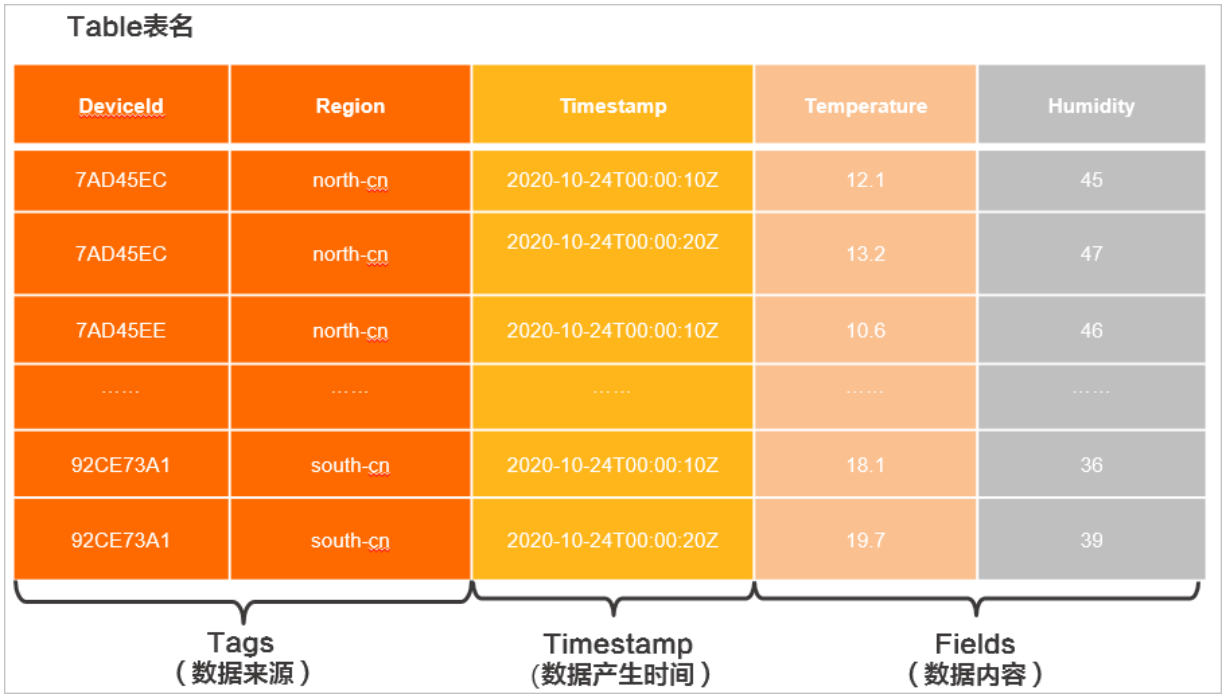
- 提供VPC的实例访问方式，充分保证实例访问的安全性。
- 提供网络白名单功能：用户可以通过设置允许访问实例的机器名单，进一步保证实例和数据的访问安全。如果一台机器在VPC内部，但不在设置的白名单内，则不能访问实例。
- 提供了数据库层面的用户权限功能，从而在网络白名单的基础上进一步对数据访问进行更细的权限管理。
- 数据存储默认采用多副本策略，充分保证数据的可用性。

8.3. 数据模型

本文介绍Lindorm时序引擎的数据模型以及相关术语。

基本概念

物联网、应用监控、工业互联网等典型的时序场景下，数据源（Data Source）通常按一定的周期持续产生时序数据，一条时序数据由Tag、Timestamp、Field等元素共同来描述，具有相同特征的一类数据存放在同一张表中，表的时序数据元素分类如下图：



时序数据元素说明

元素	描述
表名（Table）	时序表代表一系列同类时序数据的集合，例如为空气质量传感器建立一个Table，存储所有传感器的监测数据。
标签（Tag）	Tag描述数据源的特征，通常不随时间变化，例如传感器设备，包含设备DeviceId、设备所在的Region等Tag信息，数据库内部会自动为Tag建立索引，支持根据Tag来进行多维检索查询。Tag由Tag Key、Tag Value组成，两者均为String类型。 在定义时序数据表时，还可以指定特定标签列作为Primary Key来显式指定数据在多个节点上分片（*Data Sharding*）时的分片规则，从而实现贴合业务场景的性能提升。
时间戳（Timestamp）	Timestamp代表数据产生的时间点，可以写入时指定，也可由系统自动生成。
量测值（Field）	Field描述数据源的量测指标，通常随着时间不断变化，例如传感器设备包含温度、湿度等Field，Field列无需提前创建固定的Schema，可以在运行过程中动态增减；Field由Field Key、Field Value组成，Field Key为String类型，Field Value支持Boolean、Bigint、Double、String等类型。
数据点（Data Point）	数据源在某个时间产生的某个量测指标值（Field Value）称为一个数据点，数据库查询、写入时按数据点数来作为统计指标。
时间线（Time Series）	数据源的某一个指标随时间变化，形成时间线，**Tag的组合**确定一条时间线。针对时序数据的计算包括降采样、聚合（sum、count、max、min等）、插值等都基于时间线维度进行；数据库在存储数据时，会将同一条时间线的数据尽量聚类存储，提升时间线数据访问效率，同时更好的支持时序数据压缩。 在时序表中，Tag列的值都相同的一系列数据行构成了一条时间线。

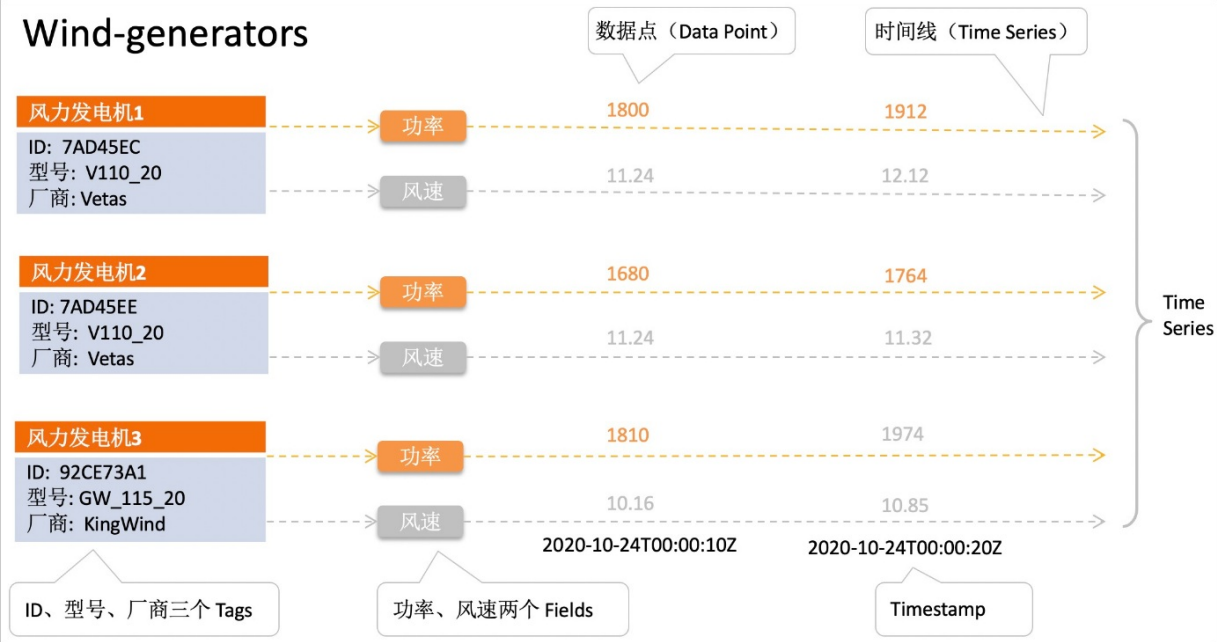
时序数据库提供一系列特有的**数据访问操作**，包括聚合（Aggregation）、降采样（Downsample）、插值（Interpolation）等，同时还提供数据生命周期（数据时效）的管理机制。

数据访问操作元素说明

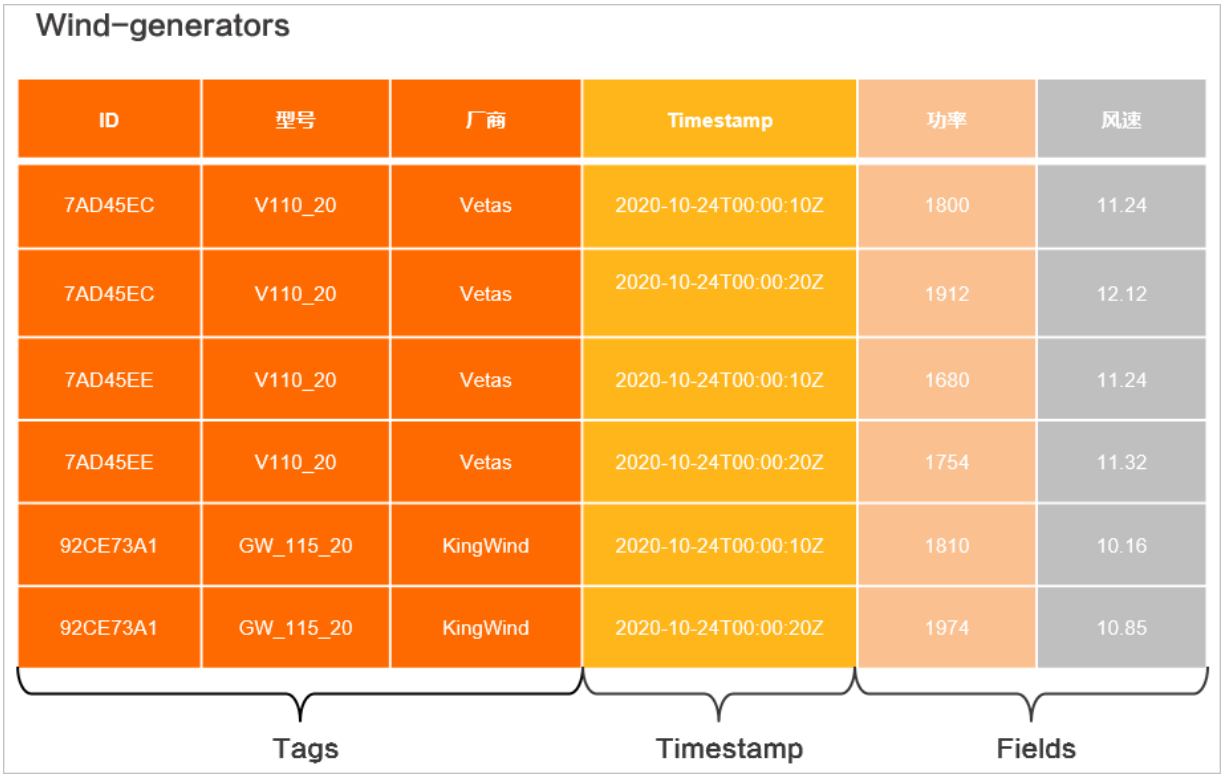
元素	描述
聚合（Aggregation）	对同一时间线、或不同时间线上的数据点进行分组、求和、计数等计算操作。
降采样（Downsample）	查询时间区间跨度较长，且原始数据采样频率较高时，可通过降采样降低数据的查询展现精度，例如按秒采集的数据，查询时按小时级别采样，减少结果集的数据点数。
插值（Interpolation）	时间线中间缺失部分数据点时，可通过插值算法进行数据点自动填充。
数据时效（TTL）	数据时效是指数据保存有效期，超过有效期的数据会被自动清理，默认数据有效期为永久保存。

场景示例

某风力电厂包含一系列的智能风力发电机设备，创建名为 `Wind-generators` 的table来存储所有设备的信息，设备由 `ID` 、`型号` 、`厂商` 等Tag信息描述，每个设备会持续上报 `功率` 、`风速` 等Field指标，指标数据通过时序数据库的API实时写入到云端时序数据库。



上述场景中三个设备产生的数据如下图所示。



时序数据库除了支持**高并发的写入吞吐**以及**高压缩率**，还能高效的满足如下类型的查询请求：

- 查询某条时间线一段时间内的数据点。
  例如：查询 `ID=7AD45EC` 的设备，在 `2020-10-24T00:00:00Z ~ 2020-10-24T00:30:00Z` 30分钟时间范围内，运行 `风速` 指标值。
- 查询多条时间线一段时间内的数据点，并按5分钟粒度降采样。

例如：查询 `ID=7AD45EC` 的设备，在 `2020-10-24T00:00:00Z ~ 2020-10-25T00:00:00Z` 一天时间范围内， `功率` 、 `风速` 的指标值明细，按 `5分钟` 粒度采样返回数据。

- 对某条时间线按时间范围做聚合。
  例如：查询 `ID=7AD45EC` 的设备，在 `2020-10-24T00:00:00Z ~ 2020-10-24T00:30:00Z` 时间范围内，运行 `功率` 的平均值。
- 查询多条时间线的最新点。
  例如：查询 `ID=7AD45EC` 的设备，最新的 `功率` 、 `风速` 。
- 按Tag做时间线聚合查询。
  例如：查询 `厂商=KingWind` 的设备，在 `2020-10-24T00:00:00Z` 的平均 `风速` 。

8.4. 与旧版TSDB的比较

本文介绍Lindorm时序引擎相比旧版时序数据库TSDB的差异和优势。
Lindorm时序引擎是孵化自阿里云时间序列数据库TSDB（以下简称**旧版TSDB**）的新一代时序引擎。Lindorm时序引擎是一款全新的自研产品，同时也在接口层面兼容了旧版TSDB的大部分API。
您可以通过下表对比Lindorm时序引擎和旧版TSDB。

	Lindorm时序引擎	旧版TSDB
访问接口	• 原生SQL能力 • 兼容大部分旧版TSDB的HTTP API • 兼容InfluxDB®的写入行协议 • 支持作为Prometheus的Remote Storage，并支持PromQL查询	• OpenTSDB风格的HTTP API • 支持作为Prometheus的Remote Storage • *基于外部组件提供少量SQL语法支持*
应用程序SDK	• JDBC • Java Native SDK • 兼容旧版TSDB的Java SDK	Java SDK
数据模型	半结构化的时序数据表模型。	OpenTSDB风格的时序数据模型。
存储引擎	自研的分布式时序存储引擎。	基于HBase的KV存储引擎。
预计算能力	• 原生支持Continuous Query • 存储引擎层面数据预降精度能力	不支持。
TTL能力	支持库级别（Database）TTL设置。	只支持全局TTL，且仅能清理数据，无法清理索引
冷热存储分层	支持库级别的冷热分层设置，支持将冷数据归档至OSS。	不支持
性能	可扩展至每秒千万点数据点写入，详细可见性能白皮书。	可扩展至每秒百万数据点写入。
应用领域	可广泛用于**物联网**、**工业互联网**、**企业应用性能监控**。	主要用于**企业应用性能监控**。

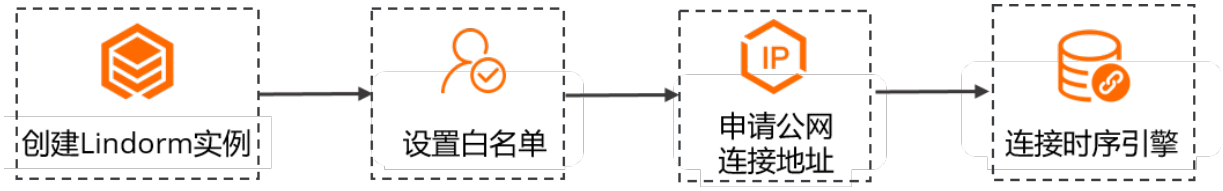
8.5. 快速入门

8.5.1. 入门概览

本文介绍连接云原生多模数据库Lindorm时序引擎的方法和相关文档，帮助您快速使用Lindorm时序引擎。

操作流程

从创建Lindorm实例到可以使用Lindorm时序引擎，您需要完成以下操作：



1. 创建Lindorm实例，具体操作请参见创建实例。
2. 设置白名单，具体操作请参见设置白名单。
3. 可选：如果您需要使用公网地址连接Lindorm时序引擎，请先申请公网连接地址。
4. 连接时序引擎。连接Lindorm时序引擎支持以下三种方法，使用场景和相关文档如下表：

连接时序引擎的方法	使用场景	相关文档
通过Java Native SDK连接（推荐）	在更高的业务模型层级访问Lindorm时序引擎，无需对连接管理或者数据转换等低维度的操作进行应用开发。	通过Java Native SDK连接并使用Lindorm时序引擎
通过JDBC Driver连接	○ 应用代码中已经使用JDBC标准API访问多种异构数据库。 ○ 应用代码中有类似MyBatis等数据访问ORM框架的应用。	通过JDBC Driver连接并使用Lindorm时序引擎
通过Lindorm-cli客户端连接	不涉及	通过Lindorm-cli连接并使用Lindorm时序引擎

8.5.2. 通过Lindorm-cli连接访问Lindorm时序引擎

本文介绍通过Lindorm-cli连接并使用云原生多模数据库Lindorm时序引擎的具体操作和使用示例。

前提条件

- 已获取Lindorm时序引擎的连接地址，具体操作请参见查看连接地址。
- 已将客户端IP地址添加至Lindorm白名单，具体操作请参见设置白名单。

连接Lidnorm时序引擎使用的网络类型

网络类型	说明
专有网络（推荐）	专有网络VPC（Virtual Private Cloud）是您自己独有的云上私有网络，不同的专有网络之间二层逻辑隔离，拥有较高的安全性和性能。Lindorm-cli部署在ECS实例上时，通过专有网络连接至Lidnorm时序引擎，可获得更高的安全性和更低的网络延迟。
公网	公网即互联网，当本地设备需要测试或管理Lidnorm时序引擎时，可在本地设备上部署Lindorm-cli，然后通过公网连接至Lidnorm时序引擎。

常用命令

- `help`：查看帮助命令。
- `connect`：连接服务器命令。
- `precision`：时间显示格式，支持rfc3339、h、m、s、 ms、u或ns。
- `exit`、`quit`、`ctrl+d`：退出当前Lindorm实例的连接。

操作步骤

1. 下载Lindorm-cli。您可以根据不同的客户端下载Lindorm-cli，下载链接如下表所示。

客户端	下载链接
Linux	lindorm-cli for linux
Mac	lindorm-cli for mac
Windows	lindorm-cli for windows-x64

2. 解压Lindorm-cli压缩包。
3. 获取连接信息并执行下述命令连接Lindorm时序引擎。
   - 如果客户端部署在Linux上，请执行以下语句。

     ```
     ./lindorm-cli -url http://<host:port> -username <用户名> -password <密码> -database <目标数据库名>
     ```

   - 如果客户端部署在Windows，请双击打开**lindorm-cli.exe**，并执行以下语句。

     ```
     connect jdbc:lindorm:tsdb:url=http://<host:port> user password
     ```

   > **说明**
   > - `host:port`：Lindorm时序引擎的连接地址和端口（例如：`ld-bp17j28j2y7pm****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242`）。获取方法参见查看连接地址。不过这里获取的连接地址和端口是不带`http://`前缀的，但参数`url`中指定的地址必须带上`http://`前缀，否则连接会失败。
   > - **用户名**：Lindorm实例的用户名。如果时序引擎的账号管理功能未启用，用户名和密码无需设置。
   > - **密码**：Lindorm实例用户名对应的密码，如果您忘记密码，可以通过Lindorm宽表引擎的集群管理系统修改密码，具体操作请参见修改用户密码。
   > - **目标数据库名**：配置使用lindorm-cli访问的数据库。默认数据库名为default，如果没有指定，则默认访问`default`数据库。可以使用`use <数据库名>`语句切换至目标数据库。
   > - 在以交互式方式启动lindorm-cli时，即使指定的`url`，`username`，`password`等参数有误也并不会影响进入lindorm-cli的界面，因为只有在lindorm-cli交互式界面下执行第一条语句时才会真正去连接时序引擎。

   返回如下结果：

   ```
   Connected to http://ld-bp17j28j2y7pm****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242
   lindorm-cli version: 1.0.15
   ```

4. 访问Lindorm时序引擎的语法示例如下。

i. 执行以下语句创建时序数据表。
创建一个名为sensor的时序数据表。其中两个Tag分别为device_id和region，Tag的数据类型为VARCHAR。两个Field命名为temperature和humidity，数据类型分别为DOUBLE、BIGINT。

```
CREATE TABLE sensor (
    device_id VARCHAR TAG,
    region VARCHAR TAG,
    time TIMESTAMP,
    temperature DOUBLE,
    humidity BIGINT,
    PRIMARY KEY(device_id));
```

> **说明**
> - 创建时序数据表时建议指定PRIMARY KEY，通常以数据源唯一标识作为PRIMARY KEY，例如物联网场景设备ID、车联网场景车辆唯一识、监控场景的应用ID或者 `ip:port` 等。
> - 时间戳列的列名必须为time，表示数据的时间，单位为毫秒（ms）。
> - 如果PRIMARY KEY相同，并且时间戳列相同，那么数据被认为是同一条数据，后写入的数据会覆盖先写入的数据。

ii. 执行以下语句查看目标数据库中的时序数据表。

```
show tables;
```

查询结果如下：

```
columnName typeName
---------- --------
default    sensor
1 rows in set (28 ms)
```

iii. 执行以下语句查看时序数据表的字段信息。

```
describe table sensor;
```

查询结果如下：

```
columnName  typeName
----------  --------
device_id   VARCHAR
region      VARCHAR
time        TIMESTAMP
temperature DOUBLE
humidity    BIGINT
5 rows in set (22 ms)
```

iv. 执行以下语句在时序数据表中写入数据。

- 单条依次写入

```
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1260','north-cn','
2021-04-22 15:33:00',12.1,45);
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1260','north-cn','
2021-04-22 15:33:10',13.2,47);
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1260','north-cn','
2021-04-22 15:33:20',10.6,46);
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1261','south-cn','
2021-04-22 15:33:00',18.1,44);
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1261','south-cn','
2021-04-22 15:33:10',19.7,44);
```

- 批量写入

```
insert into sensor (device_id, region, time, temperature, humidity)
values ('F07A1260','north-cn','2021-04-22 15:33:00',12.1,45),
       ('F07A1260','north-cn','2021-04-22 15:33:10',13.2,47),
       ('F07A1260','north-cn','2021-04-22 15:33:20',10.6,46),
       ('F07A1261','south-cn','2021-04-22 15:33:00',18.1,44),
       ('F07A1261','south-cn','2021-04-22 15:33:10',19.7,44);
```

v. 执行以下语句在时序数据表中按条件查询数据。

- 条件查询
  查询设备 `F07A1260` 在时间范围 `2021-04-22 15:33:00` 和 `2021-04-22 15:33:20` 之间的数据：

```
select device_id,region,time,temperature,humidity from sensor where device_id = 'F07A1260' and tim
e >= '2021-04-22 15:33:00' and time <= '2021-04-22 15:33:20';
```

  查询结果如下：

```
device_id region   time                      temperature humidity
--------- ------   ----                      ----------- --------
F07A1260  north-cn 2021-04-22T15:33:00+08:00 12.1        45
F07A1260  north-cn 2021-04-22T15:33:10+08:00 13.2        47
F07A1260  north-cn 2021-04-22T15:33:20+08:00 10.6        46
3 rows in set (41 ms)
```

- 降采样查询
  查询设备 `F07A1260` 在时间范围 `2021-04-22 15:33:00` 和 `2021-04-22 15:33:20` 之间的温度，按照20秒降采样后取最大值：

```
select device_id,region,time,max(temperature) as max_temperature from sensor where device_id = 'F0
7A1260' and time >= '2021-04-22 15:33:00' and time <= '2021-04-22 15:33:20' sample by 20s;
```

  查询结果如下：

```
device_id region   time                      max_temperature
--------- ------   ----                      ---------------
F07A1260  north-cn 2021-04-22T15:33:00+08:00 13.2
F07A1260  north-cn 2021-04-22T15:33:20+08:00 10.6
2 rows in set (33 ms)
```

- 聚合查询
  计算所有区域的最大温度：

```
select region,max(temperature) as max_temperature from sensor where time >= '2021-04-22 15:33:00'
and time <= '2021-04-22 15:33:20' group by region;
```

  查询结果如下：

```
region   max_temperature
------   ---------------
north-cn 13.2
south-cn 19.7
2 rows in set (76 ms)
```

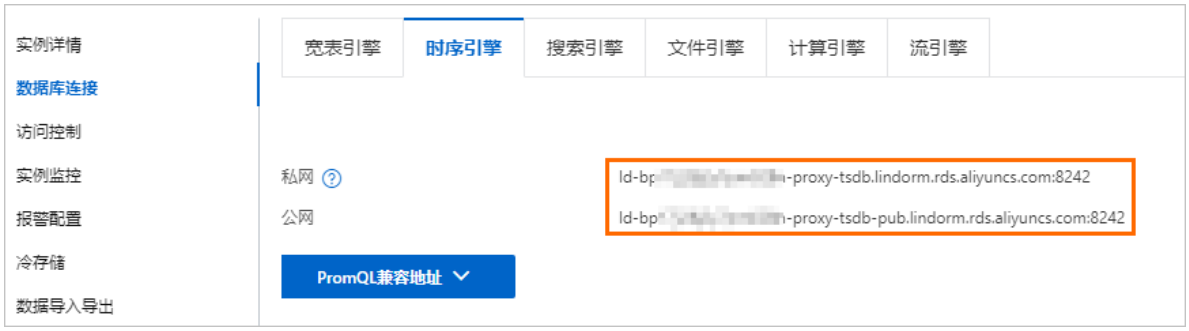
8.5.3. 通过JDBC Driver连接并访问Lindorm时序引擎

本文介绍通过Lindorm JDBC Driver连接并访问时序引擎的具体操作。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已将客户端IP地址添加至Lindorm白名单，具体操作请参见设置白名单。

- 已获取云原生多模数据库Lindorm时序引擎的连接地址，具体操作请参见查看连接地址。



操作步骤

1. 通过以下两种方式安装Lindorm JDBC Driver依赖。
   - 手动安装。
     在本地自行下载JAR包集成JDBC Driver，下载链接为：Lindorm-all-client。选择需要安装的版本，以2.0.14为例，下载lindorm-all-client-2.0.14.jar包。
   - 通过Maven下载。
     如果在Maven项目中集成JDBC Driver，创建Project并在pom.xml中添加以下依赖配置，具体内容如下：

     ```
     <dependency>
         <groupId>com.aliyun.lindorm</groupId>
         <artifactId>lindorm-all-client</artifactId>
         <version>2.0.14</version>
     </dependency>
     ```

     > **说明**
     > lindorm-all-client的版本号根据需求填写。

2. 访问Lindorm时序引擎。完整的代码示例如下。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.SQLException;

class Test {
    public static void main(String[] args) {

        String url = "jdbc:lindorm:tsdb:url=http://<host:port>";
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(url);
            Statement stmt = conn.createStatement();

            //创建时序数据表，默认访问 default database
            stmt.execute("CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time TIMESTAMP,t
emperature DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");

            //单条写入数据
            //stmt.execute("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F
07A1260','north-cn','2021-04-22 15:33:00',12.1,45)");
            //stmt.execute("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F
07A1260','north-cn','2021-04-22 15:33:10',13.2,47)");
            //stmt.execute("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F
07A1260','north-cn','2021-04-22 15:33:20',10.6,46)");
            //stmt.execute("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F
07A1261','south-cn','2021-04-22 15:33:00',18.1,44)");
```

```
            //stmt.execute("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F
07A1261','south-cn','2021-04-22 15:33:10',19.7,44)");

            //批量写入数据
            stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F0
7A1260','north-cn','2021-04-22 15:33:00',12.1,45)");
            stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F0
7A1260','north-cn','2021-04-22 15:33:10',13.2,47)");
            stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F0
7A1260','north-cn','2021-04-22 15:33:20',10.6,46)");
            stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F0
7A1261','south-cn','2021-04-22 15:33:00',18.1,44)");
            stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F0
7A1261','south-cn','2021-04-22 15:33:10',19.7,44)");
            stmt.executeBatch();
            stmt.clearBatch();

            //查询数据，强烈建议指定时间范围减少数据扫描
            ResultSet rs = stmt.executeQuery("select device_id, region,time,temperature,humidity from se
nsor where time >= '2021-04-22 15:33:00' and time <= '2021-04-22 15:33:20'");
            while (rs.next()) {
                String device_id = rs.getString("device_id");
                String region = rs.getString("region");
                Long time = rs.getLong("time");
                Double temperature = rs.getDouble("temperature");
                Long humidity = rs.getLong("humidity");
                System.out.printf("%s %s %d %f %d\n", device_id, region, time, temperature, humidity);
            }
        } catch (SQLException e) {
            // 异常处理需要结合实际业务逻辑编写
            e.printStackTrace();
        } finally {
            try {
                if (conn != null) {
                    conn.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

说明

- `host:port`：Lindorm时序引擎的连接地址和端口，例如：`ld-bp17j28j2y7pm****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242`。
- `jdbc:lindorm:tsdb:url=http://<host:port>`：表示JDBC的连接地址，具体语法和连接参数说明请参见JDBC的连接地址说明。
- JDBC Driver访问时序引擎时支持的API接口和对应的方法请参见支持的API接口和方法。
- Lindorm时序引擎支持的SQL语法请参见SQL语法。

8.5.4. 通过Java Native SDK连接并使用Lindorm时序引擎

本文介绍通过Java Native SDK连接并使用Lindorm时序引擎的方法。

前提条件

- 云原生多模数据库Lindorm实例已开通时序引擎。

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 时序引擎版本为3.4.7及以上版本，查看和升级时序引擎版本的方法请参见时序引擎版本说明。
- 已将客户端IP地址添加至Lindorm白名单，具体操作请参见设置白名单。
- 已获取云原生多模数据库Lindorm时序引擎的连接地址，获取方法请参见获取连接串。

准备工作

通过Java Native SDK连接Lindorm时序引擎前，需要安装Java Native SDK。以1.0.0版本为例，您可以通过以下三种方式安装Java Native SDK：

- （推荐）在Maven项目中使用Lindorm TSDB Java SDK。在*pom.xml*文件的 `dependencies` 中添加以下依赖项。

```
<dependency>
  <groupId>com.aliyun.lindorm</groupId>
  <artifactId>lindorm-tsdb-client</artifactId>
  <version>1.0.0</version>
</dependency>
```

> **说明** Lindorm TSDB Java SDK提供了一个基于Maven的示例工程，您可以直接下载示例工程并在本地编译和运行，也可以以示例工程为基础开发您的项目工程。

- 在Eclipse项目中导入JAR包。
  i. 下载Java SDK开发包。
  ii. 解压下载的Java SDK开发包。
  iii. 将解压后JAR包添加至Eclipse项目中。
    a. 在Eclipse中打开您的项目，右键单击该项目，选择**Properties**。
    b. 在弹出的对话框中，单击**Java Build Path > Libraries > Add JARs**，选择解压后的*lindorm-tsdb-client-1.0.0.jar*和*lib*文件中的JAR包。
    c. 单击**Apply and Close**。
- 在IntelliJ IDEA项目中导入JAR包。
  i. 下载Java SDK开发包。
  ii. 解压下载的Java SDK开发包。
  iii. 将解压后JAR包添加至IntelliJ IDEA项目中。
    a. 在IntelliJ IDEA中打开您的项目，在菜单栏单击**File > Project Structure**。
    b. 在**Project Structure**对话框的左边选择**Project Structure > Modules**。
    c. 单击右边+，选择**JARs or directories**。
    d. 在弹出的对话框中，选择解压后的*lindorm-tsdb-client-1.0.0.jar*和*lib*文件中的JAR包，并单击**OK**。
    e. 单击**Apply**。
    f. 单击**OK**。

> **说明**
> - Lindorm时序引擎的Java Native SDK各版本可以通过Maven中央仓库获取，更多信息请参见Maven Repository。
> - Lindorm时序引擎的Java Native SDK各版本说明请参见版本说明。

操作步骤

1. 创建数据库实例。新建LindormTSDBClient时，需要指定Lindorm时序引擎的连接地址，获取方法请参见获取连接串。

```
String url = "http://ld-bp17j28j2y7pm****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242";
// LindormTSDBClient线程安全，可以重复使用，无需频繁创建和销毁
ClientOptions options = ClientOptions.newBuilder(url).build();
LindormTSDBClient lindormTSDBClient = LindormTSDBFactory.connect(options);
```

2. 创建数据库demo和时序表sensor。关于创建数据库和时序表的SQL语句说明，请参见CREATE DATABASE和CREATE TABLE。

```
lindormTSDBClient.execute("CREATE DATABASE demo");
lindormTSDBClient.execute("demo","CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time BIG
INT,temperature DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");
```

3. 在表中写入数据。

> **说明** 默认情况下，为了提高写入数据的性能，LindormTSDBClient通过异步攒批的方式进行数据写入。如果需要通过同步的方式进行数据写入，可以调用 `write()` 方法返回的 `CompletableFuture<WriteResult>` 的 `join()` 方法。

```
int numRecords = 10;
List<Record> records = new ArrayList<>(numRecords);
long currentTime = System.currentTimeMillis();
for (int i = 0; i < numRecords; i++) {
    Record record = Record
        .table("sensor")
        .time(currentTime + i * 1000)
        .tag("device_id", "F07A1260")
        .tag("region", "north-cn")
        .addField("temperature", 12.1 + i)
        .addField("humidity", 45.0 + i)
        .build();
    records.add(record);
}
CompletableFuture<WriteResult> future = lindormTSDBClient.write("demo", records);
// 处理异步写入结果
future.whenComplete((r, ex) -> {
    // 处理写入失败
    if (ex != null) {
        System.out.println("Failed to write.");
        if (ex instanceof LindormTSDBException) {
            LindormTSDBException e = (LindormTSDBException) ex;
            System.out.println("Caught an LindormTSDBException, which means your request made it to Lind
orm TSDB, "
                               + "but was rejected with an error response for some reason.");
            System.out.println("Error Code: " + e.getCode());
            System.out.println("SQL State:  " + e.getSqlstate());
            System.out.println("Error Message: " + e.getMessage());
        }  else {
            ex.printStackTrace();
        }
    } else  {
        System.out.println("Write successfully.");
    }
});
// 这里作为示例，简单同步处理写入结果
System.out.println(future.join());
```

4. 查询时序表的数据。关于查询操作的SQL语句说明，请参见基本查询。

```
String sql = "select * from sensor limit 10";
ResultSet resultSet = lindormTSDBClient.query("demo", sql);
try {
    // 处理查询结果
    QueryResult result = null;
    // 查询结果使用分批的方式返回，默认每批1000行
    // 当resultSet的next()方法返回为null，表示已经读取完所有的查询结果
    while ((result = resultSet.next()) != null) {
        List<String> columns = result.getColumns();
        System.out.println("columns: " + columns);
        List<String> metadata = result.getMetadata();
        System.out.println("metadata: " + metadata);
        List<List<Object>> rows = result.getRows();
        for (int i = 0, size = rows.size(); i < size; i++) {
            List<Object> row = rows.get(i);
            System.out.println("row #" + i + " : " + row);
        }
    }
} finally {
    // 查询结束后，需确保调用ResultSet的close方法，以释放IO资源
    resultSet.close();
}
```

完整的代码示例

```
import com.aliyun.lindorm.tsdb.client.ClientOptions;
import com.aliyun.lindorm.tsdb.client.LindormTSDBClient;
import com.aliyun.lindorm.tsdb.client.LindormTSDBFactory;
import com.aliyun.lindorm.tsdb.client.exception.LindormTSDBException;
import com.aliyun.lindorm.tsdb.client.model.QueryResult;
import com.aliyun.lindorm.tsdb.client.model.Record;
import com.aliyun.lindorm.tsdb.client.model.ResultSet;
import com.aliyun.lindorm.tsdb.client.model.WriteResult;
import com.aliyun.lindorm.tsdb.client.utils.ExceptionUtils;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
public class QuickStart {
    public static void main(String[] args) {
        // 1.创建客户端实例
        String url = "http://ld-xxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242";
        // LindormTSDBClient线程安全，可以重复使用，无需频繁创建和销毁
        ClientOptions options = ClientOptions.newBuilder(url).build();
        LindormTSDBClient lindormTSDBClient = LindormTSDBFactory.connect(options);
        // 2.创建数据库demo和表sensor
        lindormTSDBClient.execute("CREATE DATABASE demo");
        lindormTSDBClient.execute("demo","CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time
BIGINT,temperature DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");
        // 3.写入数据
        int numRecords = 10;
        List<Record> records = new ArrayList<>(numRecords);
        long currentTime = System.currentTimeMillis();
        for (int i = 0; i < numRecords; i++) {
            Record record = Record
                    .table("sensor")
                    .time(currentTime + i * 1000)
                    .tag("device_id", "F07A1260")
                    .tag("region", "north-cn")
                    .addField("temperature", 12.1 + i)
                    .addField("humidity", 45.0 + i)
                    .build();
            records.add(record);
        }
        CompletableFuture<WriteResult> future = lindormTSDBClient.write("demo", records);
```

```
        // 处理异步写入结果
        future.whenComplete((r, ex) -> {
            // 处理写入失败
            if (ex != null) {
                System.out.println("Failed to write.");
                Throwable throwable = ExceptionUtils.getRootCause(ex);
                if (throwable instanceof LindormTSDBException) {
                    LindormTSDBException e = (LindormTSDBException) throwable;
                    System.out.println("Caught an LindormTSDBException, which means your request made it to
Lindorm TSDB, "
                            + "but was rejected with an error response for some reason.");
                    System.out.println("Error Code: " + e.getCode());
                    System.out.println("SQL State:  " + e.getSqlstate());
                    System.out.println("Error Message: " + e.getMessage());
                }  else {
                    throwable.printStackTrace();
                }
            } else  {
                System.out.println("Write successfully.");
            }
        });
        // 这里作为示例，简单同步等待
        System.out.println(future.join());
        // 4.查询数据
        String sql = "select * from sensor limit 10";
        ResultSet resultSet = lindormTSDBClient.query("demo", sql);
        try {
            // 处理查询结果
            QueryResult result = null;
            // 查询结果使用分批的方式返回，默认每批1000行
            // 当resultSet的next()方法返回为null，表示已经读取完所有的查询结果
            while ((result = resultSet.next()) != null) {
                List<String> columns = result.getColumns();
                System.out.println("columns: " + columns);
                List<String> metadata = result.getMetadata();
                System.out.println("metadata: " + metadata);
                List<List<Object>> rows = result.getRows();
                for (int i = 0, size = rows.size(); i < size; i++) {
                    List<Object> row = rows.get(i);
                    System.out.println("row #" + i + " : " + row);
                }
            }
        } finally {
            // 查询结束后，需确保调用ResultSet的close方法，以释放IO资源
            resultSet.close();
        }
        lindormTSDBClient.shutdown();
    }
}
```

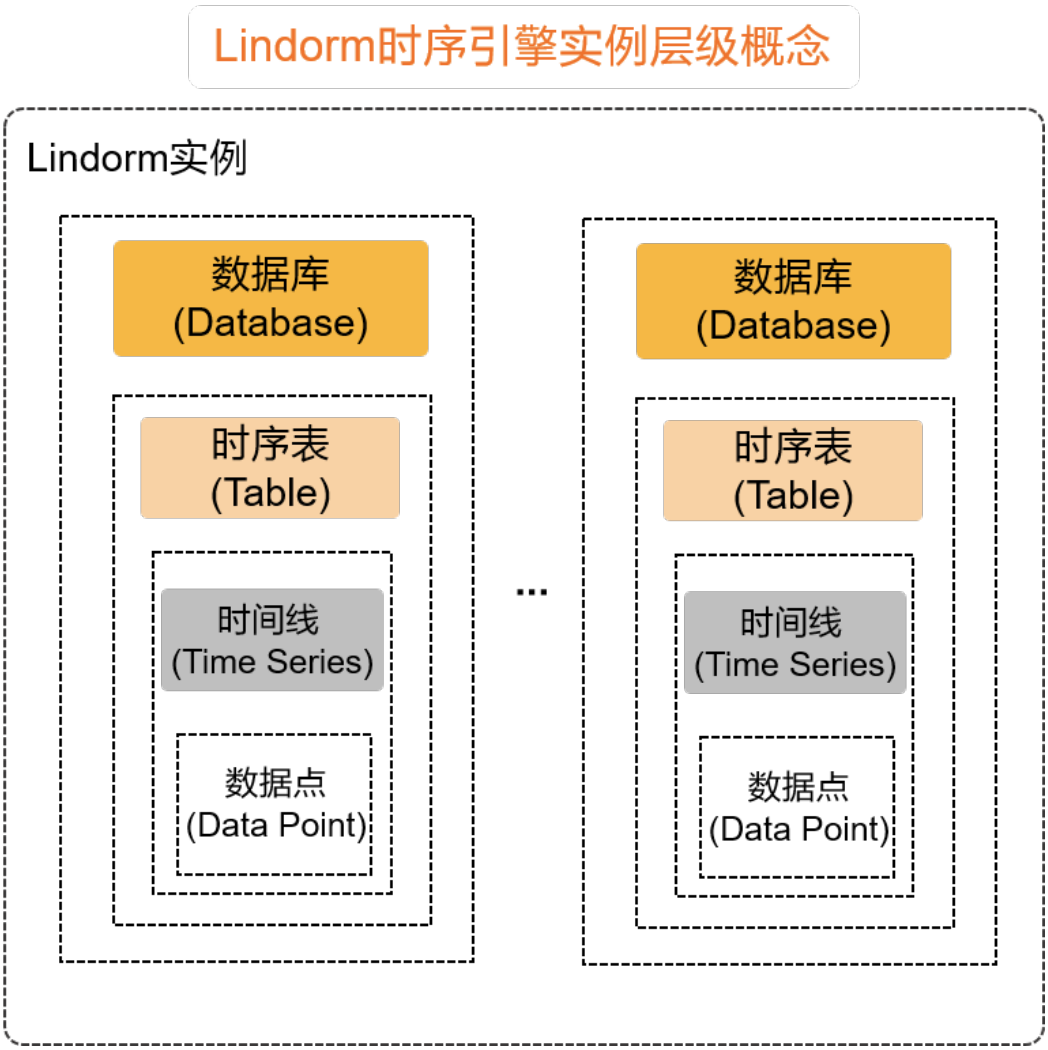
8.6. 数据管理

8.6.1. 数据库管理

本文介绍Lindorm时序引擎中的Database概念以及相关的功能。

Database的概念

每个正在运行的Lindorm时序引擎实例都管理着一个或多个数据库（简称Database）。在各个Database之间，数据会被隔离。每个Database都有若干时序数据表，结合名词解释的描述，我们可以对时序引擎的数据层级归纳为下图所示：



如果一个Lindorm时序引擎实例计划包含不相关的项目数据，在很大程度上，彼此之间是隔离的。那么在这种情况下，建议将它们放到单独的数据库中，并且调整相应的授权和访问控制用户。此外，时序引擎允许用户跨Database访问数据，前提是该用户必须拥有对目标Database的相应操作的权限。

以Database为粒度的时序数据管理功能

Database是时序表的集合，用户可以根据业务需求将拥有相同业务背景的时序表保存在一起，从而实现数据隔离。另一方面，Lindorm时序引擎的一系列功能也都是以Database为粒度提供给用户使用的。包括以下功能：

- 用户能够以Database为单位设置数据的冷热分界线。

  > 说明
  > 关于数据冷热分层的概念，可以参见时序数据的冷温热数据分层。

- 用户能够以Database为单位设置数据的存活时间（Time To Live，简称TTL）。
  当TTL被设置后，时序数据的时间戳距当前的时间差超过了TTL的时间，则这些数据将被视作过期数据，会被时序引擎清除掉。

> 注意
>
> 1. 时序领域的数据的时间戳一般分成两种概念：
>   - 事件时间：即时序数据对应的事件真实发生时的时间。时序数据元素说明中提及的时序数据模型的时间戳即是事件时间。
>   - 入库时间：即时序数据写入数据库的时间。
>
>   上述冷热分界线和TTL的能力都是基于**事件时间**进行工作。目前不支持基于入库时间设置冷热分界线或TTL。
> 2. 无论是冷数据归档还是数据到期清理工作，都是通过时序引擎的异步compaction任务执行的，因此，冷数据归档和TTL数据清理可能存在滞后的情况。
> 3. 对Database既设置了冷热分界线，又设置了TTL的情况下，已归档的冷数据的时间戳如果超过了TTL定义的存活时间，数据也会触发清理。
> 4. 对于Database设置了TTL的情况下，如果写入数据的时间戳已经超过了TTL的范围。那么写入仍然是可以成功的，但是查询时是查不到这样的数据的。

Database管理

时序引擎支持用户通过SQL对Database进行以下管理操作：

- Database的创建
  在时序引擎中创建一个新的Database。SQL语法请参见CREATE DATABASE。

  > 说明
  > 新建的Database需要单独为指定的用户赋权，用户并不会默认具有访问权限。

- Database的删除
  在时序引擎中删除一个已有的Database。SQL语法请参见DROP DATABASE。
- Database的属性修改
  在时序引擎中对一个已有Database的属性进行修改。包括冷热分界线，数据存活期间等。SQL语法请参见ALTER DATABASE。
- Database的属性查询
  在时序引擎中查看一个已有Database的属性元数据。SQL语法请参见DESCRIBE DATABASE。

在管理控制台上管理Database

在Lindorm实例的管理控制台上也可以对时序引擎的Database进行管理。可以直接在管理控制台的界面上设置Database的冷热分界线以及数据存活时间。操作界面如下图所示：



> 注意
> 目前在管理控制台上并不支持Database的创建和删除。若需要创建或删除Database，还需使用SQL来操作。

8.6.2. 用户及权限管理

本文介绍Lindorm时序引擎中访问控制机制以及如何进行用户管理及权限管理。

时序引擎的访问控制体系

Lindorm提供了一套多层次的访问控制体系，从而提供企业级应用的安全保障。访问控制体系如下所示：

- 第一层防护：实例级别的物理层面访问控制
  管理方法可参见设置白名单。
- 第二层防护：引擎级别的逻辑层面访问控制
  在引擎层面提供的基于用户认证以及权限校验为主体的访问控制。

用户认证与权限校验功能的启用

对于一个Lindorm实例的时序引擎，默认情况下**未启用**用户认证与权限校验。因此在默认情况下无论是通过哪种方式访问时序引擎，都可以不带上用户信息直接操作。
但是，对于一个投入线上生产应用的实例而言，出于安全因素考虑，强烈建议开启时序引擎的用户认证与权限校验。通过ALTER SYSTEM语句开启，具体方法如下：
通过任意方式连接到时序引擎后，执行 `ALTER SYSTEM SET USER_AUTH=TRUE;` 即可开启用户认证与权限校验功能。

> **注意**
> - 时序引擎一旦开启用户认证和权限校验，后续的所有访问都必须带上用户信息，否则将可能出现鉴权失败（包括当前执行 `ALTER SYSTEM` 语句启用用户认证和权限校验的连接）。因此，该功能开启后，已有的部分业务访问可能会失败。所以在开启该功能前，建议仔细规划并充分评估影响。
> - 用户认证与权限校验功能开启后也可通过ALTER SYSTEM语句关闭。执行语句如下：
>
>   ```
>   ALTER SYSTEM SET USER_AUTH=FALSE;
>   ```
>
> - 查看当前实例是否已经开启用户认证与权限校验功能，可通过SHOW PARAMETER语句查看。执行语句如下：
>
>   ```
>   SHOW PARAMETER USER_AUTH;
>   ```

时序引擎基于用户认证与权限校验的访问控制

当用户认证与权限校验的功能开启后，业务对时序引擎的访问都需要在连接时带上用户名和密码。同时，无论是写入还是查询操作，连接所属用户必须要具有访问目标的对应权限，操作才能成功。通过这种方式，时序引擎可以对业务访问进行严格的安全控制以及权限隔离。

在使用SQL时带上认证信息的方法

随着业务形态不同，在连接时指定用户名、密码的方式也有所差异。具体如下：

- 使用Lindorm-Cli访问时序引擎的场景
  请参见Lindorm-Cli访问的命令行示例。
- 使用JDBC构建应用的场景
  请参见URL语法与连接参数。
- 基于通用SQL接口构建应用的场景
  请参见用户认证信息指定。

用户管理

时序引擎通过SQL提供了用户管理的能力，提供的能力如下：

- 增加用户
  请参见CREATE USER。
- 删除用户
  请参见DROP USER。
- 修改用户密码
  请参见ALTER USER。
- 查看已有用户
  请参见SHOW。

> **说明**
> 实例创建后会默认创建一个初始用户。它的用户名为root，密码也是root，可以用于创建其他业务所需的用户并赋权。

> **警告**
> 强烈建议实例创建完毕后立即修改上述初始用户密码，否则可能会造成很大的安全隐患。

权限管理

时序引擎中的权限分为四种，如下所示：

权限名	含义
READ	**读权限**。当查询数据时，要求用户对查询所涉及的数据表具有读权限。
WRITE	**写权限**。当写入数据时，要求用户对查询所设计的数据表具有写权限。
ADMIN	**管理权限**。当对时序引擎中的数据对象进行管理时，需要用户对操作的数据对象具有管理权限。
SYSTEM	**系统权限**。当一些操作可能会影响整个实例的行为时，这样的操作需要用户具有系统权限。

另一方面，时序引擎可以根据不同的业务需求限定对用户权限赋予以及回收的范围。当前时序引擎支持的权限赋予和回收的粒度如下：

- GLOBAL
  全局范围的授权与撤回。
  当用户被赋予全局范围的任意种类权限后，用户则拥有了对于整个时序引擎内所有数据对象的对应操作权限。比如一个全局的READ权限将使得用户可以查询任意数据库中的任意时序数据表。
- DATABASE
  数据库范围的授权与撤回。
  当用户被赋予某个数据库的指定种类权限后，用户则拥有了对于该数据库内部所有数据对象的对应操作权限。比如某个数据库DB1的READ权限将使得用户可以查询该数据库内的所有时序数据表。

时序引擎支持通过GRANT语句对用户进行授权，同时支持通过REVOKE语句对用户已有权限进行撤回。

> 注意
> - 尝试对用户进行权限授予或撤回时，要求当前连接所属的用户本身拥有全局ADMIN权限。
> - 权限的优先级是GLOBAL>DATABASE。即如果拥有了GLOBAL的READ权限，则能查询任意Database中的时序表；但如果只是拥有特定Database的READ权限，则只能查询该Database下的时序表。
> - SYSTEM权限只能是GLOBAL范围的权限。

常用SQL所需权限

常用的SQL语句所需的权限可参见下表：

> 说明
> 下表中○表示需要的权限，×表示不需要的权限。

SQL语句	GLOBAL				DATABASE			备注
	READ	WRITE	ADMIN	SYSTEM	READ	WRITE	ADMIN	
SELECT...FROM...	○	×	×	×	○	×	×	-
INSERT INTO...VALUES...	×	○	×	×	×	○	×	-
INSERT INTO...SELECT...	○	○	×	×	○	○	×	INSERT目标表需要WRITE权限，同时SELECT源表需要READ权限。
DESCRIBE DATABASE ...	×	×	○	×	×	×	○	-
CREATE DATABASE...	×	×	○	×	×	×	×	-

SQL语句	GLOBAL				DATABASE			备注
	READ	WRITE	ADMIN	SYSTEM	READ	WRITE	ADMIN	
ALTER DATABASE...	×	×	○	×	×	×	○	拥有目标数据库的ADMIN权限也可变更。
DROP DATABASE...	×	×	○	×	×	×	○	拥有目标数据库的ADMIN权限也可删除。
SHOW DATABASES	×	×	○	×	×	×	×	-
DESCRIBE TABLE ...	○	×	×	×	○	×	×	-
CREATE TABLE...	×	×	○	×	×	×	○	-
DROP TABLE...	×	×	○	×	×	×	○	-
CREATE USER	×	×	○	○	×	×	×	-
ALTER USER...	×	×	○	○	×	×	×	-
DROP USER...	×	×	○	○	×	×	×	-
SHOW USERS	×	×	○	○	×	×	×	-
GRANT...	×	×	○	○	×	×	×	-
REVOKE...	×	×	○	○	×	×	×	-
SHOW PRIVILEGES...	×	×	○	○	×	×	×	-
ALTER SYSTEM...	×	×	×	○	×	×	×	-
SHOW PARAMETER...	×	×	×	○	×	×	×	-

访问OpenTSDB兼容API时的用户鉴权

对于部分使用OpenTSDB兼容的HTTP API访问时序引擎的应用（参见已有OpenTSDB或时序数据库TSDB应用程序迁移指引），当用户认证与权限校验功能启用时，发送到这些API的请求也需要带上用户信息。

- 访问OpenTSDB兼容API时指定用户信息的方法
  指定方法类似调用通用SQL接口，通过在HTTP请求头中加入基于Basic Authentication协议编码的用户认证信息，详细可参见用户认证信息指定。

  > **说明**
  > 对于使用旧版TSDB的SDK的业务，用户名和密码可以通过创建 `TSDBConfig` 对象时调用 `basicAuth` 方法来指定。

- 访问OpenTSDB兼容API所需的权限
  调用这些API时都需要用户具备GLOBAL权限，各个API所需的权限如下所示：

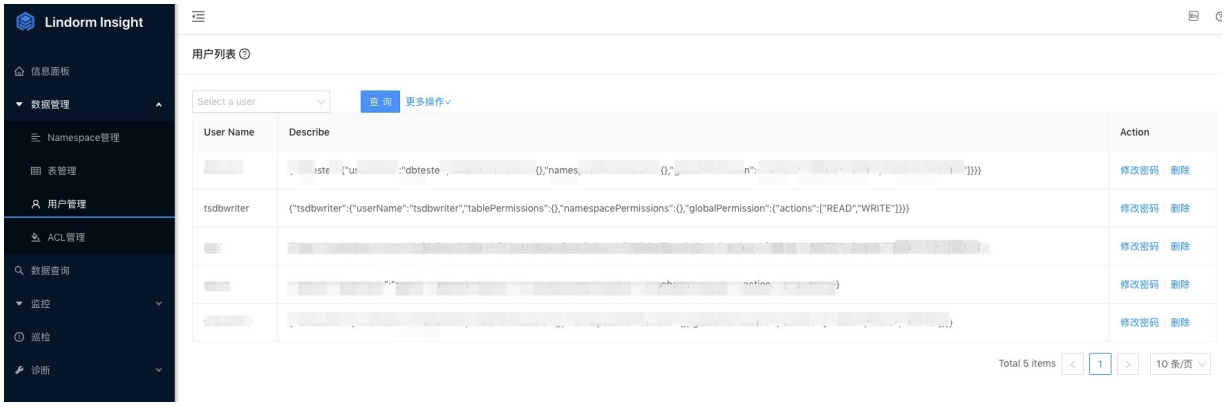
  > **说明**
  > 下表中○表示需要的权限，×表示不需要的权限。

API种类	GLOBAL WRITE	GLOBAL READ	GLOBAL ADMIN
/api/put	○	×	×

API种类	GLOBAL WRITE	GLOBAL READ	GLOBAL ADMIN
/api/query	×	○	×
/api/query/last	×	○	×
/api/mput	○	×	×
/api/mquery	×	○	×
/api/query/mlast	×	○	×
/api/suggest	×	○	×
/api/dump_meta	×	○	×
/api/search/lookup	×	○	×
/api/ttl	×	×	○
/api/truncate	×	×	○

与宽表引擎访问控制的融合

时序引擎的用户信息与权限数据与宽表引擎可以通用，因此如果创建的Lindorm实例中也包含宽表引擎的话，在时序引擎中创建的用户也可以用于访问宽表引擎。此时，时序引擎的用户也可在宽表引擎的集群管理工具中进行查阅和管理，如下图所示：



同理，宽表引擎中创建的用户同样也可用于访问时序引擎，可以通过在时序引擎中对宽表引擎的用户赋予特定时序引擎Database的权限来实现。

> **注意**
>
> - 宽表引擎的用户可能被授予TRASH权限，该权限目前在时序引擎中并无实际意义。
> - 如果宽表引擎中的一个Namespace与时序引擎的Database同名，且某个用户拥有该Namespace的权限，则该用户默认就拥有了对时序引擎同名Database的相应访问权限。反之亦然。

8.6.3. 冷数据归档

本文介绍时序引擎的冷数据归档功能的使用方法。

功能概要

在时序数据库持续写入的场景下，早先写入的时序数据会随着时间推移，查询频度逐渐降低。这样的数据通常被称为“冷数据”，与之相对的概念被称为“热数据”。随着冷数据体量的不断变大，存储成本也会水涨船高。所以降低冷数据存储成本，提升热数据读取性能，对于使用时序引擎的企业用户具有现实意义。
在Lindorm时序引擎中，随着时间的不断推移，数据呈现出很明显的冷温热分层。其相应的概念如下：

- 热数据
  最近写入的时序数据。这些时序数据的时间戳往往距离当前很近，在时序应用的场景中会被频繁查询。

- 温数据
  已写入一段时间的数据。这些时序数据查询的频率相对较低。
- 冷数据
  已经写入了很长时间的数据，通常可以视作归档数据。在时序应用的场景中，这些数据的数据量往往非常庞大，但是极少用于查询。

基于冷温热数据的数据量以及查询频度的特点，Lindorm时序引擎实现了对时序数据冷温热分层存储。如下所示：



对于数据量最大的冷数据，时序引擎会自动根据用户设置的基于业务时间戳的冷热分界线自动将冷数据归档到冷存储中。当需要查询冷数据时，用户可以无感知地按正常查询的方式进行冷数据查询。

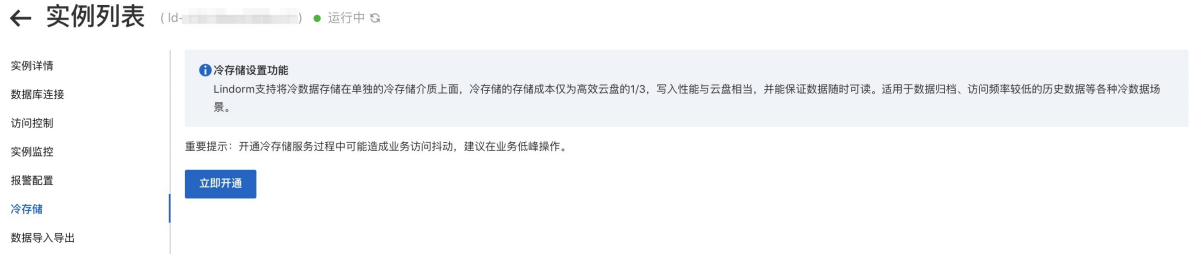
使用前提

时序引擎实例创建后本身就具备了对热数据和温数据的分层管理能力。但是对于冷数据，如果需要启用上文所说的冷数据自动归档到冷存储介质的能力，则需要预先在Lindorm实例层面开通冷存储。
开通冷存储的方法如下：

- 申购Lindorm实例时，在购买页面决定是否购买冷存储并决定冷存储的大小。



- 创建实例时没有购买冷存的情况下，也可在实例创建后通过Lindorm实例的管理控制台开通冷存储。管理控制台上开通冷存储的入口如下所示。



开通了冷存储后，即可通过时序引擎设置冷热分界线来控制冷数据自动归档至冷存储介质了。

配置冷热分界线

冷热分界线的含义

冷热分界线是一个在业务层面定义区分数据冷热的分界线，由用户指定。即用户根据数据存储量和查询时间覆盖范围，确定多长时间之前的数据需要转移到冷存储。
在时序引擎中，冷热分界线的设置单位是**天**。含义是某条时序数据的时间戳如果满足下述条件，则该数据被视作冷数据。

- 冷数据的判定标准

```
当前时间 - 数据的时间戳 >= 冷热分界线(天)
```

设置冷热分界线的方法

有以下两个方式设置冷热分界线：

1. 通过SQL语句在Database的创建或修改中设置冷热分界线。

   > **说明**
   > 通过SQL设置冷热分界线的方法可参见 CREATE DATABASE 或 ALTER DATABASE。

2. 通过时序引擎的管理控制台设置Database的冷热分界线。

   > **说明**
   > 通过管理控制台设置冷热分界线的方法可参见 在管理控制台上管理Database 。

查询冷数据

用户在查询数据时不会意识到被查询数据是否是冷数据，通过正常的SQL语句查询即可，无需刻意区分。
但需注意的是，如果被查询的数据已经归档到冷存储介质上，那么查询的延迟将会增加。

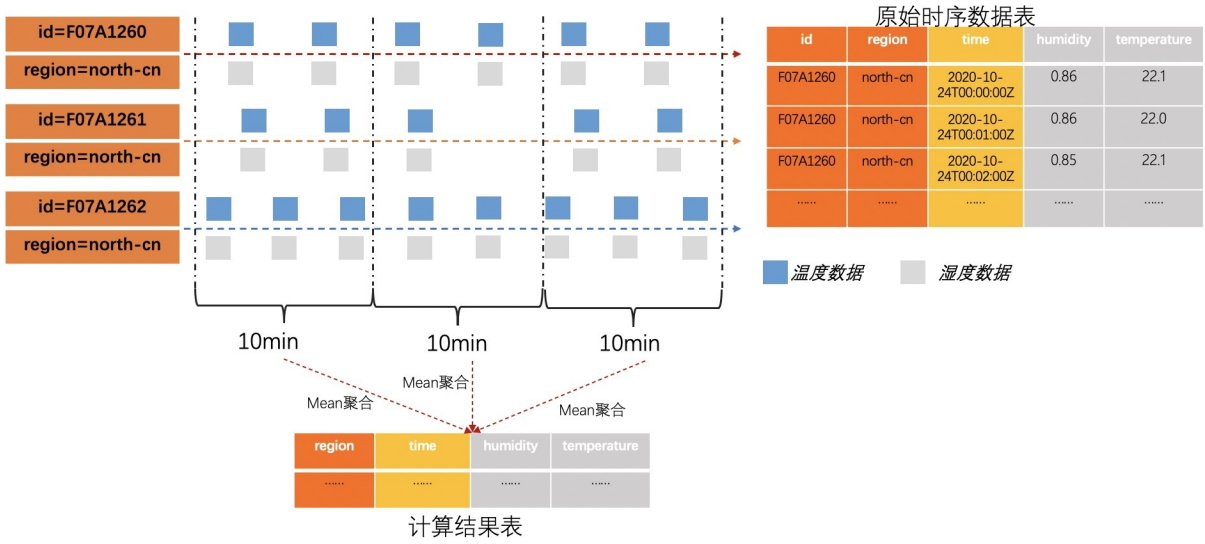
8.6.4. 数据的持续查询

本文介绍Lindorm时序引擎中的持续查询的概念以及相关功能。

持续查询的概念

持续查询（Continuous Queries，简称CQ）是时序引擎内部自动定期执行的时序查询。

在时序应用的场景下，对于依照时间推进顺序写入的实时数据，用户有时会希望每隔一段固定时间，就能够按照一定的查询条件对该时间范围内的时序数据做一次计算（例如：对该时间范围内的数据求一次聚合计算），并将计算结果另行保存下来。该应用场景示例如下所示：



持续查询所针对的便是此类应用场景。在时序引擎中，持续查询提供了一种简化的流计算能力，在时间窗口结束后对窗口内的数据进行计算。

使用持续查询

持续查询的管理

Lindorm时序引擎提供SQL能力用于管理数据库中的持续查询。

- 持续查询的创建
  在指定的Database下创建一个持续查询。SQL语法可参见CREATE CONTINUOUS QUERY。
- 持续查询的删除
  从指定的Database下删除一个已存在的持续查询。SQL语法可参见DROP CONTINUOUS QUERY。
- 持续查询信息的展示
  查询已有的持续查询元数据。SQL语法可参见SHOW CONTINUOUS QUERIES。

持续查询的窗口调度

持续查询运用的是时序引擎节点本地的时间戳来确定何时执行计算，并以此为参照推移计算窗口。持续查询会在每一个时间窗口关闭后、下一个窗口开始的时候对前一窗口内的数据进行计算。例如，一个持续计算指定的窗口间隔为1小时的话，该持续查询会在每个小时开始的时候执行一次计算。在晚上20:00触发计算时，计算的数据所属时间范围是 `[19:00:00, 20:00:00)` 。

> 注意
>
> - 尽管持续查询会在其定义的时间间隔结束后立刻开始计算，但根据其查询计算内容的不同，计算的结果可能也需要花一些时间才能保存到目标表中。延时取决于查询内容以及实例的实时负载。
> - 持续查询的结果精度取决于原始数据的有序性。当数据写入原始数据表时存在乱序的情况，即下一个窗口开始后仍然有上一窗口的数据零星写入。那么这部分数据并不会参与到上一窗口的计算中。
> - 数据库删除时，所有属于该数据库的持续查询都会被直接删除。

持续查询的场景与示例

持续计算的常用场景

- 数据的长期保存与成本平衡
  对于数据量较大的场景，存储成本会成为用户关心的问题。因此可以运用持续查询并结合数据库管理的TTL或冷数据归档能力对长期数据进行有选择的降成本。
- 通过持续计算提升查询性能
  对于某些查询较大时间范围的降采样查询或者跨时间线的聚合查询，如果直接对原始数据进行查询，耗时可能会很高。此时可以通过连续查询将结果进行预计算，之后在需要的时候直接查询计算后的结果表，其性能可能会更好。对于查询吞吐也可以有效提升。

示例

下面这个示例，展示了如何通过持续查询加上数据库的TTL来实现对数据进行高效的长期存储。
比如用户原始数据量较大，可以保存最近一个月，降精度后可以保留几年的数据。下面这个实例中，原始数据采样周期为1秒，降采样到分钟粒度进行长期存储。

1. 将原始数据保存在db_sensor_month中，保存周期为30天，创建数据库：

```
CREATE DATABASE db_sensor_month WITH (ttl=30);
```

2. 创建另一个数据库用于长期保存持续计算后的数据，数据库创建如下：

```
CREATE DATABASE db_sensor_year WITH (ttl=365);
```

3. 创建用于保存计算结果的表：

```
CREATE TABLE db_sensor_year.sensor (
    device_id VARCHAR TAG,
    region VARCHAR TAG,
    time TIMESTAMP,
    temperature DOUBLE,
    humidity BIGINT);
```

4. 创建持续查询（执行降采样计算）如下：

```
CREATE CONTINUOUS QUERY db_sensor_year.my_cq WITH (INTERVAL='1m')
AS
  INSERT into db_sensor_year.sensor(time, temperature, humidity, device_id,region)
  SELECT time, mean(temperature) AS temperature, mean(humidity) humidity, device_id, region
  FROM db_sensor_month.sensor
  SAMPLE BY 60s;
```

8.7. 开发指南

8.7.1. 时序引擎应用开发简介

本文面向应用开发者概要介绍了如何基于Lindorm时序引擎开发应用程序。
Lindorm提供了以SQL为主的多种形态接口方便用户编写访问时序引擎的应用程序，对于应用开发者，通常有以下两种方式可用于开发访问Lindorm时序引擎的应用程序：

- 对于Java语言的应用开发者，可以借助Lindorm JDBC Driver基于JDBC标准API连接时序引擎并执行SQL语句。
- 对于其他语言的应用开发者，提供了一个基于HTTP的API (api/v2/sql)方便应用程序发送SQL语句与时序引擎进行交互。

Lindorm时序引擎提供的SQL语法明细，可参考 SQL语法参考 。

此外，对于先前基于 OpenTSDB 或 时序数据库TSDB 开发的应用程序，Lindorm时序引擎也提供了一系列能力兼容的 API 方便这些应用迁移至Lindorm时序引擎。

JDBC应用开发指引

Lindorm JDBC Driver基于JDBC API 4.1 标准进行实现，在应用开发中安装并使用它连接Lindorm时序引擎的基本方法可以参见 通过JDBC Driver构建Lindorm时序引擎应用。

> **说明**
> Lindorm JDBC Driver支持的API 以及功能范围 可以参考 JDBC API Reference 。

基于通用SQL API的应用开发指引

Lindorm时序引擎提供了一个名为 `api/v2/sql` 的HTTP API便于任意编程语言可直接发送SQL语句读写时序数据。开发者可以根据所用语言和类库遵照该HTTP API的接口定义拼装 HTTP请求来发送SQL语句访问时序引擎。

> **说明**
> 关于 `api/v2/sql` 的详细用法可参见 /api/v2/sql 应用指南 。

已有 OpenTSDB 或 时序数据库TSDB 应用程序迁移指引

Lindorm时序引擎兼容了大部分开源OpenTSDB的HTTP API，同时对于云上时序数据库TSDB产品的能力也有90%以上的兼容。原先基于 OpenTSDB 的HTTP API开发的应用程序或者基于云上时序数据库TSDB的应用程序只需少量修改甚至是无修改即可直接适用于 Lindorm时序引擎。
对于原有OpenTSDB的应用或者时序数据库TSDB的应用，可以按照以下步骤来进行应用程序的迁移：

1. 修改应用程序中的连接字符串设置，将连接的目标从原有的OpenTSDB或者时序数据库TSDB改为Lindorm时序引擎。Lindorm时序引擎连接字符串的获取方法可参见 获取连接串。
2. 结合Lindorm时序引擎的能力修改应用程序
   i. 如果是基于HTTP API 开发的应用，结合 Lindorm时序引擎支持的HTTP API 对使用了有差异的API的代码进行修改。Lindorm 时序引擎支持的HTTP API 基本兼容 时序时空数据库TSDB 的API。详细可参见 时序时空TSDB的HTTP API概览。
   ii. 如果是基于时序数据库TSDB的SDK开发的Java应用，基本可无缝迁移，因为Lindorm时序引擎兼容了时序数据库TSDB的90%以上的能力。
3. 应用代码连接到Lindorm时序引擎调试并测试无误后即可上线运行。

> 注意
> 1. 时序引擎对于 OpenTSDB/时序数据库TSDB 风格的API 写入的数据并不会真正的创建时序数据表，而是将这些数据按照其metric虚拟出逻辑上的时序数据表。如果需要通过SQL来查询这些数据，则需要手动建表。
> 2. 在时间、成本允许的情况下，我们**推荐基于SQL重写 OpenTSDB/时序数据库TSDB应用中的数据库访问逻辑**(含数据写入与查询)。我们基于SQL提供了更多的时序特性的功能表达和更好的性能优化。

8.7.2. 关于时序数据的Schema约束

本文面向应用开发者概要介绍了Lindorm时序引擎中对于时序数据模型的Schema约束，以及应用开发中与之相关的注意事项。

Lindorm时序模型的Schema

Lindorm时序引擎基于对时序领域中业务数据模型的理解与洞察，抽象出了时序数据表这一概念。因此，开发者在开发基于Lindorm时序引擎的应用程序时，可以类比传统关系型数据库中的关系表来访问时序数据。

但是，时序数据模型本质上与关系型数据模型还是有所区别的。无论是在APM领域，还是IoT领域，或者是更偏传统的工业领域，时序数据所表现出的数据模型更像是一个半结构化数据——有其遵循结构化定义的一面、也有可灵活扩展的一面，而不是完全等同于关系型数据库的纯结构化。而在实际应用中，随着设备的不断改进、升级、迭代，设备上报的数据模型变化可能也会很频繁，包括设备上报字段的增加，字段类型的变更等等。

尽管在Lindorm时序引擎中，提供了ALTER TABLE语句用于变更时序数据表。但是在一部分时序应用场景下，如果对表的Schema约束严格如关系型数据库，那么则有可能使应用开发变得更加困难。

在此背景下，Lindorm时序引擎提供了更加灵活的Schema约束策略供用户使用。

Schema约束的含义

通常所说的Schema约束往往是体现在数据写入层面。即写入的字段是否与预先定义的表结构中包含的列一致；写入的字段数据类型是否与预先定义的数据类型一致或兼容，甚至写入的表是否存在等等。在传统的关系型数据库中，这些约束往往是非常严格地执行。
在Lindorm时序引擎中所述的Schema约束指的是以下行为：

- 数据写入的目标表存在与否的校验
- 数据写入的TAG或FIELD是否被预先定义的校验
- 数据写入的数据类型与预定义是否一致的校验
- 数据查询时字段返回类型的一致性保障

支持的Schema约束策略

Lindorm时序引擎目前提供3种Schema约束策略，分别是**强约束**、**弱约束**和**无约束**。用户可结合具体的业务需求进行选择。

- **强约束**
  与传统关系型数据库类似，时序引擎会严格依据预先定义的表结构对写入数据的表名、字段名、类型进行校验。不符合即出错。
- **弱约束**
  相较**强约束**存在以下灵活性：
  - 写入数据的所属表不存在时引擎不会报错，而是会自动创建对应的表。
  - 写入数据的新增一个标签或字段时，引擎不会校验报错，而是会在对应的时序数据表中自动添加一个TAG列或FIELD列。
  - 写入数据的相同字段的数据类型发生变化时，引擎也不会校验，而是会将最新一次写入的数据类型作为该FIELD列的数据类型。

  > 注意
  > i. 弱约束策略并不意味着完全没有校验。比如当尝试给一个TAG列写入一个非字符串，或给时间戳列写入非时间戳类型时，仍然会触发校验失败。
  > ii. 在弱约束策略下同一个字段的类型发生变换时，后续再查询该字段的数据会依据最新写入的数据类型来进行查询。因此对于历史写入的数据，可能会出现查询不到的情况。

- **无约束**
  不做任何约束。代价是无法直接通过SQL查询写入的数据。

> 注意
>
> i. 无约束这一Schema约束策略并不能主动选择。出于兼容旧版TSDB接口而保留的一种策略。
>
> ii. 无约束策略下写入的数据无法直接通过SQL进行查询。

数据写入方式与Schema约束的选择

在Lindorm时序引擎中，数据写入时的Schema约束策略与数据的写入方式是密切相关的，以下表格将基于 时序引擎应用开发简介 中介绍的数据访问方式以及数据生态 中介绍的开源协议兼容的写入方式分别概括一下不同的写入方式下其对应的Schema约束策略：

写入方式	Schema约束策略	备注
通过JDBC API写入数据	**强约束**	-
通过Java Native SDK入门写入数据	可选。**强约束** 或 **弱约束** 均支持。	策略配置方法参见Java Native SDK使用手册。不配置时的默认约束是 **弱约束**。
通过api/v2/sql直接用INSERT写入数据	**强约束**	-
通过 InfluxDB®行协议 写入数据	**弱约束**	-

8.7.3. 通过JDBC Driver连接并访问时序引擎

本文介绍通过Lindorm JDBC Driver连接并访问时序引擎的具体操作，以及JDBC Driver访问时序引擎时支持的API接口和方法。

背景信息

Lindorm时序引擎的JDBC Driver遵循JDBC 4.1规范定义的接口。由于时序引擎自身是一个时序数据库，所以JDBC Driver中提供的实际上是JDBC 1.0定义的基础功能集中的一个子集。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已将客户端IP地址添加至Lindorm白名单，具体操作请参见设置白名单。
- 已获取云原生多模数据库Lindorm时序引擎的连接地址，具体操作请参见查看连接地址。

操作步骤

1. 通过以下两种方式安装Lindorm JDBC Driver依赖。
   - 手动安装。
     在本地自行下载JAR包集成JDBC Driver，下载链接为：Lindorm-all-client。选择需要安装的版本，以2.0.14为例，下载lindorm-all-client-2.0.14.jar包。
   - 通过Maven下载。
     如果在Maven项目中集成JDBC Driver，创建Project并在pom.xml中添加以下依赖配置，具体内容如下：

     ```
     <dependency>
         <groupId>com.aliyun.lindorm</groupId>
         <artifactId>lindorm-all-client</artifactId>
         <version>2.0.14</version>
     </dependency>
     ```

     > 说明
     > lindorm-all-client的版本号根据需求填写。
2. 访问Lindorm时序引擎。完整的代码示例请参考访问示例。

i. 建立Lindorm时序引擎和数据之间的连接。

```
String url = "jdbc:lindorm:tsdb:url=http://<host:port>";
Connection conn = DriverManager.getConnection(url);
```

> 说明
>
> - `host:port`：Lindorm时序引擎的连接地址和端口，例如：`ld-bp17j28j2y7pm****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242`。
> - `jdbc:lindorm:tsdb:url=http://<host:port>`：表示JDBC的连接地址，具体语法和连接参数说明请参见JDBC的连接地址说明。

ii. 创建Lindorm时序数据库和表。

```
Statement stmt = conn.createStatement();

stmt.execute("CREATE DATABASE test");
stmt.execute("USE test");
stmt.execute("CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time TIMESTAMP,temperatu
re DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");
```

iii. 在表sensor中插入数据。

```
//PreparedStatement 批量写入数据（推荐使用）
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO sensor (device_id, region, time, temper
ature, humidity) VALUES (?, ?, ?, ?, ?)");
int batchSize = 100;
long ts = System.currentTimeMillis();

for (int i = 0; i < batchSize; i++) {

     //按INSERT语句中的列参数顺序依次绑定参数
     pstmt.setString(1, "F07A1260");
     pstmt.setString(2, "north-cn");
     pstmt.setLong(3, ts + i * 10000);
     pstmt.setDouble(4, 12.1);
     pstmt.setLong(5, 45);

     pstmt.addBatch();
}

pstmt.executeBatch();
pstmt.clearBatch();

//批量写入数据
Statement stmt = conn.createStatement();
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1260',
'north-cn',1619076780000,12.1,45)");
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1260',
'north-cn',1619076790000,13.2,47)");
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1260',
'north-cn',1619076800000,10.6,46)");
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1261',
'south-cn',1619076780000,18.1,44)");
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1261',
'south-cn',1619076790000,19.7,44)");
stmt.executeBatch();
stmt.clearBatch();

//单条写入数据（不推荐使用）
Statement stmt = conn.createStatement();
stmt.execute("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1260','
north-cn',1619076780000,12.1,45)");
```

iv. 查询表sensor中的数据。

```
ResultSet rs = stmt.executeQuery("select device_id, region,time,temperature,humidity from sensor whe
re time >= 1619076780000 and time <= 1619076800000");

while (rs.next()) {
    String device_id = rs.getString("device_id");
    String region = rs.getString("region");
    Long time = rs.getLong("time");
    Double temperature = rs.getDouble("temperature");
    Long humidity = rs.getLong("humidity");
    System.out.printf("%s %s %d %f %d\n", device_id, region, time, temperature, humidity);
}
```

支持的API接口和方法

JDBC Driver访问时序引擎时支持的API接口和对应的方法如下表所示。

> **说明**
> 下表未列出的API接口部分兼容上层框架，会返回默认值。如果您有其他问题，请提交工单。

接口	支持的方法
Connection	• `close()` • `createStatement(int, int)` • `createStatement(int, int, int)` • `getSchema()` • `prepareStatement(String, int, int)` • `prepareStatement(String, int, int, int)` • `setSchema(String)` **说明** 由于Lindorm JDBC Driver的Connection接口实现不是线程安全的，所以建议不要在多个线程中共用同一个Connection对象。
Statement	• `addBatch(String)` • `close()` • `execute(String)` • `executeQuery(String)` • `getMoreResult` • `isClosed()` • `setFetchSize(int)`
ResultSet	• `addBatch()` • `clearBatch()` • `close()` • `execute()` • `executeBatch()` • `executeQuery()` • `getParameterNumber()` • `setNull(int)` • `setXXX (int, T)` **说明** Lindorm时序引擎支持的数据类型请参见数据类型。

接口	支持的方法
PreparedStatement	• `close()` • `getStatement()` • `next()` • `wasNull()` • `getXXX(int)` **说明** Lindorm时序引擎支持的数据类型请参见数据类型。

JDBC的连接地址说明

JDBC的连接地址是应用程序基于Connection接口建立数据库连接时的重要依据，通常用于指定数据库连接的协议，实际是通过JDBC Driver实现。

JDBC的连接地址语法如下所示：

```
jdbc:lindorm:tsdb:url=http://${host:port}[;${connection_parameter}=${connection_value}]...
```

说明

`host:port`：Lindorm时序引擎的连接地址和端口，例如：`ld-bp17j28j2y7pm****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242`。

JDBC的连接地址中支持的连接参数和说明如下表：

参数	说明
user	Lindorm实例的用户名。用于指定连接时序引擎时进行用户认证的用户名。 **说明** 当引擎的用户鉴权功能未启用时，用户名可以不指定。
password	用户名对应的密码。用于指定连接时序引擎时进行用户认证的密码。如果您忘记了密码，可以通过Lindorm宽表引擎的集群管理系统修改密码，具体操作请参见修改用户密码。 **说明** 当引擎的用户鉴权功能未启用时，密码可以不指定。
database	连接访问的Database。如果该参数未指定，默认访问`default`数据库。
lindorm.tsdb.driver.connect.timeout	连接时序引擎服务端的超时时间，单位为ms（毫秒）。默认值60秒（即60000毫秒）。
lindorm.tsdb.driver.socket.timeout	Socket读写间隔超时时间，单位为ms（毫秒）。默认值为-1，表示无超时。

> **注意**
> 服务端为了提高资源利用率，在连接Lindorm客户端空闲90s后，服务端会主动断开连接。如果还需要使用该连接，会出现 `com.aliyun.lindorm.client.shaded.org.apache.calcite.avatica.http.ConnectionDisconnectedException` 报错，重新建立连接即可解决。

使用Druid连接池访问时序引擎

使用Java开发访问时序引擎的应用程序时，通常建议通过JDBC Driver连接Lindorm时序引擎，但是JDBC访问逻辑的基础是需要创建Connection对象。如果每次连接时序引擎都创建一个新的Connection对象会带来很大的开销。为了降低开销，解决方法是使用连接池来执行连接的创建和管理。
目前可以选择的连接池有很多种，以下介绍开源连接池Alibaba Druid的使用步骤。

1. 在Maven项目中添加Druid连接池的依赖。打开客户端，创建Project并在pom.xml中配置Maven依赖，具体内容如下：

```
<dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid</artifactId>
      <version>1.2.6</version>
    </dependency>
```

2. 在Maven项目中添加Lindorm JDBC Driver依赖，具体请参见操作步骤中的步骤1。
3. 执行以下代码创建DataSource对象，为DataSource设置Lindorm时序引擎的JDBC连接地址。

```
dataSource = new DruidDataSource();
        try {
            dataSource.setDriver(DriverManager.getDriver("jdbc:lindorm:tsdb:url=http://<host:port>"));
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
```

> **说明**
> `host:port`：Lindorm时序引擎的连接地址和端口，例如：`ld-bp17j28j2y7pm****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242`。

4. 基于DataSource创建JDBC Connection并访问Lindorm时序引擎。完整的代码如下：

```
import com.alibaba.druid.pool.DruidDataSource;
import java.sql.*;

public class App {
    private DruidDataSource dataSource;

    public void setUp() {
        dataSource = new DruidDataSource();
        try {
            dataSource.setDriver(DriverManager.getDriver("jdbc:lindorm:tsdb:url=http://ld-bp17j28j2y7pm*
***-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242"));
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        dataSource.setUrl("jdbc:lindorm:tsdb:url=http://ld-bp17j28j2y7pm****-proxy-tsdb-pub.lindorm.rds.
aliyuncs.com:8242");
        dataSource.setValidationQuery("select 1");
        dataSource.setMaxActive(1);
        dataSource.setUsername("username");
        dataSource.setPassword("****");
        dataSource.setKeepAlive(true);
        dataSource.setMinEvictableIdleTimeMillis(80);
        dataSource.setMaxWait(10000);
        dataSource.init();

    }

    public void close() {
        dataSource.close();
    }

    public void testQuery() {
        try (Connection conn = this.dataSource.getConnection()) {
            try (Statement stmt = conn.createStatement()) {
                ResultSet rs = stmt.executeQuery("select * from sensor");
                long count = 0;
                while (rs.next()) {
                    count++;
                }
                System.out.println("count=" + count + " ");
            } catch (RuntimeException e) {
                e.printStackTrace();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        } catch (RuntimeException e) {
            System.out.println("connection failed.");
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main( String[] args ) {
        App app = new App();
        app.setUp();
        app.testQuery();
        app.close();
    }
}
```

8.7.4. SQL接口应用指南

本文介绍时序引擎提供的基于HTTP协议的通用SQL接口的定义和用法。
对于任意非Java编程语言的应用开发，可以使用下述API直接向时序引擎发送SQL语句；对于Java语言的应用开发，建议基于JDBC Driver构建时序引擎的应用程序。

请求路径与方法

请求路径	请求方法	描述
/api/v2/sql	POST	发送并执行SQL语句

请求内容

请求上述 API 时，直接将 SQL语句写入HTTP请求的请求体即可，并且建议在请求头设置 `Content-Type` 为 `text/plain` 。

以Python代码为例，在Python代码中向 `/api/v2/sql` 传送一条SQL语句的做法示例如下：

```
import urllib2

reqheaders={'Content-Type':'text/plain'}
req=urllib2.Request('http://ld-xxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/v2/sql', 'SHOW TABLES',
reqheaders)
response=urllib2.urlopen(req).read()
```

> **注意**
> 时序引擎遵循SQL-92标准，支持将半角分号(;)作为SQL语句的终止符。但是在调用 /api/v2/sql 传递SQL语句时，不能在结尾加上分号，否则会引发解析错误。

支持的查询参数

/api/v2/sql 支持的URL查询参数如下所示：

名称	描述
database	SQL执行时的默认Database。 当指定的SQL中写入或查询的时序表名未显式指定所属数据库名时，时序引擎默认会在该参数指定的database中搜寻操作对象的表名。未指定该查询参数时，引擎会默认在名为 `default` 的Database中搜寻表名。
chunked	是否流式返回结果数据。默认为false。 当指定为true时，查询结果数据会以多个JSON块的方式返回，每个JSON块最多包含chunk_size行数据。在解析结果数据时，只需要逐个解析JSON块即可。每个JSON块的结构参考响应内容部分描述。
chunk_size	指定流式返回结果时每次返回的最大行数。chunked=true时生效，默认为1000。

用户认证信息指定

当Lindorm时序引擎开启用户鉴权时，通过 `/api/v2/sql` 发送SQL请求时需要向HTTP请求头中填入用户认证信息。目前 `/api/v2/sql` 遵循的是 `BASIC AUTH` 认证方式，编码后的认证信息需要指定在HTTP请求头的 **`Authorization`** 字段中。

基本认证凭据的格式如下：

```
BASIC {BASE64编码的认证信息}
```

其中需要被BASE64编码的**认证信息**明文为 `${用户名}:${对应的用户密码}` （中间一定以冒号相隔）。

> **说明**
> 各编程语言编码并填充BASIC AUTH认证信息的方法，请查询所用语言的相关类库的使用文档，不在此罗列。

响应内容

查询成功的HTTP响应码为200，响应内容为JSON格式数据。说明如下：

名称	类型	描述
columns	Array	字符串数组 表示返回结果集中列名。
metadata	Array	字符串数组 表示返回结果集中每一列的数据类型名称。可能返回的数据类型请参见数据类型。
rows	Array	数组的数组 表示返回结果集的行的集合。其中的每一个数组代表着一行数据，每行数据中的具体数据与columns中返回的列一一对应。

SQL执行出错时的响应消息中的HTTP响应码为400，响应消息体是一个JSON格式数据，其包含的字段说明如下：

名称	类型	描述
code	int	错误码
sqlstate	String	sqlstate
message	String	错误详情

> 说明
> 关于错误码对应的具体错误，请参考Lindorm TSDB的常见错误码参考文档。

示例

以下示例通过常用的curl命令，展示了如果通过该HTTP接口对Lindorm时序引擎执行SQL。

- 创建名为DB1的Database。

```
curl -X POST http://ld-xxxxxxxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/v2/sql -d 'CREATE DATABA
SE DB1'
```

- 在DB1下创建名为SENSOR的时序数据表。

```
curl -X POST http://ld-xxxxxxxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/v2/sql?database=DB1 -d '
CREATE TABLE SENSOR (device_id VARCHAR TAG,region VARCHAR TAG,time TIMESTAMP,temperature DOUBLE,humidity D
OUBLE)'
```

或者

```
curl -X POST http://ld-xxxxxxxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/v2/sql -d 'CREATE TABLE
DB1.SENSOR (device_id VARCHAR TAG,region VARCHAR TAG,time TIMESTAMP,temperature DOUBLE,humidity DOUBLE)'
```

- 当用户鉴权开关开启时，以一个名为 `tsdbuser` 的用户查询上述SENSOR时序表。

```
curl -X POST -u tsdbuser:password http://ld-xxxxxxxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/v2/
sql?database=DB1 -d 'SELECT device_id, region, time, MAX(temperature) as max_t FROM SENSOR WHERE time >= 1
619076780000 AND time <= 1619076800000 SAMPLE BY 20s'
```

> 说明
> 本示例中假定已事先对用户tsdbuser赋予了时序表SENSOR的相关权限。

8.7.5. Java Native SDK

8.7.5.1. 快速入门

本文介绍通过Java Native SDK连接并使用Lindorm时序引擎的方法。

前提条件

- 云原生多模数据库Lindorm实例已开通时序引擎。
- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 时序引擎版本为3.4.7及以上版本，查看和升级时序引擎版本的方法请参见时序引擎版本说明。
- 已将客户端IP地址添加至Lindorm白名单，具体操作请参见设置白名单。
- 已获取云原生多模数据库Lindorm时序引擎的连接地址，获取方法请参见获取连接串。

准备工作

通过Java Native SDK连接Lindorm时序引擎前，需要安装Java Native SDK。以1.0.0版本为例，您可以通过以下三种方式安装Java Native SDK：

- （推荐）在Maven项目中使用Lindorm TSDB Java SDK。在*pom.xml*文件的 `dependencies` 中添加以下依赖项。

```
<dependency>
  <groupId>com.aliyun.lindorm</groupId>
  <artifactId>lindorm-tsdb-client</artifactId>
  <version>1.0.0</version>
</dependency>
```

> **说明** Lindorm TSDB Java SDK提供了一个基于Maven的示例工程，您可以直接下载示例工程并在本地编译和运行，也可以以示例工程为基础开发您的项目工程。

- 在Eclipse项目中导入JAR包。
  i. 下载Java SDK开发包。
  ii. 解压下载的Java SDK开发包。
  iii. 将解压后JAR包添加至Eclipse项目中。
    a. 在Eclipse中打开您的项目，右键单击该项目，选择**Properties**。
    b. 在弹出的对话框中，单击**Java Build Path > Libraries > Add JARs**，选择解压后的*lindorm-tsdb-client-1.0.0.jar*和*lib*文件中的JAR包。
    c. 单击**Apply and Close**。
- 在IntelliJ IDEA项目中导入JAR包。
  i. 下载Java SDK开发包。
  ii. 解压下载的Java SDK开发包。
  iii. 将解压后JAR包添加至IntelliJ IDEA项目中。
    a. 在IntelliJ IDEA中打开您的项目，在菜单栏单击**File > Project Structure**。
    b. 在**Project Structure**对话框的左边选择**Project Structure > Modules**。
    c. 单击右边+，选择**JARs or directories**。
    d. 在弹出的对话框中，选择解压后的*lindorm-tsdb-client-1.0.0.jar*和*lib*文件中的JAR包，并单击**OK**。
    e. 单击**Apply**。
    f. 单击**OK**。

> **说明**
> - Lindorm时序引擎的Java Native SDK各版本可以通过Maven中央仓库获取，更多信息请参见Maven Repository。
> - Lindorm时序引擎的Java Native SDK各版本说明请参见版本说明。

操作步骤

1. 创建数据库实例。新建LindormTSDBClient时，需要指定Lindorm时序引擎的连接地址，获取方法请参见获取连接串。

```
String url = "http://ld-bp17j28j2y7pm****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242";
// LindormTSDBClient线程安全，可以重复使用，无需频繁创建和销毁
ClientOptions options = ClientOptions.newBuilder(url).build();
LindormTSDBClient lindormTSDBClient = LindormTSDBFactory.connect(options);
```

2. 创建数据库demo和时序表sensor。关于创建数据库和时序表的SQL语句说明，请参见CREATE DATABASE和CREATE TABLE。

```
lindormTSDBClient.execute("CREATE DATABASE demo");
lindormTSDBClient.execute("demo","CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time BIG
INT,temperature DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");
```

3. 在表中写入数据。

> **说明** 默认情况下，为了提高写入数据的性能，LindormTSDBClient通过异步攒批的方式进行数据写入。如果需要通过同步的方式进行数据写入，可以调用 `write()` 方法返回的 `CompletableFuture<WriteResult>` 的 `join()` 方法。

```
int numRecords = 10;
List<Record> records = new ArrayList<>(numRecords);
long currentTime = System.currentTimeMillis();
for (int i = 0; i < numRecords; i++) {
    Record record = Record
        .table("sensor")
        .time(currentTime + i * 1000)
        .tag("device_id", "F07A1260")
        .tag("region", "north-cn")
        .addField("temperature", 12.1 + i)
        .addField("humidity", 45.0 + i)
        .build();
    records.add(record);
}
CompletableFuture<WriteResult> future = lindormTSDBClient.write("demo", records);
// 处理异步写入结果
future.whenComplete((r, ex) -> {
    // 处理写入失败
    if (ex != null) {
        System.out.println("Failed to write.");
        if (ex instanceof LindormTSDBException) {
            LindormTSDBException e = (LindormTSDBException) ex;
            System.out.println("Caught an LindormTSDBException, which means your request made it to Lind
orm TSDB, "
                               + "but was rejected with an error response for some reason.");
            System.out.println("Error Code: " + e.getCode());
            System.out.println("SQL State:  " + e.getSqlstate());
            System.out.println("Error Message: " + e.getMessage());
        }  else {
            ex.printStackTrace();
        }
    } else  {
        System.out.println("Write successfully.");
    }
});
// 这里作为示例，简单同步处理写入结果
System.out.println(future.join());
```

4. 查询时序表的数据。关于查询操作的SQL语句说明，请参见基本查询。

```
String sql = "select * from sensor limit 10";
ResultSet resultSet = lindormTSDBClient.query("demo", sql);
try {
    // 处理查询结果
    QueryResult result = null;
    // 查询结果使用分批的方式返回，默认每批1000行
    // 当resultSet的next()方法返回为null，表示已经读取完所有的查询结果
    while ((result = resultSet.next()) != null) {
        List<String> columns = result.getColumns();
        System.out.println("columns: " + columns);
        List<String> metadata = result.getMetadata();
        System.out.println("metadata: " + metadata);
        List<List<Object>> rows = result.getRows();
        for (int i = 0, size = rows.size(); i < size; i++) {
            List<Object> row = rows.get(i);
            System.out.println("row #" + i + " : " + row);
        }
    }
} finally {
    // 查询结束后，需确保调用ResultSet的close方法，以释放IO资源
    resultSet.close();
}
```

完整的代码示例

```
import com.aliyun.lindorm.tsdb.client.ClientOptions;
import com.aliyun.lindorm.tsdb.client.LindormTSDBClient;
import com.aliyun.lindorm.tsdb.client.LindormTSDBFactory;
import com.aliyun.lindorm.tsdb.client.exception.LindormTSDBException;
import com.aliyun.lindorm.tsdb.client.model.QueryResult;
import com.aliyun.lindorm.tsdb.client.model.Record;
import com.aliyun.lindorm.tsdb.client.model.ResultSet;
import com.aliyun.lindorm.tsdb.client.model.WriteResult;
import com.aliyun.lindorm.tsdb.client.utils.ExceptionUtils;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
public class QuickStart {
    public static void main(String[] args) {
        // 1.创建客户端实例
        String url = "http://ld-xxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242";
        // LindormTSDBClient线程安全，可以重复使用，无需频繁创建和销毁
        ClientOptions options = ClientOptions.newBuilder(url).build();
        LindormTSDBClient lindormTSDBClient = LindormTSDBFactory.connect(options);
        // 2.创建数据库demo和表sensor
        lindormTSDBClient.execute("CREATE DATABASE demo");
        lindormTSDBClient.execute("demo","CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time
BIGINT,temperature DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");
        // 3.写入数据
        int numRecords = 10;
        List<Record> records = new ArrayList<>(numRecords);
        long currentTime = System.currentTimeMillis();
        for (int i = 0; i < numRecords; i++) {
            Record record = Record
                    .table("sensor")
                    .time(currentTime + i * 1000)
                    .tag("device_id", "F07A1260")
                    .tag("region", "north-cn")
                    .addField("temperature", 12.1 + i)
                    .addField("humidity", 45.0 + i)
                    .build();
            records.add(record);
        }
        CompletableFuture<WriteResult> future = lindormTSDBClient.write("demo", records);
```

```
        // 处理异步写入结果
        future.whenComplete((r, ex) -> {
            // 处理写入失败
            if (ex != null) {
                System.out.println("Failed to write.");
                Throwable throwable = ExceptionUtils.getRootCause(ex);
                if (throwable instanceof LindormTSDBException) {
                    LindormTSDBException e = (LindormTSDBException) throwable;
                    System.out.println("Caught an LindormTSDBException, which means your request made it to
Lindorm TSDB, "
                            + "but was rejected with an error response for some reason.");
                    System.out.println("Error Code: " + e.getCode());
                    System.out.println("SQL State:  " + e.getSqlstate());
                    System.out.println("Error Message: " + e.getMessage());
                }  else {
                    throwable.printStackTrace();
                }
            } else  {
                System.out.println("Write successfully.");
            }
        });
        // 这里作为示例，简单同步等待
        System.out.println(future.join());
        // 4.查询数据
        String sql = "select * from sensor limit 10";
        ResultSet resultSet = lindormTSDBClient.query("demo", sql);
        try {
            // 处理查询结果
            QueryResult result = null;
            // 查询结果使用分批的方式返回，默认每批1000行
            // 当resultSet的next()方法返回为null，表示已经读取完所有的查询结果
            while ((result = resultSet.next()) != null) {
                List<String> columns = result.getColumns();
                System.out.println("columns: " + columns);
                List<String> metadata = result.getMetadata();
                System.out.println("metadata: " + metadata);
                List<List<Object>> rows = result.getRows();
                for (int i = 0, size = rows.size(); i < size; i++) {
                    List<Object> row = rows.get(i);
                    System.out.println("row #" + i + " : " + row);
                }
            }
        } finally {
            // 查询结束后，需确保调用ResultSet的close方法，以释放IO资源
            resultSet.close();
        }
        lindormTSDBClient.shutdown();
    }
}
```

8.7.5.2. 客户端初始化

LindormTSDBClient是Lindorm时序引擎的Java客户端，用于读写Lindorm时序引擎的数据。使用Java SDK发起Lindorm时序引擎读写请求前，您需要初始化一个LindormTSDBClient实例，并根据需要修改ClientOptions的默认配置项。

新建LindormTSDBClient

新建LindormTSDBClient时，需要指定数据库连接串URL。Lindorm时序引擎连接字符串的获取方法可参见获取连接串。在创建LindormTSDBClient时除了指定必要的URL等参数外，还可以配置连接超时时间、攒批写入的批次大小、最大重试次数等可选参数。创建LindormTSDBClient的示例如下所示。

```
// Lindorm TSDB实例控制台获取的连接串，注意连接串最后不要添加额外的'/'等字符
String url = "http://ld-xxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242";
// 用户名
String username = "username";
// 用户密码
String password = "password";

// ClientOptions参数的具体含义和默认值，可以参考'配置LindormTSDBClient'小节的内容
ClientOptions options = ClientOptions
                // 必选
                .newBuilder(url)
                // 若实例未开启鉴权，则可选；否则必选
                .setUsername(username)
                // 若实例未开启鉴权，则可选；否则必选
                .setPassword(password)
                // 可选
                .setNumBatchThreads(1)
                // 可选
                .setMaxRetries(3)
                // 可选
                .setRetryBackoffMs(1000)
                 // 可选
                .setRequestTimeoutMs(30000)
                 // 可选
                .setConnectTimeoutMs(30000)
                 // 可选
                .setMaxWaitTimeMs(300)
                 // 可选
                .setBatchSize(500)
                .build();

// 创建LindormTSDBClient实例，LindormTSDBClient线程安全，可以重复使用，无需频繁创建和销毁
LindormTSDBClient client = LindormTSDBFactory.connect(options);

// 关闭LindormTSDBClient实例
client.shutdown();
```

配置LindormTSDBClient

ClientOptions是LindormTSDBClient的配置类，您可通过此类来配置连接超时、异步攒批写入的批次大小、最大重试次数等参数。可设置的参数如下：

参数	是否必填	描述	方法
`url`	必填	Lindorm时序引擎的连接地址。	ClientOptions.Builder()
`username`	若Lindorm时序引擎开启鉴权，则必须指定。	用户名	ClientOptions.Builder#setUsername
`password`	若Lindorm时序引擎开启鉴权，则必须指定。	密码	ClientOptions.Builder#setPassword
`numBatchThreads`	可选	攒批异步写入的线程数，默认为1。	ClientOptions.Builder#setNumBatchThreads
`maxRetries`	可选	当写入遇到服务端内部错误或者网络错误时最大重试次数，默认值为3。	ClientOptions.Builder#setMaxRetries
`retryBackoffMs`	可选	写入失败后重试等待时间，单位为毫秒，默认值为1000，即失败后等待1000毫秒后再尝试发送数据。	ClientOptions.Builder#setRetryBackoffMs

requestTimeoutMs	可选	请求超时时间，单位为毫秒，默认值为90000。	ClientOptions.Builder#setRequestTimeoutMs
connectTimeoutMs	可选	连接超时时间，单位为毫秒，默认值为90000。	ClientOptions.Builder#setConnectTimeoutMs
maxWaitTimeMs	可选	数据点攒批时在队列中的最大等待时间，单位毫秒，默认值为300。	ClientOptions.Builder#setMaxWaitTimeMs
batchSize	可选	攒批写入时每次向服务端发送的批次大小，默认值为500。	ClientOptions.Builder#setBatchSize
defaultDatabase	可选	读写默认数据库，默认值为default；Lindorm时序引擎实例会默认创建一个名为default的数据库。	ClientOptions.Builder#setDefaultDatabase
queryChunkSize	可选	查询时的默认每批返回的行数大小，默认值为1000。	ClientOptions.Builder#setQueryChunkSize
schemaPolicy	可选	表的元数据约束策略，可设置以下Strong和Weak两种策略。默认为Strong策略。 说明 关于SchemaPolicy可详细参见关于时序数据的Schema约束。	ClientOptions.Builder#setSchemaPolicy

8.7.5.3. 接口说明

本文介绍Lindorm TSDB SDK常用接口的使用方法。
Lindorm TSDB SDK的接口主要分成管理接口、写入接口以及查询接口等3类。

管理接口

Lindorm时序引擎支持使用SQL进行DDL、DCL操作，这里管理接口相关的示例是直接通过提交SQL实现对应的DDL和DCL操作。在Lindorm TSDB SDK中，支持以下两个重载方法向Lindorm时序引擎提交SQL实现DDL和DCL操作。

```
// 直接提交SQL
Result execute(String sql);

// 在指定的数据库database下进行DDL等操作，比如在指定的database下创建表
Result execute(String database, String sql);
```

其中，Result对象表示SQL的执行结果，比如提交数据库的列表语句（"SHOW DATABASES"）后，可以通过Result获取到已经存在的数据库列表。Result对象有columns、metadata、rows等3个字段，其中，columns为结果的列名集合，metadata为对应列的数据类型集合，rows为结果的按行返回的集合。

```
public class Result {
    private List<String> columns;
    private List<String> metadata;
    private List<List<Object>> rows;

    ....
 }
```

数据库和表管理示例

数据库和表的DDL操作主要包括创建、删除、查看等。代码示例如下：

```
// 1.查看数据库列表
String showDatabase = "show databases";
Result result = lindormTSDBClient.execute(showDatabase);
System.out.println("before create, db list: " +  result.getRows().stream().map(e -> (String) e.get(0)).colle
ct(Collectors.toList()));
```

```
// 2.创建数据库
String createDatabase = "create database demo";
result = lindormTSDBClient.execute(createDatabase);
System.out.println("create database:" + result.isSuccessful());

// 3.查看数据库列表
result = lindormTSDBClient.execute(showDatabase);
System.out.println("after create, db list: " +  result.getRows().stream().map(e -> (String) e.get(0)).collec
t(Collectors.toList()));

String database = "demo";

// 4.查看表列表
String showTables = "show tables";
result = lindormTSDBClient.execute(database, showTables);
System.out.println("before create, table list: " +  result.getRows().stream().map(e -> (String) e.get(0)).co
llect(Collectors.toList()));

// 5.创建表
String createTable = "CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time BIGINT,temperature
DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))";
result = lindormTSDBClient.execute(database, createTable);
System.out.println("create table: " + result.isSuccessful());

// 6.查看列表
result = lindormTSDBClient.execute(database, showTables);
System.out.println("after create, table list: " +  result.getRows().stream().map(e -> (String) e.get(0)).col
lect(Collectors.toList()));

// 7.描述表详情
String describeTable = "describe table sensor";
result = lindormTSDBClient.execute(database, describeTable);
System.out.println("------------ describe table --------------------");
List<String> columns = result.getColumns();
System.out.println("columns: " + columns);
List<String> metadata = result.getMetadata();
System.out.println("metadata: " + metadata);
List<List<Object>> rows = result.getRows();
for (int i = 0, size = rows.size(); i < size; i++) {
    List<Object> row = rows.get(i);
    System.out.println("column #" + i + " : " + row);
}

System.out.println("------------ describe table --------------------");
// 8.删除表
String dropTable = "drop table sensor";
result = lindormTSDBClient.execute(database, dropTable);
System.out.println("drop table: " + result.isSuccessful());

// 9.查看表列表
result = lindormTSDBClient.execute(database, showTables);
System.out.println("after drop, table list: " +  result.getRows().stream().map(e -> (String) e.get(0)).colle
ct(Collectors.toList()));

// 10.删除数据列表
String dropDatabase = "drop database demo";
result = lindormTSDBClient.execute(dropDatabase);
System.out.println("drop database:" + result.isSuccessful());

// 11.查看数据库列表
result = lindormTSDBClient.execute(showDatabase);
System.out.println("after drop, db list : " +  result.getRows().stream().map(e -> (String) e.get(0)).collect
(Collectors.toList()));
```

运行结果示例：

```
before create, db list: [default]
create database:true
after create, db list: [default, demo]
before create, table list: []
create table: true
after create, table list: [sensor]
------------ describe table -------------------
columns: [columnName, typeName, columnKind]
metadata: [VARCHAR, VARCHAR, VARCHAR]
column #0 : [device_id, VARCHAR, TAG]
column #1 : [region, VARCHAR, TAG]
column #2 : [time, TIMESTAMP, TIMESTAMP]
column #3 : [temperature, DOUBLE, FIELD]
column #4 : [humidity, DOUBLE, FIELD]
------------ describe table -------------------
drop table: true
after drop, table list: []
drop database:true
after drop, db list : [default]
```

> **说明**
> 对于这里没有列举的其他管理相关的操作，比如连续查询的创建、删除等DDL操作，都可以根据Lindorm时序引擎的对应的SQL语法说明构造对应的SQL语句来实现。关于Lindorm时序引擎的SQL语句说明，可以参考SQL语法参考。

写入接口

Lindorm TSDB SDK的写入接口为多个相同名称但参数不同的write方法，但根据处理写入结果方式，可以分成两类，分别为使用CompletableFuture<WriteResult>处理写入结果的接口和使用回调函数Callback处理异步写入结果的接口。
默认情况下，LindormTSDBClient为提高写入性能，使用异步攒批的方式进行数据写入。若需要同步写入，只需要调用下write方法返回的CompletableFuture<WriteResult>的join方法即可。

写入记录

Lindorm TSDB SDK使用Record对象表示表中的一行写入记录。Record需要指定表名、时间戳、标签和量测值。其中，时间戳和标签将用于构建索引。

```
Record record = Record
    // 指定表名
    .table("sensor")
    // 指定时间戳，单位为毫秒
    .time(currentTime)
    // 指定标签
    .tag("device_id", "F07A1260")
    .tag("region", "north-cn")
    // 指定量测值
    .addField("temperature", 12.1)
    .addField("humidity", 45.0)
    .build();
```

使用CompletableFuture处理写入结果的接口

- 每次只提交一行记录。

```
// 写入默认数据库
CompletableFuture<WriteResult> future = lindormTSDBClient.write(record);

// 写入指定的数据库
String database = "demo";
CompletableFuture<WriteResult> future = lindormTSDBClient.write(database, record);
```

- 每次提交多行记录（推荐）。这种方式提交方式，可减少提交时的SDK异步队列锁竞争，效率较高。

```
List<Record> records;
// 写入默认数据库
CompletableFuture<WriteResult> future = lindormTSDBClient.write(records);

// 写入指定的数据库
String database = "demo";
CompletableFuture<WriteResult> future = lindormTSDBClient.write(database, records);
```

- 处理CompletableFuture<WriteResult>完成时的结果。这里仅是一个示例，业务上可以结合实际情况，调用CompletableFuture的其他方法进行结果处理。

```
CompletableFuture<WriteResult> future = lindormTSDBClient.write(records);
// 处理异步写入结果
future.whenComplete((r, ex) -> {
    if (ex != null) { // 发送异常，一般都是写入失败
        System.out.println("Failed to write.");
        Throwable throwable = ExceptionUtils.getRootCause(ex);
        if (throwable instanceof LindormTSDBException) {
            LindormTSDBException e = (LindormTSDBException) throwable;
            System.out.println("Caught an LindormTSDBException, which means your request made it to Lindorm
TSDB, "
                               + "but was rejected with an error response for some reason.");
            // errorCode的取值和对应的错误含义，请参考：https://help.aliyun.com/document_detail/360486.html
            System.out.println("Error Code: " + e.getCode());
            System.out.println("SQL State:  " + e.getSqlstate());
            System.out.println("Error Message: " + e.getMessage());
        }  else {
            throwable.printStackTrace();
        }
    } else  { // 一般都是写入成功
        if (r.isSuccessful()) {
            System.out.println("Write successfully.");
        } else {
            System.out.println("Write failure.");
        }
    }
});
```

> **注意**
> 不要在CompletableFuture的whenComplete等方法中做复杂耗时的计算，否则可能会阻塞写入；若确实需要做复杂的耗时计算，请将计算逻辑提交到其他独立现场池中。

使用回调函数Callback处理异步写入结果的接口

- 写入回调接口Callback的定义如下所示。onCompletion方法上的result表示写入结果，records表示这个回调函数关联的写入记录，异常e表示写入失败时的异常。

```
public interface Callback {
    void onCompletion(WriteResult result, List<Record> records, Throwable e);
}
```

下面是使用Callback接口处理写入结果的一个示例。

```
Callback callback =  new Callback() {
    @Override
    public void onCompletion(WriteResult result, List<Record> list,
                             Throwable throwable) {
        if (throwable != null) { // 写入失败
            if (throwable instanceof LindormTSDBException) {
                LindormTSDBException ex = (LindormTSDBException) throwable;
                // errorCode的取值和对应的错误含义，请参考：https://help.aliyun.com/document_detail/360486.html
                System.out.println("errorCode: " + ex.getCode());
                System.out.println("sqlstate: " + ex.getSqlstate());
                System.out.println("message: " + ex.getMessage());
            } else {
                // 其他错误
                throwable.printStackTrace();
            }
        } else {
            if (result.isSuccessful()) {
                System.out.println("Write successfully.");
            } else {
                System.out.println("Write failure.");
            }
        }
    }
};
```

> **注意**
> 不要在Callback的onCompletion中做复杂耗时的计算，否则可能会阻塞写入；若确实需要做复杂的耗时计算，请将计算逻辑提交到其他独立现场池中。

- 每次提交一行记录。

```
// 写入默认数据库
lindormTSDBClient.write(record, callback);

// 写入指定的数据库
String database = "demo";
lindormTSDBClient.write(database, record, callback);
```

- 每次提交多行记录（推荐）。这种方式提交方式，可减少提交时的SDK异步队列锁竞争，效率较高。

```
List<Record> records;
// 写入默认数据库
lindormTSDBClient.write(records, callback);

// 写入指定的数据库
String database = "demo";
lindormTSDBClient.write(database, records, callback);
```

查询接口

Lindorm TSDB SDK的查询接口是通过SQL查询数据的，关于SQL语句说明可以参考SQL语法参考。
Lindorm TSDB SDK的查询接口如下所示，其中，入参有要查询的数据库database, 查询语句sql，查询结果每批返回的行数大小chunkSize。

```
ResultSet query(String database, String sql, int chunkSize);
```

Lindorm TSDB SDK 使用ResultSet接口表示SQL查询结果，其定义如下所示。

```
public interface ResultSet extends Closeable {

    QueryResult next();

    void close();
}
```

在处理查询结果时，可以循环调用ResultSet的next方法获取查询结果，当next方法返回的QueryResult对象为null时，表示已经读取完全部查询结果。 当查询结束时，调用ResultSet的close方法释放对应的IO资源。
QueryResult表示每批返回的查询结果，该对象有columns、metadata、rows等3个字段，其中，columns为查询结果的查询的列名集合，metadata为对应列的数据类型集合，rows为查询结果的按行返回的集合。其中，关于Lindorm时序引擎查询支持的数据类型集合，请参考数据类型。下面是处理查询结果ResultSet的示例。

```
String sql = "select * from sensor";
int chunkSize = 100;
ResultSet resultSet = lindormTSDBClient.query("demo", sql, chunkSize);

// 处理查询结果
try {
    QueryResult result = null;
    // 当resultSet的next()方法返回为null，表示已经读取完所有的查询结果
    while ((result = resultSet.next()) != null) {
        List<String> columns = result.getColumns();
        System.out.println("columns: " + columns);
        List<String> metadata = result.getMetadata();
        System.out.println("metadata: " + metadata);
        List<List<Object>> rows = result.getRows();
        for (int i = 0, size = rows.size(); i < size; i++) {
            List<Object> row = rows.get(i);
            System.out.println("row #" + i + " : " + row);
        }
    }
} finally {
    // 查询结束后，需确保调用ResultSet的close方法，以释放IO资源
    resultSet.close();
}
```

> **注意**
> 不管查询是否成功，在查询结束时，都需显式调用ResultSet的close方法，以释放IO资源，否则会造成网络连接泄露。

另外，Lindorm TSDB SDK也提供了几个查询接口重载实现，如下所示，业务上可以根据实际情况选择使用不同实现。

```
// SQL查询语句
String sql = "xxxx";

// 1.使用SQL语句，查询默认数据库， 默认每批返回1000行数据
ResultSet resultSet = lindormTSDBClient.query(sql);

String database = "demo";
// 2.使用SQL语句，查询指定数据库，默认每批返回1000行数据
ResultSet resultSet = lindormTSDBClient.query(database, sql);
```

8.7.5.4. 异常处理

本文介绍在应用代码中如何处理Lindorm TSDB SDK抛出的异常。
Lindorm TSDB SDK 包含两类异常，一类是客户端异常ClientException, 另一类是服务端异常LindormTSDBException, 它们均继承自RuntimeException。

ClientException

ClientException指客户端尝试向Lindorm时序引擎发送请求以及数据传输时遇到的异常。例如，当发送请求时网络连接不可用，则会抛出ClientException。

LindormTSDBException

LindormTSDBException指服务器端异常，它来自于对服务器错误信息的解析。LindormTSDBException包含Lindorm时序引擎返回的错误码和错误信息，便于定位问题，并做出适当的处理。LindormTSDBException通常包含以下错误信息：

参数	描述
code	Lindorm时序引擎返回的错误码。
sqlstate	Lindorm时序引擎返回的sqlstate值。
message	Lindorm时序引擎返回的详细错误信息。

> 说明
> 关于Lindorm时序引擎返回的错误对应的详细错误，可以参考Lindorm时序引擎的错误码列表：常见错误码参考。

异常处理示例

以下代码用于展示异常处理：

```
try {
    // Lindorm TSDB操作，比如查询数据等；
    lindormTSDBClient.query(....);
} catch (LindormTSDBException e) {
    System.out.println("Caught an LindormTSDBException, which means your request made it to Lindorm TSDB, "
            + "but was rejected with an error response for some reason.");
    // errorCode的取值和对应的错误含义，请参考：https://help.aliyun.com/document_detail/360486.html
    System.out.println("Error Code: " + e.getCode());
    System.out.println("SQL State:  " + e.getSqlstate());
    System.out.println("Error Message: " + e.getMessage());
} catch(ClientException ce) {
    System.out.println("Caught an ClientException, which means the client encountered "
            + "a serious internal problem while trying to communicate with Lindorm TSDB, "
            + "such as not being able to access the network.");
    System.out.println("Error Message: " + ce.getMessage());
}
```

8.7.5.5. 版本说明

本文主要进行Lindorm时序引擎的Java Native SDK版本说明。

> 说明
> Lindorm时序引擎的Java Native SDK的所有版本均可在Maven中央仓库获取使用，更多详情请参见maven repository。

V1.0.1

- 增加HTTPS支持。
- 将依赖的okhttp3库shade，避免依赖冲突。
- 增加带参数的同步写入方法writeSync，增加写入方式的灵活性。
- 增加强制flush方法，用于强制将异步队列中的数据发送出去。
- 增加健康检查接口isHealth，用于检查服务端是否健康。
- 增加在创建客户端时检查URL是否可用。
- 增加服务端版本号校验，最小要求服务端版本V3.4.8。

V1.0.0

首次发布，支持Basic鉴权、异步攒批写入、数据流式查询等基础功能。

8.7.6. SQL语法参考

8.7.6.1. 数据类型

本文介绍Lindorm时序引擎中支持的数据类型。
Lindorm时序引擎SQL支持常用的数据类型，本文主要介绍SQL支持的数据类型。

数据类型	描述
BIGINT	整数类型。
DOUBLE	浮点数类型。
VARCHAR	字符串类型。
BOOLEAN	布尔类型，true或false。
TIMESTAMP	时间戳类型。用户可以通过以下两种方式写入时间戳类型的数据： • 时间戳的字符串形式： 即按yyyy-MM-DD HH:mm:ss(.sss)的形式指定的时间戳字符串。如 `2022-01-01 12:00:00` • 时间戳的整型形式（64位Unix时间戳，精确到毫秒）： 即UTC时间1970年1月1日00:00:00.000以来的**毫秒数**。

> **注意**
>
> 1. 目前创建时序表时，只能指定一列为TIMESTAMP类型。
> 2. 时序引擎对于时间戳的编码和解析，统一按照北京时间（GMT+8）来进行。
> 3. 在一些较旧的系统中，其Unix时间戳仍然是按照32位进行处理。这类时间戳写入Lindorm时序引擎前建议乘以1000。如果不加转换地将这些时间戳直接写入时序引擎，将会在语义上的解释引起偏差。
>
>   比如时间戳**1641009600** 在这类较旧系统中会被解释为 `2022-01-01 12:00:00` ，但在Lindorm时序引擎中则会被解释为 `1970-01-20 07:50:09(GMT+8)` 。

8.7.6.2. 常用时序函数

本文介绍时序SQL支持的常用函数。

> **说明**
> 本文介绍的时序函数，需满足内核版本 >= 3.4.7

示例数据

表中示例数据如下，共两条时间线（关于时间线概念，请参见名词解释）：

- 时间线 1：device_id=F07A1260, region=north-cn，两个指标temperature、humidity
- 时间线 2：device_id=F07A1261, region=south-cn，两个指标temperature、humidity

```
 > describe sensor;
+-------------+-----------+------------+
| columnName  | typeName  | columnKind |
+-------------+-----------+------------+
| device_id   | VARCHAR   | TAG        |
| region      | VARCHAR   | TAG        |
| time        | TIMESTAMP | TIMESTAMP  |
| temperature | DOUBLE    | FIELD      |
| humidity    | DOUBLE    | FIELD      |
+-------------+-----------+------------+

> select * from sensor;
+-----------+----------+---------------------------+-------------+-----------+
| device_id |  region  |           time            | temperature | humidity  |
+-----------+----------+---------------------------+-------------+-----------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 12.100000   | 45.000000 |
| F07A1260  | north-cn | 2021-04-22T15:33:10+08:00 | 13.200000   | 47.000000 |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   | 46.000000 |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 12.100000   | 51.000000 |
| F07A1261  | south-cn | 2021-04-22T15:51:21+08:00 | 13.200000   | 52.000000 |
| F07A1261  | south-cn | 2021-04-22T15:51:27+08:00 | 10.600000   | 53.000000 |
| F07A1261  | south-cn | 2021-04-22T15:51:47+08:00 | 20.600000   | 55.000000 |
+-----------+----------+---------------------------+-------------+-----------+
```

聚合函数

SUM函数

```
SELECT SUM(field_name) FROM tb_name [WHERE clause] [SAMPLE BY interval];
```

- **函数说明**：计算表中field_name列的和
- **参数说明**：field_name，指定FIELD字段
- **返回类型**：返回DOUBLE或长整型BIGINT
- **应用字段**：FIELD字段，不能用于VARCHAR、BOOLEAN类型

示例

1、设备F07A1260和F07A1261 按照20s时间窗口，使用sum进行降采样聚合

```
> select device_id, region, time, sum(temperature) as temperature from sensor where time <= '2021-04-22T15:5
1:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 25.300000   |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 35.900000   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
4 rows in set (34 ms)
```

2、设备F07A1260和F07A1261，分别使用sum聚合计算各自温度和

```
> select device_id, region, time, sum(temperature) as temperature from sensor where  time <= '2021-04-22T15:
51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 1970-01-01T08:00:00+08:00 | 35.900000   |
| F07A1261  | south-cn | 1970-01-01T08:00:00+08:00 | 56.500000   |
+-----------+----------+---------------------------+-------------+
```

3、使用sum计算所有设备温度之和

```
> select sum(temperature) as temperature from sensor where time <= '2021-04-22T15:51:47+08:00' and time >= '
2021-04-22T15:33:00+08:00'
+-------------+
| temperature |
+-------------+
| 92.400000   |
+-------------+
```

AVG函数

```
SELECT AVG(field_name) FROM tb_name [WHERE clause] [SAMPLE BY 0 | SAMPLE BY interval];
```

- **函数说明**：计算表中field_name列的均值
- **参数说明**：field_name，指定FIELD字段
- **返回类型**：返回DOUBLE类型
- **应用字段**：FIELD字段，不能用于VARCHAR、BOOLEAN类型

示例

1、设备F07A1260和F07A1261 按照20s时间窗口，使用avg进行均值降采样聚合

```
> select device_id, region, time, avg(temperature) as temperature from sensor where time <= '2021-04-22T15:5
1:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 12.650000   |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 11.966667   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261，分别使用avg聚合计算各自均值

```
> select device_id, region, time, avg(temperature) as temperature from sensor where time <= '2021-04-22T15:5
1:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 1970-01-01T08:00:00+08:00 | 11.966667   |
| F07A1261  | south-cn | 1970-01-01T08:00:00+08:00 | 14.125000   |
+-----------+----------+---------------------------+-------------+
```

3、使用avg计算所有设备温度的均值

```
> select avg(temperature) as temperature from sensor where time <= '2021-04-22T15:51:47+08:00' and time >= '
2021-04-22T15:33:00+08:00';
+-------------+
| temperature |
+-------------+
| 13.200000   |
+-------------+
```

COUNT函数

```
SELECT COUNT(field_name) FROM tb_name [WHERE clause] [SAMPLE BY interval];
```

- **函数说明**：统计表中field_name列的值个数
- **参数说明**：field_name，指定FIELD字段

- **返回类型**：返回BIGINT类型
- **应用字段**：FIELD字段

示例

1、设备F07A1260和F07A1261 按照20s时间窗口，使用count统计各个窗口里的值个数

```
> select device_id, region, time, count(temperature) as temperature from sensor where time <= '2021-04-22T15
:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 2           |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 3           |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 1           |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 1           |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261，分别使用count计算各自温度值个数

```
> select device_id, region, time, count(temperature) as temperature from sensor where time <= '2021-04-22T15
:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 1970-01-01T08:00:00+08:00 | 3           |
| F07A1261  | south-cn | 1970-01-01T08:00:00+08:00 | 4           |
+-----------+----------+---------------------------+-------------+
```

3、使用count统计所有设备温度值个数

```
> select count(temperature) as temperature from sensor where time <= '2021-04-22T15:51:47+08:00' and time >=
'2021-04-22T15:33:00+08:00';
+-------------+
| temperature |
+-------------+
| 7           |
+-------------+
```

选择函数

MIN函数

```
SELECT MIN(field_name) FROM tb_name [WHERE clause] [SAMPLE BY interval];
```

- **函数说明**：统计表中field_name列的最小值
- **参数说明**：field_name，指定FIELD字段
- **返回类型**：返回DOUBLE或长整型BIGINT
- **应用字段**：FIELD字段，不能用于VARCHAR、BOOLEAN类型

示例

1、设备F07A1260和F07A1261 按照20s时间窗口，使用min查询各个窗口内的最小值

```
> select device_id, region, time, min(temperature) as temperature from sensor where time <= '2021-04-22T15:5
1:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 12.100000   |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 10.600000   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261，分别使用min查询各自温度的最小值

```
> select device_id, region, time, min(temperature) as temperature from sensor where time <= '2021-04-22T15:5
1:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:27+08:00 | 10.600000   |
+-----------+----------+---------------------------+-------------+
```

3、使用min查询所有设备温度的最小值

```
> select min(temperature) as temperature from sensor where time <= '2021-04-22T15:51:47+08:00' and time >= '
2021-04-22T15:33:00+08:00';
+-------------+
| temperature |
+-------------+
| 10.600000   |
+-------------+
```

MAX函数

```
SELECT MAX(field_name) FROM tb_name [WHERE clause] [SAMPLE BY interval];
```

- **函数说明**：统计表中field_name列的最大值
- **参数说明**：field_name，指定FIELD字段
- **返回类型**：返回DOUBLE或长整型BIGINT
- **应用字段**：FIELD字段，不能用于VARCHAR、BOOLEAN类型

示例

1、设备F07A1260和F07A1261 按照20s时间窗口，使用max查询各个窗口内的温度最大值

```
> select device_id, region, time, max(temperature) as temperature from sensor where time <= '2021-04-22T15:5
1:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 13.200000   |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 13.200000   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261，使用max分别查询各自温度的最大值

```
> select device_id, region, time, max(temperature) as temperature from sensor where time <= '2021-04-22T15:5
1:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:10+08:00 | 13.200000   |
| F07A1261  | south-cn | 2021-04-22T15:51:47+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

3、使用max查询所有设备温度的最大值

```
> select max(temperature) as temperature from sensor where time <= '2021-04-22T15:51:47+08:00' and time >= '
2021-04-22T15:33:00+08:00';
+-------------+
| temperature |
+-------------+
| 20.600000   |
+-------------+
```

FIRST函数

```
SELECT FIRST(field_name) FROM tb_name [WHERE clause] SAMPLE BY interval;
```

- **函数说明**：获取表中field_name列的第一个值
- **参数说明**：field_name，指定FIELD字段
- **返回类型**：返回应用的字段的类型
- **应用字段**：FIELD字段

示例

1、设备F07A1260和F07A1261 按照20s时间窗口，使用first获取各个窗口内的第一个温度值

```
> select device_id, region, time, first(temperature) as temperature from sensor where time <= '2021-04-22T15
:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 12.100000   |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 12.100000   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261，使用first分别获取各自温度的第一个值

```
> select device_id, region, time, first(temperature) as temperature from sensor where time <= '2021-04-22T15
:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 12.100000   |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 12.100000   |
+-----------+----------+---------------------------+-------------+
```

LAST函数

```
SELECT LAST(field_name) FROM tb_name [WHERE clause] SAMPLE BY interval;
```

- **函数说明**：获取表中field_name列的最后一个值
- **参数说明**：field_name，指定FIELD字段
- **返回类型**：返回应用的字段的类型
- **应用字段**：FIELD字段

示例

1、设备F07A1260和F07A1261 按照20s时间窗口，使用last获取各个窗口的最后一个温度值

```
> select device_id, region, time, last(temperature) as temperature from sensor where time <= '2021-04-22T15:
51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 13.200000   |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 10.600000   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261，使用last分别获取各自温度的最后一个值

```
> select device_id, region, time, last(temperature) as temperature from sensor where time <= '2021-04-22T15:
51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:47+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

PERCENTILE函数

```
SELECT PERCENTILE(field_name,P) FROM tb_name [WHERE clause] SAMPLE BY interval;
```

- **函数说明**：计算与字段field_name关联的第P个百分位字段值
- **参数说明**：field_name，指定FIELD字段；P介于0和100（含）之间的整数或浮点数；默认值 50
- **返回类型**：返回DOUBLE类型
- **应用字段**：FIELD字段，不能用于VARCHAR、BOOLEAN类型

示例

1、设备F07A1260和F07A1261 按照20s时间窗口，使用percentile获取各个窗口内的90分位值

```
> select device_id, region, time, percentile(temperature, 90) as temperature from sensor where time <= '2021
-04-22T15:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 13.200000   |
| F07A1261  | south-cn | 2021-04-22T15:51:20+08:00 | 13.200000   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261，分别使用percentile获取各自温度的90分位值

```
> select device_id, region, time, percentile(temperature, 90) as temperature from sensor where time <= '2021
-04-22T15:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 1970-01-01T08:00:00+08:00 | 13.200000   |
| F07A1261  | south-cn | 1970-01-01T08:00:00+08:00 | 20.600000   |
+-----------+----------+---------------------------+-------------+
```

转换函数

RATE函数

```
SELECT RATE(field_name, 'interval units') FROM tb_name [WHERE clause] SAMPLE BY 0;
```

- **函数说明**：计算表中field_name列值的斜率，具体公式说明，每个数据点的时间戳与值分别是：

应用rate()函数之后，返回 N-1 个数据点，每个数据点的时间戳与值分别是：

> **说明**
> 公式中interval对应指定的变化率单位，计算时统一换算为秒处理

- **参数说明**：
  - field_name，指定FIELD字段；
  - interval units，指定变化率单位；默认1s（ `s` 代表秒， `m` 代表分， `h` 代表小时， `d` 代表天）
- **返回类型**：返回DOUBLE类型
- **应用字段**：FIELD字段，不能用于VARCHAR、BOOLEAN类型

示例

1、设备F07A1260和F07A1261，计算每个设备温度的秒级变化率

```
> select device_id, region, time, rate(temperature) as temperature from sensor where time <= '2021-04-22T15:
51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:10+08:00 | 0.110000    |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | -0.260000   |
| F07A1261  | south-cn | 2021-04-22T15:51:21+08:00 | 1.100000    |
| F07A1261  | south-cn | 2021-04-22T15:51:27+08:00 | -0.433333   |
| F07A1261  | south-cn | 2021-04-22T15:51:47+08:00 | 0.500000    |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261，计算每个设备温度分钟变化率

```
> select device_id, region, time, rate(temperature, '1m') as temperature from sensor where time <= '2021-04-
22T15:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:10+08:00 | 6.600000    |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | -15.600000  |
| F07A1261  | south-cn | 2021-04-22T15:51:21+08:00 | 66.000000   |
| F07A1261  | south-cn | 2021-04-22T15:51:27+08:00 | -26.000000  |
| F07A1261  | south-cn | 2021-04-22T15:51:47+08:00 | 30.000000   |
+-----------+----------+---------------------------+-------------+
```

3、设备F07A1260和F07A1261，先分别按照20s 对温度avg降采样，然后对降采样后的值计算变化率

```
> select device_id, region, time, rate(avg(temperature), '1m') as temperature from sensor where time <= '202
1-04-22T15:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | -6.150000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 25.900000   |
+-----------+----------+---------------------------+-------------+
```

> **说明**
> ：RATE函数使用限制，目前SQL语句只支持RATE算子单独使用

例如，如下混用方式不支持

```
> select region, time, rate(temperature), rate(humidity) from sensor where time <= '2021-04-22T15:51:47+08:0
0' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
ERROR 9001 (0A000): UNSUPPORTED: latest/rate/delta must be the only aggregator in a single SQL statement

> select region, time, rate(temperature), latest(humidity) from sensor where time <= '2021-04-22T15:51:47+08
:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
ERROR 9001 (0A000): UNSUPPORTED: latest/rate/delta must be the only aggregator in a single SQL statement
```

DELTA函数

```
SELECT DELTA(field_name) FROM tb_name [WHERE clause] SAMPLE BY 0;
```

- **函数说明**：计算表中field_name列的值与前一行对应值的差。

计算公式说明，每个数据点的时间戳与值分别是：
--
应用delta()函数之后，返回 N-1 个数据点，每个数据点的时间戳与值分别是：

- **参数说明**：field_name，指定FIELD字段
- **返回类型**：返回DOUBLE类型
- **应用字段**：FIELD字段，不能用于VARCHAR、BOOLEAN类型

示例

1、设备F07A1260和F07A1261 ，计算每个设备温度与前一行差值

```
> select device_id, region, time, delta(temperature) as temperature from sensor where time <= '2021-04-22T15
:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:10+08:00 | 1.100000    |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | -2.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:21+08:00 | 1.100000    |
| F07A1261  | south-cn | 2021-04-22T15:51:27+08:00 | -2.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:47+08:00 | 10.000000   |
+-----------+----------+---------------------------+-------------+
```

2、设备F07A1260和F07A1261 ，先分别按照20s 对温度avg降采样，然后对降采样后的值计算差值

```
> select device_id, region, time, delta(avg(temperature)) as temperature from sensor where time <= '2021-04-
22T15:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 20s;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | -2.050000   |
| F07A1261  | south-cn | 2021-04-22T15:51:40+08:00 | 8.633333    |
+-----------+----------+---------------------------+-------------+
```

> 说明
> ：DELTA函数限制，目前SQL语句只支持DELTA算子单独使用

例如，如下混用方式不支持

```
> select region, time, delta(temperature), rate(humidity) from sensor where time <= '2021-04-22T15:51:47+08:
00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
ERROR 9001 (0A000): UNSUPPORTED: latest/rate/delta must be the only aggregator in a single SQL statement
```

LATEST最新值

```
SELECT LATEST(field_name, N) FROM tb_name [WHERE clause] SAMPLE BY 0;
```

- **函数说明**：统计表中field_name列的最新的N个值
- **参数说明**：
  - field_name：指定field字段
  - N：整型，用于指定查询最新的N条数据；默认1，取最新一条数据
- **返回类型**：返回应用字段对应的类型
- **应用字段**：FIELD字段

示例

1、查询设备F07A1260和F07A1261各自的最新温度测点值

```
> select device_id, region, time, latest(temperature) as temperature from sensor where time <= '2021-04-22T1
5:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1261  | south-cn | 2021-04-22T15:51:47+08:00 | 20.600000   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
+-----------+----------+---------------------------+-------------+
```

2、查询设备F07A1260和F07A1261各自最新的两条温度测点值

```
> select device_id, region, time, latest(temperature, 2) as temperature from sensor where time <= '2021-04-2
2T15:51:47+08:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1261  | south-cn | 2021-04-22T15:51:27+08:00 | 10.600000   |
| F07A1261  | south-cn | 2021-04-22T15:51:47+08:00 | 20.600000   |
| F07A1260  | north-cn | 2021-04-22T15:33:10+08:00 | 13.200000   |
| F07A1260  | north-cn | 2021-04-22T15:33:20+08:00 | 10.600000   |
+-----------+----------+---------------------------+-------------+
```

> 说明
> ：LATEST函数限制，目前只支持单独使用LATEST，不支持与其他函数混合使用

例如，如下混用方式不支持

```
> select region, time, latest(temperature), rate(humidity) from sensor where time <= '2021-04-22T15:51:47+08
:00' and time >= '2021-04-22T15:33:00+08:00' sample by 0;
ERROR 9001 (0A000): UNSUPPORTED: latest/rate/delta must be the only aggregator in a single SQL statement
```

8.7.6.3. DDL

8.7.6.3.1. CREATE DATABASE

本文介绍创建数据库语法。

语法

```
create_database_statement ::= CREATE DATABASE database_name [ WITH attribute_statement ]
attribute_statement       ::= '(' attribute_definition (',' attribute_definition)* ')'
attribute_definition      ::= attr_name '=' attr_val
```

参数说明

参数	说明
attribute_statement	指定Database的附加属性。

参数	说明
attr_name	属性名。必须是SQL常量或标识符。
attr_val	属性值。必须是SQL常量。

当前支持的数据库属性如下：

属性名	取值类型	默认值	描述	示例
ttl	BIGINT	无，表示数据永久保存。	数据保存有效期，超过有效期的数据会被自动清理。单位为天。设置为0表示数据永久保存。	60
cold_boundary	BIGINT	0，表示不启用冷存储。	冷热存储分界线，单位为天。	30

> 说明
> - 如果同时设置了ttl和cold_boundary，ttl的设置时长必须大于cold_boundary的时间。
> - 如果希望设置cold_boundary，则需要先在Lindorm管理控制台上开通冷存储。

示例

以下例子展示了创建一个名为DB1的数据库。其中数据存满30天后转入冷存储，存满60天后会被清除。

```
CREATE DATABASE DB1 WITH (cold_boundary=30, ttl=60);
```

8.7.6.3.2. CREATE TABLE

创建时序数据表的语法说明。

语法

```
create_table_statement ::= CREATE TABLE table_name '('
                             tag_definition
                             ( ',' tag_definition )*
                             ( ',' time_definition)?
                             ( ',' field_definition )*
                             [',' primary_key_definition]
                             ')'
tag_definition         ::= tag_name [ VARCHAR ] TAG
time_definition        ::= time_column_name TIMESTAMP
field_definition       ::= field_name TYPE
primary_key_definition ::= PRIMARY KEY '(' tag_name [',' tagname]* ')'
```

说明

参数	说明
`tag_name`	定义标签的列名，数据类型固定为VARCHAR，且必须使用关键字TAG来显式标识，否则会被引擎理解为Field字段。此处数据类型的声明可以省略。
`time_column_name`	定义时间戳的列名，数据类型必须为TIMESTAMP，精度支持到毫秒（ms）
`field_name`	定义数据字段的列名，目前支持的数据类型为DOUBLE、VARCHAR或BOOLEAN。

- 写入数据前，需要提前创建表，并指定列类型。

- PRIMARY KEY 与传统关系型数据库的主键并不相同，主要是用于自定义时序数据表的数据分片规则，其中指定的列名必须是 `tag_definition` 中的标签列集合的子集。PRIMARY KEY的语法是自引擎版本3.4.9 开始支持（时序引擎版本说明）。
- 通常以数据源唯一标识作为 PRIMARY KEY，例如物联网场景设备ID，车联网场景车辆唯一识，监控场景的应用ID 或者 ip:port 等。详细可以参见 PRIMARY KEY设计的最佳实践。

> 注意
>
> 1. 目前创建时序表时，只能指定一列为TIMESTAMP类型。
> 2. 时序引擎对于时间戳的编码和解析，统一按照北京时间（GMT+8）来进行。
> 3. 在一些较旧的系统中，其Unix时间戳仍然是按照32位进行处理。这类时间戳写入Lindorm时序引擎前建议乘以1000。如果不加转换地将这些时间戳直接写入时序引擎，将会在语义上的解释引起偏差。
>
>   比如时间戳**1641009600** 在这类较旧系统中会被解释为 `2022-01-01 12:00:00` ，但在Lindorm时序引擎中则会被解释为 `1970-01-20 07:50:09(GMT+8)` 。

示例

创建名为sensor的表，拥有device_id和region两个Tag，以及temperature和humidity两个Field。同时由于大部分查询场景会查询单个设备的瞬时数据，因此显式指定 device_id 作为数据分片的唯一标签列。

满足上述需求的CREATE TABLE语句如下：

```
CREATE TABLE sensor (
    device_id VARCHAR TAG,
    region VARCHAR TAG,
    time    TIMESTAMP,
    temperature DOUBLE,
    humidity DOUBLE,
    PRIMARY KEY(device_id));
```

8.7.6.3.3. CREATE USER

创建用户的语法介绍。

语法

```
create_user_statement ::= CREATE USER [IF NOT EXISTS] user_name IDENTIFIED BY password_definition
```

参数说明

参数	说明
user_name	用户名。可以是SQL常量或标识符。
password_definition	用户密码。必须是VARCHAR类型的SQL常量。

> 注意
>
> 若尝试创建的是一个已存在的用户名，在未指定IF NOT EXISTS时，则会提示用户已存在的错误；若此时指定了IF NOT EXISTS，则并不会报错而是中止创建过程。

示例

以下示例创建了一个名为 `tsdbuser` ，密码为 `asdf1234` 的用户。

```
CREATE USER IF NOT EXISTS `tsdbuser` IDENTIFIED BY 'asdf1234';
```

8.7.6.3.4. CREATE CONTINUOUS QUERY

本文介绍创建持续查询的SQL语法。

语法

```
create_cq_statement ::= CREATE CONTINUOUS QUERY [database_name.] cq_name WITH (  cq_attribute_statement ) AS
insert_select_statement

cq_attribute_statement ::= INTERVAL 'interval_string'
```

参数说明

参数	说明
database_name	持续查询对象所属数据库的名称。不指定的情况下默认是当前数据库。 必须是SQL常量或标识符。
cq_name	持续查询的名称。
interval_string	用于指定持续查询的计算时间窗口，时间窗口的最小单位为秒。格式如下： %d%h%m%s。比如1h30s表示1小时30秒, 1d 表示一天。 其中： • d：天 • h：小时 • m：分 • s：秒

> 注意
>
> 1. insert_select_statement的语法可参见 数据写入。
> 2. 指定insert_select_statement时可以不用指定查询的时间范围，引擎会自动在当前时间窗口内进行计算。

示例

以下示例创建了一个持续查询以"天"为粒度，每天计算原始时序表sensor中的平均温度和平均湿度，并将数据写入一个新库 `db_sensor_year` 中的表 `sensor` 。

```
CREATE CONTINUOUS QUERY `db_sensor`.`my_cq` WITH (INTERVAL='1d') AS
INSERT INTO `db_sensor_year`.`sensor`
SELECT MEAN(`temperature`) AS `temperature`,
       MEAN(`humidity`) AS `humidity`, `device_id`, `region`
FROM  `db_sensor`.`sensor`;
```

8.7.6.3.5. ALTER DATABASE

修改数据库属性的语法介绍。

语法

```
alter_database_statement ::= ALTER DATABASE database_name [ WITH attribute_statement ]
attribute_statement            ::= '(' attribute_definition (',' attribute_definition)* ')'
attribute_definition             ::= attr_name '=' attr_val
```

参数说明

参数	说明
attribute_statement	指定Database的附加属性。

参数	说明
attr_name	属性名。必须是SQL常量或标识符。
attr_val	属性值。必须是SQL常量。

> 说明
> 当前支持的数据库属性，请参见 CREATE DATABASE 。

> 注意
> ALTER DATABASE只能修改数据库属性，无法修改Database名。

示例

以下例子展示了将一个名为 DB1 的数据库的TTL修改为40天。

```
ALTER DATABASE DB1 WITH (ttl=40);
```

8.7.6.3.6. ALTER TABLE

修改时序数据表的语法说明。

语法

```
alter_table_statement ::=    ALTER TABLE table_name
alter_option
 ( ','
alter_option
 )*
alter_option          ::=    ADD | ALTER [COLUMN]
column_definition

column_definition     ::=    column_name TYPE [TAG]
```

参数说明

参数	说明
table_name	表名
column_name	列名

> 注意
> 1. 当前ALTER TABLE语句只支持以下两种操作：
>   i. 增加列，且新增的列只能是TAG列或FIELD列。
>   ii. 修改FIELD列的类型（TAG列或时间戳列的类型无法修改）。
> 2. 无法将新增的TAG列声明为PRIMARY KEY。
> 3. 将一个FIELD列的类型修改后，后续查询该列的数据时，对于先前已写入的数据将会按最新生效的数据类型来处理，因此有可能会出现先前写入的数据无法被查到的情况。

示例

以下示例为已有的表 `aqm` 新增了一个名为 `co` 的FIELD列：

```
ALTER TABLE aqm ADD co DOUBLE;
```

以下示例为已有的表 `aqm` 新增了一个名为 `region` 的TAG列：

```
ALTER TABLE aqm ADD region VARCHAR TAG;
```

以下示例将表 `aqm` 新增的 `co` 列的类型修改为BIGINT：

```
ALTER TABLE aqm ALTER COLUMN co BIGINT;
```

8.7.6.3.7. ALTER SYSTEM

本文介绍修改系统动态配置参数的语法。

语法

```
alter_system_statement  ::= ALTER SYSTEM SET configuration_name = configuration_value;
```

参数说明

参数	说明
configuration_name	动态配置参数名。必须是一个SQL标识符。
configuration_value	动态配置参数值。必须是一个SQL常量。 根据变更动态配置参数名不同，这里的SQL常量可以是BIGINT类型型常量、DOUBLE类型常量、BOOLEAN型常量或者是VARCHAR类型常量。

支持使用ALTER SYSTEM变更的动态参数

配置参数	参数值类型	说明
USER_AUTH	boolean	设置时序引擎用户认证与鉴权功能的开关。 • TRUE 启用用户认证与鉴权。任何与时序引擎的连接都需要带上用户名和密码，时序引擎会对用户进行认证，并该连接发起的SQL中的数据对象进行鉴权。 • FALSE 关闭用户认证与鉴权。与时序引擎的连接无需带上用户名和密码。 实例的默认值：FALSE

示例

- 开启实例的用户认证与鉴权功能

  ```
  ALTER SYSTEM SET USER_AUTH=TRUE;
  ```

  - 时序引擎一旦开启用户认证和权限校验，后续的所有访问都必须带上用户信息，否则将可能出现鉴权失败（包括当前执行 `ALTER SYSTEM` 语句启用用户认证和权限校验的连接）。因此，该功能开启后，已有的部分业务访问可能会失败。所以在开启该功能前，建议仔细规划并充分评估影响。
  - 用户认证与权限校验功能开启后也可通过ALTER SYSTEM语句关闭。执行语句如下：

    ```
    ALTER SYSTEM SET USER_AUTH=FALSE;
    ```

  - 查看当前实例是否已经开启用户认证与权限校验功能，可通过SHOW PARAMETER语句查看。执行语句如下：

    ```
    SHOW PARAMETER USER_AUTH;
    ```

- 关闭实例的用户认证与鉴权功能

```
ALTER SYSTEM SET USER_AUTH=FALSE;
```

8.7.6.3.8. ALTER USER

更改用户属性的语法介绍。

语法

```
alter_user_statement ::= ALTER USER [IF EXISTS] user_name SET PASSWORD '=' password_definition [ WHERE OLDPA
SSWORD '=' password_definition];
```

参数说明

参数	说明
user_name	用户名。可以是SQL常量或标识符。
password_definition	用户密码。必须是VARCHAR类型的SQL常量。

注意

1. 若尝试修改的是一个不存在的用户名，在未指定IF EXISTS时，则会报错提示用户不存在；若此时指定了IF EXISTS，则并不会报错而是中止变更过程。
2. 如果当前修改的用户名就是当前连接登录的用户，则需要指定 `WHERE OLDPASSWORD = password_definition` 子句进行当前的密码校验。
3. ALTER USER语句当前只支持修改密码，不支持修改用户名。若希望更改用户名，则需要先使用DROP USER语句删除用户，然后使用CREATE USER语句重新创建用户。

示例

以下示例展示了以一个拥有全局ADMIN权限的用户连接后，尝试将用户 `tsdbuser` 的密码修改为 `newpassword` 。

```
ALTER USER IF EXISTS `tsdbuser` SET PASSWORD = 'newpassword';
```

以下示例则展示了以用户 `tsdbuser` 连接后试图变更自己密码为 `newpassword` 。

```
ALTER USER `tsdbuser` SET PASSWORD = 'newpassword' WHERE OLDPASSWORD = 'asdf1234';
```

8.7.6.3.9. DROP DATABASE

删除数据库的语法介绍。

语法

```
drop_database_statement ::= DROP DATABASE database_name;
```

注意

默认数据库 `default` 无法被删除。

示例

以下例子展示了如何删除一个数据库DB1：

```
DROP DATABASE DB1;
```

8.7.6.3.10. DROP USER

删除用户的语法介绍。

语法

```
drop_user_statement  ::=  DROP USER [IF EXISTS] user_name;
```

参数说明

参数	说明
user_name	用户名。可以是SQL常量或标识符。

> 注意
> 若尝试删除的是一个不存在的用户名，在未指定IF EXISTS时，则会报错提示用户不存在；若此时指定了IF EXISTS，则并不会报错而是中止删除过程。

示例

以下示例展示了删除一个名为 `tsdbuser` 的用户。

```
DROP USER IF EXISTS `tsdbuser`;
```

8.7.6.3.11. DROP CONTINUOUS QUERY

本文介绍删除持续查询的SQL语法。

语法

```
drop_cq_statement ::= DROP CONTINUOUS QUERY [database_name.]cq_name
```

参数说明

参数	说明
database_name	持续查询对象所属数据库的名称。不指定的情况下默认是当前数据库。
cq_name	持续查询的名称。

示例

以下示例展示了将 CREATE CONTINUOUS QUERY示例 中创建的持续查询进行删除的方法。

```
DROP CONTINUOUS QUERY `db_sensor`.`my_cq`;
```

8.7.6.3.12. DESCRIBE DATABASE

查询指定数据库属性的语法。

语法

```
describe_database_statement ::= DESCRIBE DATABASE database_name;
```

DESCRIBE DATABASE语句的结果集以表的形式返回，其结果集的shema如下：

列名	类型
attribute	VARCHAR
value	VARCHAR

示例

以下例子展示查询名为 DB1 的数据库的属性：

```
DESCRIBE DATABASE DB1;
+-----------+-------+
| attribute | value |
+-----------+-------+
| ttl       |    60 |
+-----------+-------+
```

8.7.6.3.13. DESCRIBE TABLE

查询指定时序数据表的Schema信息。

语法

```
describe_table_statement ::= DESCRIBE TABLE table_name
```

DESCRIBE TABLE语句的结果集以表的形式返回，其结果集的shema如下：

列名	类型	结果说明
columnName	VARCHAR	该列的列名。
typeName	VARCHAR	该列的数据类型。
columnKind	VARCHAR	该列的种类。 根据建表语句的制定，可能的种类有3种： • **TAG**：表明该列是一个标签列。 • **TIMESTAMP**：表明该列是一个时间戳列。一张时序表中只会有一个种类为TIMESTAMP的列。 • **FIELD**：除了标签列和时间戳列以外的列都是数据列。用于记录上报的真实事件数据。

示例

```
> DESCRIBE TABLE sensor;
+-------------+-----------+-------------+
| columnName  | typeName  | columnKind |
+-------------+-----------+-------------+
| device_id   | VARCHAR   | TAG         |
| region      | VARCHAR   | TAG         |
| time        | TIMESTAMP | TIMESTAMP   |
| temperature | DOUBLE    | FIELD       |
| humidity    | BIGINT    | FIELD       |
+-------------+-----------+-------------+
5 rows selected (0.075 seconds)
```

8.7.6.3.14. SHOW

用户可以通过SHOW语句查看Lindorm时序引擎中指定类型的元数据。

语法

```
show_statement  ::= SHOW target_statement
target_statement ::= { DATABASES | TABLES | USERS | PRIVILEGES }
```

> 说明
>
> 1. SHOW语句支持的对象如下所示：
>   - DATABASES：查询当前实例中的全部数据库对象一览。
>   - TABLES：查询当前实例中的全部时序数据表一览。
>   - USERS：查询当前实例中的全部用户。
>   - PRIVILEGES：查询当前实例中的用户权限。
> 2. 随着SHOW语句的对象不同，执行SHOW语句所需的权限也略有区别。详细可参见 常用SQL所需权限 。

返回结果集说明

所有的SHOW语句返回的结果，都以表的形式返回结果集。以下将对各语句的结果集进行说明。

SHOW DATABASES

列名	类型	结果说明
database	VARCHAR	实例中所有数据库名。

SHOW TABLES

列名	类型	结果说明
Tables_In_${Database名}	VARCHAR	当前数据库中的所有表名。 若想查看其他数据库中的表，可用USE语句切换当前数据库。

SHOW USERS

列名	类型	结果说明
name	VARCHAR	当前实例中的所有用户名。

SHOW PRIVILEGES

列名	类型	结果说明
user	VARCHAR	拥有任意访问权限的用户名。
global privileges	VARCHAR	该用户拥有的GLOBAL权限集合。
database privileges	VARCHAR	该用户拥有的DATABASE权限的集合。
table privileges	VARCHAR	该权限拥有的TABLE级权限的集合。

> **说明**
>
> 1. 为了便于应用处理，权限列中的数据都是以JSON的形式展示。JSON的结构如下：
>   - key：访问对象的名称（如一个数据库名或一个表名）。全局权限的key固定为 `GLOBAL` 。
>   - value：权限的列表。详情可见下文的示例。
>
>   即使没有对应级别的权限，值也会是一个空JSON对象的字符串。
> 2. 当前时序引擎暂不支持TABLE级的权限。

示例

展示当前实例中的数据库一览：

```
> SHOW DATABASES;
+-----------+
| database |
+-----------+
| default  |
| metadata |
+-----------+
2 rows in set (13 ms)
```

展示当前实例中的时序数据表一览：

```
> SHOW TABLES;
+-------------+------------+
| tableSchem  | tableName  |
+-------------+------------+
| default     | sensor     |
| metadata    | COLUMNS    |
| metadata    | TABLES     |
+-------------+------------+
3 rows selected (0.064 seconds)
```

展示当前实例中的用户一览：

```
> SHOW USERS;
+------------+
|   name     |
+------------+
| testuser   |
| dbwriter   |
| root       |
+------------+
3 rows in set (11 ms)
```

展示当前实例中的权限一览：

```
> SHOW PRIVILEGES;
+------------+-------------------------------------+-------------------------------------+-----------------
-+
|    user    |          global privileges          |         database privileges         | table privileges
|
+------------+-------------------------------------+-------------------------------------+-----------------
-+
| testuser   | {"GLOBAL":["READ"]}                 | {}                                  | {}
|
| dbaccessor | {"GLOBAL":["ADMIN","READ","WRITE"]} | {"DB2":["READ"],"default":["WRITE"]} | {}
|
+------------+-------------------------------------+-------------------------------------+-----------------
-+
```

8.7.6.3.15. SHOW PARAMETER

本文介绍查询系统动态配置参数的语法。

语法

```
show_parameter_statement ::= SHOW PARAMETER configuration_name
```

参数说明

参数	说明
configuration_name	动态配置参数名。必须是一个SQL标识符。

> 说明
>
> 1. 该语句通常与 ALTER SYSTEM 语句成对使用。
> 2. 当前支持展示的动态配置项请参见 支持使用ALTER SYSTEM变更的动态参数。

示例

展示用户认证与鉴权开关的当前值

```
> SHOW PARAMETER USER_AUTH;
+------------+
| USER_AUTH |
+------------+
| FALSE     |
+------------+
1 rows in set (48 ms)
```

8.7.6.3.16. SHOW CONTINUOUS QUERIES

本文介绍显示当前数据库中已有持续查询对象的SQL语法。

语法

```
show_cq_statement
    ::=  SHOW CONTINUOUS QUERIES
```

> 说明
>
> 显示的是当前数据库中的所有已存在的持续查询的对象。想要查看其他数据库中的持续查询对象，可以通过 USE 语句先进行数据库的切换。

返回结果集说明

列名	类型	结果说明
name	VARCHAR	持续查询的名称。
interval	VARCHAR	该持续查询的计算窗口间隔。即 CREATE CONTINUOUS QUERY语句中指定的interval_string。
query	VARCHAR	持续查询中具体的查询计算的语句。

示例

以下示例展示了一个显示当前数据库下的持续查询的语句示例。

```
SHOW CONTINUOUS QUERIES;
+-------+----------+--------------------------------+
| name  | interval |             query              |
+-------+----------+--------------------------------+
| my_cq | 1d       | INSERT INTO                    |
|       |          | `db_sensor_year`.`sensor`      |
|       |          | (SELECT MEAN(`temperature`)  |
|       |          | AS `temperature`,              |
|       |          | MEAN(`humidity`) AS          |
|       |          | `humidity`, `device_id`,       |
|       |          | `region` FROM                  |
|       |          | `db_sensor_month`.`sensor`)    |
+-------+----------+--------------------------------+
1 rows in set (66 ms)
```

8.7.6.3.17. USE

本文介绍切换至目标数据库的SQL语法。

语法

```
use_statement
   ::= USE database_name;
```

参数说明

参数	说明
database_name	切换的目标数据库名，需要满足以下两点： • 目标数据库已存在。 • 目标数据库名必须是一个SQL常量或标识符。

> **说明**
>
> - 通过USE语句显式目标数据库后，在SQL中指定数据库对象（例如时序表）时，如果不显式指定所属的数据库名，则默认会在当前上下文中的数据库内寻找名称符合的数据库对象。
> - 通过Lindorm-cli或者JDBC访问时序引擎建立连接时，通常会在连接串中指定一个数据库名用于默认访问的数据库。如果在此情况下执行USE语句，也同样会在当前连接的上下文中切换数据库。

8.7.6.4. DML

8.7.6.4.1. 数据写入

本文介绍时序引擎SQL的数据写入语法。

语法

```
insert_statement ::= INSERT INTO table_name'('
                              tag_name
                              (',' tag_name)*
                              ',' time ','
                              field_name
                              (',' field_name)*
                           ')' VALUES
                           ('('
                              tag_value
                              (',' tag_value)*
                              ',' time_value ','
                              field_value
                              (',' field_value)*
                           ')')+
insert_select_statement ::= INSERT INTO table_name '('
                              tag_name
                              (',' tag_name)*
                              ',' time ','
                              field_name
                              (',' field_name)*
                           ')' select_statement
```

参数说明

参数	说明
`tag_name`	写入数据的Tag名称，与CREATE TABLE所指定的Tag对应，CREATE TABLE所指定的Tag列表必须都出现。必选。
`time`	写入数据的时间，单位ms，仅支持正整数，必选项。
`field_name`	写入数据的列名，建议数据类型与CREATE TABLE时数据类型保持一致。否则以最后写入的数据类型为准。当某列未指定时，默认为null。

> 说明
>
> - 时序引擎对相同时间线的数据采用覆盖写入的方式。
>   当INSERT一行数据时，如果该数据行属于表中已有的时间线，且数据带的时间戳也与该时间线中的已有数据完全相同，那么写入时将不会增加新的数据行，而是会覆盖已有的数据行。
> - insert_select_statement中写入的目标表需要提前创建好，并且需要保证其对应的列能够与语句中的SELECT语句的选择列表中的列的属性与类型一致。

示例

- 单条依次写入：

```
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1260','north-cn','2021-04-22 15:33:00',12.1,45);
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1260','north-cn','2021-04-22 15:33:10',13.2,47);
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1260','north-cn','2021-04-22 15:33:20',10.6,46);
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1261','south-cn','2021-04-22 15:33:00',18.1,44);
insert into sensor (device_id, region, time, temperature, humidity) values('F07A1261','south-cn','2021-04-22 15:33:10',19.7,44);
```

- 批量写入：

```
insert into sensor (device_id, region, time, temperature, humidity) values
    ('F07A1260','north-cn','2021-04-22 15:33:00',12.1,45),
    ('F07A1260','north-cn','2021-04-22 15:33:10',13.2,47),
    ('F07A1260','north-cn','2021-04-22 15:33:20',10.6,46),
    ('F07A1261','south-cn','2021-04-22 15:33:00',18.1,44),
    ('F07A1261','south-cn','2021-04-22 15:33:10',19.7,44);
```

8.7.6.4.2. 基本查询

本文介绍SELECT的基本查询方法。

语法

```
select_statement ::=  SELECT ( select_clause | '*' )
                      FROM table_name
                      WHERE where_clause
                      [ SAMPLE BY time_interval [ FILL fill_option ] ]
                      [ GROUP BY group_by_clause ]
                      [ ORDER BY ordering_clause ]
                      [ LIMIT integer [ OFFSET integer ] ]
select_clause    ::=  selector [ AS identifier ] ( ',' selector [ AS identifier ] )
selector         ::=  tag_name
                      | field_name
                      | function_name '(' [ selector ( ',' selector )* ] ')'
                      | COUNT '(' '*' ')'
where_clause     ::=  relation ( ( AND | OR ) relation )*
relation         ::=  ( field_name | tag_name ) operator term
operator         ::=  '=' | '<' | '>' | '<=' | '>=' | '!=' | IN | CONTAINS | CONTAINS KEY
group_by_clause  ::=  field_name ( ',' field_name )*
ordering_clause  ::=  field_name [ ASC | DESC ] ( ',' field_name [ ASC | DESC ] )*
```

说明

- 查询条件需要带上时间范围：`time > xxx and time < xxx` 。
- 目前limit是后过滤，效率不高，建议您增加限制条件减少扫描的数据量。

示例

- 统计设备ID `F07A1260` 某一时刻上报的所有数据条数：

```
> SELECT count(*) FROM sensor WHERE device_id = 'F07A1260' AND time = '2021-04-22 15:33:00';
+---------+
| EXPR$0 |
+---------+
| 1      |
+---------+
1 rows in set (97 ms)
```

- 查询设备ID `F07A1260` 在一段时间内的所有测点：

```
> SELECT * FROM sensor WHERE device_id = 'F07A1260' AND time >= '2021-04-22 15:33:00' AND time < '2021-04-
22 15:33:20';
+-----------+----------+---------------------------+-------------+-----------+
| device_id |  region  |           time            | temperature | humidity  |
+-----------+----------+---------------------------+-------------+-----------+
| F07A1260  | north-cn | 2021-04-22T15:33:00+08:00 | 12.100000   | 45.000000 |
| F07A1260  | north-cn | 2021-04-22T15:33:10+08:00 | 13.200000   | 47.000000 |
+-----------+----------+---------------------------+-------------+-----------+
2 rows in set (57 ms)
```

- 查询表 `sensor` 中所有的 `device_id` ：

```
> SELECT DISTINCT(device_id) FROM sensor;
+----------+
|          |
+----------+
| F07A1260 |
+----------+
1 rows in set (114 ms)
```

- 查询设备ID `F07A1260` 在一段时间内的最大温度：

```
> SELECT max(temperature) AS max_temp FROM sensor WHERE device_id = 'F07A1260' AND time >= '2021-04-22 15:
33:00' AND time < '2021-04-22 15:33:20';
+-----------+
| max_temp  |
+-----------+
| 13.200000 |
+-----------+
1 rows in set (103 ms)
```

8.7.6.4.3. 降采样查询

降采样查询是在时间维度的聚合查询，是时序数据应用场景常用的降低采样率的查询。本文介绍在时序引擎中降采样查询的语法。

前提条件

时序引擎版本需要3.4.15及以上，升级版本的方法请参见升级小版本。

语法

```
select_sample_by_statement ::=  SELECT ( select_clause | '*' )
                                FROM table_name
                                WHERE where_clause
                                SAMPLE BY (time_interval) [ FILL fill_option ]
                                OFFSET (offset_interval)
select_clause              ::=  selector [ AS identifier ] ( ',' selector [ AS identifier ] )
selector                   ::=  tag_name | time | function_name '(' field_name [ ',' function_args ] ')'
where_clause               ::=  relation ( AND relation )* (OR relation)*
relation                   ::=  ( field_name | tag_name ) operator term
operator                   ::=  '=' | '<' | '>' | '<=' | '>=' | '!=' | IN | CONTAINS | CONTAINS KEY
time_interval              ::= interval units | 0
offset_interval            ::= interval units
```

其中 `time_interval` 说明：

- `interval` ：指数值，如5、60等整数，表示时间线聚合的时间窗口的数值。
- `units` ：指单位，s代表秒，m代表分，h代表小时，d代表天，n代表月，y代表年。

> 说明
> - 默认按照时间戳取模对齐，即“对齐时间戳 = 数据时间戳 -（数据时间戳 % interval）”。
> - 支持基于日历时间间隔的降采样。要使用日历对齐，需要在时间单位units后添加一个 `c` 。例如， `1dc` 代表从当日零点到次日零点之间的24小时；1nc表示从当月零点到次月零点之间的日历月。

降采样支持的聚合函数列表

`SAMPLE BY` 是针对每条独立的时间线（关于时间线概念，请参见名词解释）上的降采样操作。

`SAMPLE BY` 支持的函数列表：

函数	说明
SUM	每个指定时间窗口内求和，具体请参见SUM函数。
AVG	每个指定时间窗口内求均值，具体请参见AVG函数。
COUNT	每个指定时间窗口内值个数，具体请参见COUNT函数。
MIN	每个指定时间窗口内最小值，具体请参见MIN函数。
MAX	每个指定时间窗口内最大值，具体请参见MAX函数。
FIRST	每个指定时间窗口内第一个值，具体请参见FIRST函数。
LAST	每个指定时间窗口内最后值，具体请参见LAST函数。
PERCENTILE	每个指定时间窗口内求百分位，具体请参见PERCENTILE函数。
LATEST	整个时间区间最新值，具体请参见LATEST最新值。
RATE	与前一行对应值的变化率，具体请参见RATE函数。
DELTA	与前一行对应值的差，具体请参见DELTA函数。

示例

SELECT查询指定的tag列无需指定降采样函数，其他field列必须指定降采样函数。

假定查询的sensor表数据如下：

```
SELECT * FROM sensor;

+-----------+----------+---------------------------+-------------+-----------+
| device_id |  region  |           time            | temperature | humidity  |
+-----------+----------+---------------------------+-------------+-----------+
| F07A1260  | north-cn | 2021-01-01T09:00:00+08:00 | 0.000000    | 9.000000  |
| F07A1260  | north-cn | 2021-01-01T12:01:00+08:00 | 1.000000    | 45.000000 |
| F07A1260  | north-cn | 2021-01-01T14:03:00+08:00 | 2.000000    | 46.000000 |
| F07A1260  | north-cn | 2021-01-01T20:00:00+08:00 | 10.000000   | 47.000000 |
| F07A1261  | north-cn | 2021-02-10T12:00:30+08:00 | 3.000000    | 40.000000 |
| F07A1261  | north-cn | 2021-03-01T12:01:00+08:00 | 4.000000    | 41.000000 |
| F07A1261  | north-cn | 2021-03-08T12:08:00+08:00 | 5.000000    | 42.000000 |
| F07A1261  | north-cn | 2021-05-01T13:00:00+08:00 | 6.000000    | 43.000000 |
+-----------+----------+---------------------------+-------------+-----------+
```

降采样与子查询示例

- 默认UTC对齐降采样：时间线按照8H时间区间聚合分别求count。

```
SELECT device_id,region,time,count(humidity) AS count_humidity FROM sensor WHERE device_id='F07A1260' sample
by 8h;

+-----------+----------+---------------------------+----------------+
| device_id |  region  |           time            | count_humidity |
+-----------+----------+---------------------------+----------------+
| F07A1260  | north-cn | 2021-01-01T08:00:00+08:00 | 3              |
| F07A1260  | north-cn | 2021-01-01T16:00:00+08:00 | 1              |
+-----------+----------+---------------------------+----------------+
```

- 默认UTC对齐降采样，指定窗口offset偏移：时间线按照8H时间区间聚合，开始窗口偏移3h，求count。

```
SELECT device_id,region,time,count(humidity) AS count_humidity FROM sensor WHERE device_id='F07A1260' sample
by 8h offset 3h;

+-----------+----------+--------------------------+----------------+
| device_id |  region  |           time           | count_humidity |
+-----------+----------+--------------------------+----------------+
| F07A1260  | north-cn | 2021-01-01T03:00:00+08:00 | 1              |
| F07A1260  | north-cn | 2021-01-01T11:00:00+08:00 | 2              |
| F07A1260  | north-cn | 2021-01-01T19:00:00+08:00 | 1              |
+-----------+----------+--------------------------+----------------+
```

- 本地日历按月聚合降采样。

```
SELECT device_id, time, count(temperature) FROM sensor WHERE device_id='F07A1261' sample by 1nc;

+-----------+---------------------------+--------+
| device_id |           time            | EXPR$2 |
+-----------+---------------------------+--------+
| F07A1261  | 2021-02-01T00:00:00+08:00 | 1      |
| F07A1261  | 2021-03-01T00:00:00+08:00 | 2      |
| F07A1261  | 2021-05-01T00:00:00+08:00 | 1      |
+-----------+---------------------------+--------+
```

- sample by目前不支持与group by、limit offset、order by配合，但可以配合子查询使用。

```
SELECT device_id, max(avg_humidity) AS max_humidity  FROM (SELECT device_id,region,time,avg(humidity) AS avg
_humidity FROM sensor sample by 8h) group by device_id;

+-----------+--------------+
| device_id | max_humidity |
+-----------+--------------+
| F07A1261  | 43.000000    |
| F07A1260  | 47.000000    |
+-----------+--------------+
```

- 配合limit offset限制结果条数。

```
SELECT device_id,region, avg_humidity FROM (select device_id,region,time,avg(humidity) AS avg_humidity FROM
sensor sample by 8h) limit 1 offset 1;

+-----------+----------+--------------+
| device_id |  region  | avg_humidity |
+-----------+----------+--------------+
| F07A1261  | north-cn | 40.000000    |
+-----------+----------+--------------+
```

降采样窗口插值示例

- 固定值插值的用法，示例如下：

```
SELECT * from (select device_id,region,time, avg(humidity) AS humidity FROM sensor WHERE device_id='F07A1260
' sample by 2h fill 1) order by device_id;

+-----------+----------+---------------------------+-----------+
| device_id |  region  |           time            | humidity  |
+-----------+----------+---------------------------+-----------+
| F07A1260  | north-cn | 2021-01-01T08:00:00+08:00 | 9.000000  |
| F07A1260  | north-cn | 2021-01-01T10:00:00+08:00 | 1.000000  |
| F07A1260  | north-cn | 2021-01-01T12:00:00+08:00 | 45.000000 |
| F07A1260  | north-cn | 2021-01-01T14:00:00+08:00 | 46.000000 |
| F07A1260  | north-cn | 2021-01-01T16:00:00+08:00 | 1.000000  |
| F07A1260  | north-cn | 2021-01-01T18:00:00+08:00 | 1.000000  |
| F07A1260  | north-cn | 2021-01-01T20:00:00+08:00 | 47.000000 |
+-----------+----------+---------------------------+-----------+
```

- 后值插值示例如下：

```
SELECT * from (select device_id,region,time,avg(humidity) AS humidity FROM sensor WHERE device_id='F07A1260'
sample by 2h fill after) order by device_id;

+-----------+----------+---------------------------+-----------+
| device_id |  region  |           time            | humidity  |
+-----------+----------+---------------------------+-----------+
| F07A1260  | north-cn | 2021-01-01T08:00:00+08:00 | 9.000000  |
| F07A1260  | north-cn | 2021-01-01T10:00:00+08:00 | 45.000000 |
| F07A1260  | north-cn | 2021-01-01T12:00:00+08:00 | 45.000000 |
| F07A1260  | north-cn | 2021-01-01T14:00:00+08:00 | 46.000000 |
| F07A1260  | north-cn | 2021-01-01T16:00:00+08:00 | 47.000000 |
| F07A1260  | north-cn | 2021-01-01T18:00:00+08:00 | 47.000000 |
| F07A1260  | north-cn | 2021-01-01T20:00:00+08:00 | 47.000000 |
+-----------+----------+---------------------------+-----------+
```

降采样后再转换示例

- 先按照2h窗口avg降采样，然后对降采样后的数据计算 `rate` 斜率，示例如下：

```
SELECT device_id,region,time,rate(avg(humidity)) AS rate_humidity FROM sensor WHERE device_id='F07A1260' sam
ple by 2h;

+-----------+----------+---------------------------+---------------+
| device_id |  region  |           time            | rate_humidity |
+-----------+----------+---------------------------+---------------+
| F07A1260  | north-cn | 2021-01-01T12:00:00+08:00 | 0.002500      |
| F07A1260  | north-cn | 2021-01-01T14:00:00+08:00 | 0.000139      |
| F07A1260  | north-cn | 2021-01-01T20:00:00+08:00 | 0.000046      |
+-----------+----------+---------------------------+---------------+
```

- 先按照2h窗口降采样，然后对降采样后的数据计算差值delta，示例如下：

```
SELECT device_id,region,time,delta(avg(humidity)) AS humidity FROM sensor WHERE device_id='F07A1260' sample
by 2h;

+-----------+----------+---------------------------+-----------+
| device_id |  region  |           time            | humidity  |
+-----------+----------+---------------------------+-----------+
| F07A1260  | north-cn | 2021-01-01T12:00:00+08:00 | 36.000000 |
| F07A1260  | north-cn | 2021-01-01T14:00:00+08:00 | 1.000000  |
| F07A1260  | north-cn | 2021-01-01T20:00:00+08:00 | 1.000000  |
+-----------+----------+---------------------------+-----------+
```

插值

降采样先把所有时间线按照指定时间窗口切分，并把每个降采样区间内的数据做一次运算，降采样后如果某个精度区间没有值，插值可以指定在这个时间点填充具体的值。比如某条时间线降采样后的时间戳为：t+0, t+20, t+30，此时如果不指定插值，只有3个值，如果指定了插值为1，此时间线会有4个值，其中t+10时刻的值为1。
插值函数表：

Fill Policy	填充值
none	默认行为，不填值。
zero	固定填入0。
linear	线性填充值。
previous	之前的一个值。
near	邻近的一个值。
after	之后的一个值。
fixed	用指定的一个固定填充值。

8.7.6.4.4. 最新值查询

本文介绍时序引擎中查询每条时间线最新值的语法。

语法

```
select_latest_by_statement ::=  SELECT ( select_clause )
                                FROM table_name
                                WHERE where_clause
                                SAMPLE BY 0
select_clause              ::=  selector [ AS identifier ] ( ',' selector [ AS identifier ] )
selector                   ::=  tag_name | time | latest '(' field_name [ ',' integer ] ')'
where_clause               ::=  relation ( AND relation )* (OR relation)*
relation                   ::=  ( field_name | tag_name ) operator term
operator                   ::=  '=' | '<' | '>' | '<=' | '>=' | '!=' | IN
```

> 说明
>
> - 最新值查询用于查询匹配时间线的最新时间点的值，支持查询最新的N条数据：`latest(field_name, N)`。
> - 最新值查询需要与`SAMPLE BY 0`配合使用，表示查询每条时间线各自的最新值。
> - 最新值查询不支持与其他聚合函数：rate/delta/max/min等联合使用，例如：`select time, latest(f1), max(f2) from sensor sample by 0`。
> - `latest`函数说明，请参见LATEST最新值。

示例

查询设备`F07A1260`和`F07A1261`的最新温度：

```
lindorm> select device_id,region,time,latest(temperature) as temperature from sensor where device_id in ('F0
7A1260', 'F07A1261') sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-01-01T20:00:00+08:00 | 10.000000   |
| F07A1261  | north-cn | 2021-05-01T13:00:00+08:00 | 6.000000    |
+-----------+----------+---------------------------+-------------+
```

查询设备 `F07A1260` 和 `F07A1261` 的最新温度和最新湿度：

```
lindorm> select device_id,region,time,latest(temperature) as temperature, latest(humidity) as humidity from
sensor where device_id in ('F07A1260', 'F07A1261') sample by 0;
+-----------+----------+---------------------------+-------------+-----------+
| device_id |  region  |           time            | temperature | humidity  |
+-----------+----------+---------------------------+-------------+-----------+
| F07A1260  | north-cn | 2021-01-01T20:00:00+08:00 | 10.000000   | 47.000000 |
| F07A1261  | north-cn | 2021-05-01T13:00:00+08:00 | 6.000000    | 43.000000 |
+-----------+----------+---------------------------+-------------+-----------+
```

latest查询多列的最新值时，如果多个列最新值在时间上无法对齐，则对应填写null，如下示例：

```
lindorm> select device_id,region,time,latest(temperature) as temperature, latest(humidity) as humidity from
sensor where device_id in ('F07A1260') sample by 0;
+-----------+----------+---------------------------+-------------+-----------+
| device_id |  region  |           time            | temperature | humidity  |
+-----------+----------+---------------------------+-------------+-----------+
| F07A1260  | north-cn | 2021-01-01T12:00:00+08:00 | 15.200000   | null      |
| F07A1260  | north-cn | 2021-01-01T13:00:00+08:00 | null        | 47.000000 |
+-----------+----------+---------------------------+-------------+-----------+
```

查询设备 `F07A1260` 和 `F07A1261` 最新的两条温度测点：

```
lindorm> select device_id,region,time,latest(temperature, 2) as temperature from sensor where device_id in (
'F07A1260', 'F07A1261') sample by 0;
+-----------+----------+---------------------------+-------------+
| device_id |  region  |           time            | temperature |
+-----------+----------+---------------------------+-------------+
| F07A1260  | north-cn | 2021-01-01T14:03:00+08:00 | 2.000000    |
| F07A1260  | north-cn | 2021-01-01T20:00:00+08:00 | 10.000000   |
| F07A1261  | north-cn | 2021-03-08T12:08:00+08:00 | 5.000000    |
| F07A1261  | north-cn | 2021-05-01T13:00:00+08:00 | 6.000000    |
+-----------+----------+---------------------------+-------------+
```

> **说明** 最新值查询不能直接与offset、limit、group by field等子句一起使用，但可以通过子查询的方式配合使用。

latest配合group by进行分组：

```
lindorm> select device_id, max(temperature) as max_temperature from (select device_id, region,time, latest(t
emperature,2) as temperature from sensor sample by 0) group by device_id;
+-----------+-----------------+
| device_id | max_temperature |
+-----------+-----------------+
| F07A1261  | 6.000000        |
| F07A1260  | 10.000000       |
+-----------+-----------------+
```

latest配合limit、offset限制结果条数：

```
lindorm> select device_id, region, temperature from (select device_id, region, time, latest(temperature,2) a
s temperature from sensor sample by 0) limit 1 offset 1;
+-----------+----------+-------------+
| device_id |  region  | temperature |
+-----------+----------+-------------+
| F07A1260  | north-cn | 10.000000   |
+-----------+----------+-------------+
```

8.7.6.5. DCL

8.7.6.5.1. GRANT

用户授权的语法介绍。

语法

用户授权的语义表达是“将某个对象的指定权限赋予给指定用户”。其语法如下：

```
grant_statement       ::=  GRANT privilege_definition ON resource_definition TO user_name
privilege_definition  ::=  ALL [PRIVILEGES] | READ | WRITE | ADMIN | SYSTEM
resource_definition   ::=  GLOBAL | DATABASE database_name
```

参数说明

参数	说明
user_name	权限被授予的用户名。必须是SQL标识符。
privilege_definition	被授予的权限。取值为以下类型： • READ 具备“查询”权限。 • WRITE 具备“写入”权限。 • ADMIN 具备“管理”权限。 • SYSTEM 具备对于集群层面的管理权限。其内涵包含了 **GLOBAL ADMIN**权限。 • ALL 或 ALL PRIVILEGES 具备“所有”权限。包含READ、WRITE、ADMIN。
resource_definition	被授权的资源范围。资源粒度如下： • GLOBAL 授予全局所有资源的特定权限。 • DATABASE 授予指定Database的指定权限。
database_name	当授权的资源范围为Database时，表示具体的授权Database名。必须是SQL标识符。

> 注意
>
> 1. 权限的授权范围优先级：GLOBAL > DATABASE。
> 即如果拥有了GLOBAL的READ权限，则能查询任意Database中的时序表；但如果只是拥有特定Database的READ权限，则只能查询该Database下的时序表。**建议不要轻易赋予 GLOBAL权限，特别是GLOBAL ALL PRIVILEGES。**
> 2. ADMIN权限与READ、WRITE权限是独立权限，授予了某个Database的ADMIN权限并不意味着就能够读写该Database下的时序表。
> 3. 不支持授予DATABASE粒度的SYSTEM权限。
> 4. 不支持授予一个不存在的资源对象的权限；同样，也不支持向一个不存在的用户授权。查看已有资源的方法可参见SHOW 语句。
> 5. GRANT语句是否可以执行成功，还取决于执行者自身拥有的权限。详细请参见用户及权限管理。

示例

以下例子展示了给一个名为 `tsdbuser` 的用户赋予名为 `DB1` 的Database的读权限。

```
GRANT READ ON DATABASE `DB1` TO `tsdbuser`;
```

以下例子展示了给一个名为 `tsdbuser` 的用户赋予名为 `DB2` 的Database的写权限。

```
GRANT WRITE ON DATABASE `DB2` TO `tsdbuser`;
```

以下例子展示了给一个名为 `tsdbuser` 的用户赋予名为 `DB3` 的Database的全部权限。

```
GRANT ALL PRIVILEGES ON DATABASE `DB3` TO `tsdbuser`;
```

8.7.6.5.2. REVOKE

权限回收的语法介绍。

语法

权限回收的语义表达是“将某个对象的指定权限从指定用户收回”。其语法如下：

```
revoke_statement      ::=  REVOKE privilege_definition ON resource_definition FROM user_name
privilege_definition  ::=  ALL [PRIVILEGES] | READ | WRITE | ADMIN | SYSTEM
resource_definition   ::=  GLOBAL | DATABASE database_name
```

参数说明

参数	说明
user_name	权限撤回操作的对象用户名。必须是SQL标识符。
privilege_definition	被撤回的权限。取值为以下类型： • READ 具备“查询”权限。 • WRITE 具备“写入”权限。 • ADMIN 具备“管理”权限。 • SYSTEM 具备对于集群层面的管理权限。其内涵包含了 **GLOBAL ADMIN**权限。 • ALL 或 ALL PRIVILEGES 具备“所有”权限。包含READ、WRITE、ADMIN。
resource_definition	被撤回权限的资源范围。资源粒度如下： • GLOBAL 撤回全局所有资源的特定权限。 • DATABASE 撤回指定Database的特定权限。
database_name	当撤回权限的粒度为Database时，表示撤回权限所属的Database名。必须是SQL标识符。

> **注意**
> REVOKE语句是否可以执行成功，还取决于执行者自身拥有的权限。详细请参见用户及权限管理。

示例

以下例子展示了将一个名为 `DB1` 的Database的读权限从一个名为 `tsdbuser` 的用户处收回。

```
REVOKE READ ON DATABASE `DB1` FROM `tsdbuser`;
```

以下例子展示了将一个名为 `DB2` 的Database的写权限从一个名为 `tsdbuser` 的用户处收回。

```
REVOKE WRITE ON DATABASE `DB2` FROM `tsdbuser`;
```

以下例子展示了将一个名为 `DB3` 的Database的所有已赋予权限从一个名为 `tsdbuser` 的用户处收回。

```
REVOKE ALL PRIVILEGES ON DATABASE `DB3` FROM `tsdbuser`;
```

8.7.7. 问题排查

8.7.7.1. 常见错误码参考

通过SQL访问时序引擎时常见的错误码即其释义可参考下文。

权限错误

ErrorCode	SQLSTATE	错误消息格式	处理建议
1227	42501	%s	确认所使用用户的真实权限，确保对访问对象拥有足够的权限。

DDL错误

ErrorCode	SQLSTATE	错误消息格式	处理建议
1006	HY000	Can't create database '%s' (errno: %d - %s)	无法创建database, 请根据错误消息，检查输入的参数是否正确。
1007	HY000	Can't create database '%s'; database exists	该database已存在，请使用别的名称创建。
1008	HY000	Can't drop database '%s'; database doesn't exist	该database不存在，无法进行删除，请提供存在的database名称。
1009	HY000	Error dropping database (can't delete '%s', errno: %d - %s)	删除database失败，请从错误消息中获取失败原因； 若依然无法解决，请联系产品工程师寻求解决方法。
1045	28000	%s	确认所用用户的认证信息是否存在记忆错误，笔误等情况并输入正确的认证信息。
1049	42000	Unknown database '%s'	未知database, 请指定要访问的database。
1050	42S01	Table '%s' already exists	该表已经存在，请使用其他表名称。
1064	42000	You have an error in your SQL syntax; %s	SQL语句中存在语法错误，请参考SQL开发文档进行校正。
1146	42S02	Table '%s.%s' doesn't exist	该表不存在，请检查是否输入的名称有误。

ErrorCode	SQLSTATE	错误消息格式	处理建议
8000	HY000	Can't alter database '%s' (errno: %d - %s)	无法修改database, 请根据错误消息，检查输入的参数是否有误。
8001	HY000	Can't create continuous query '%s' (errno: %d - %s)	无法创建连续查询（continuous query），请根据错误消息，检查输入的参数是否有误。
8002	HY000	Can't create continuous query '%s'; continuous query exists	该连续查询（continuous query）已经存在，请使用其他名称进行创建。
8004	HY000	Continuous query '%s.%s' doesn't exist	该连续查询（continuous query）不存在，请检查输入的名称是否有误。

DML错误

ErrorCode	SQLSTATE	错误消息格式	处理建议
3024	HY000	Query execution was interrupted, maximum statement execution time exceeded	查询超时，请重试； 若重试依然超时，请联系产品工程师寻求解决方法。
8005	HY000	Table '%s' not found	表不存在，请检查表的名称是否有误。
8006	HY000	Query failed; %s	查询失败， 请根据错误消息分析错误原因； 若依然无法解决，请联系产品工程师寻求解决方法。
8007	HY000	Insert failed; %s	写入失败，请根据错误消息分析错误原因； 若依然无法解决，请联系产品工程师寻求解决方法。
8008	HY000	Quota exceeded; %s	超出资源限制，请根据错误消息提示，若是查询请缩小查询时间范围，或者增加where条件，减少查询命中的数据量；若是写入，请限制写入TPS。

系统错误

ErrorCode	SQLSTATE	错误消息格式	处理建议
9000	HY000	Server internal error; %s	请联系产品工程师寻求解决方法。
9001	0A000	UNSUPPORTED: %s	对照时序引擎的SQL语法参考，避免使用尚不支持的SQL语法。

8.8. 数据生态

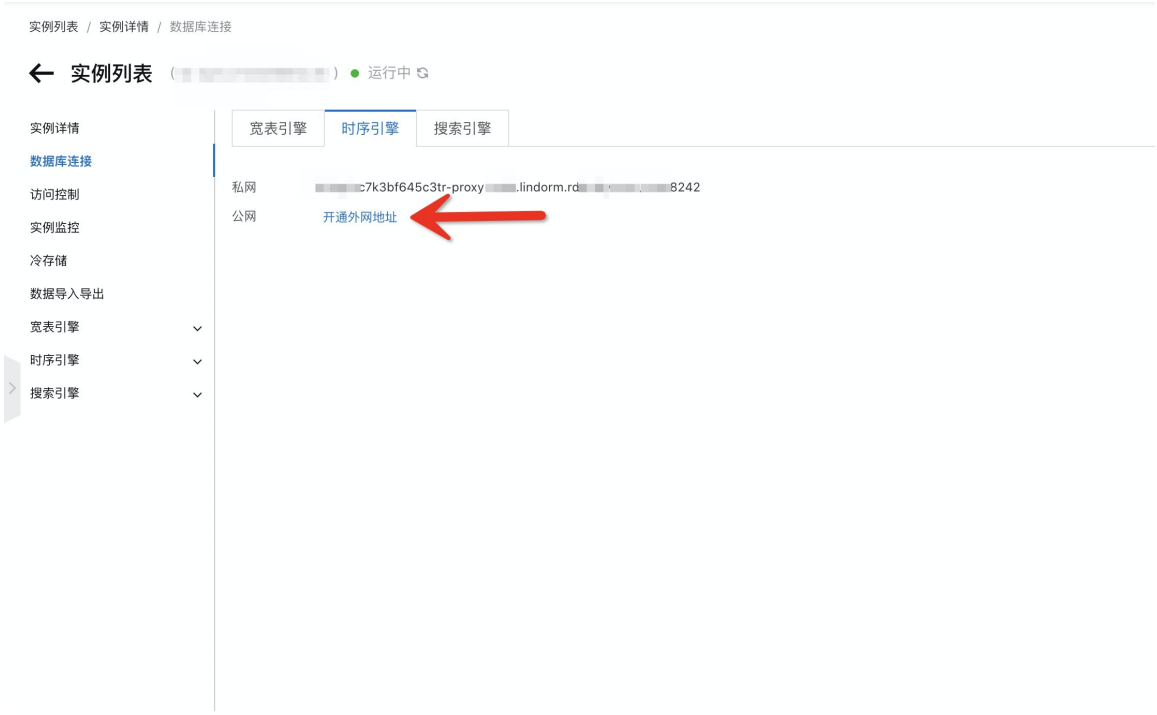
8.8.1. 使用Telegraf写入数据到Lindorm

Telegraf是一款开源的数据采集代理。其设计目标是通过插件来构建各种服务和第三方组件的 metrics 收集。Telegraf的内置插件支持将采集的数据上报给OpenTSDB，Lindorm时序引擎全面兼容OpenTSDB，用户可以使用Telegraf来对接时序引擎。

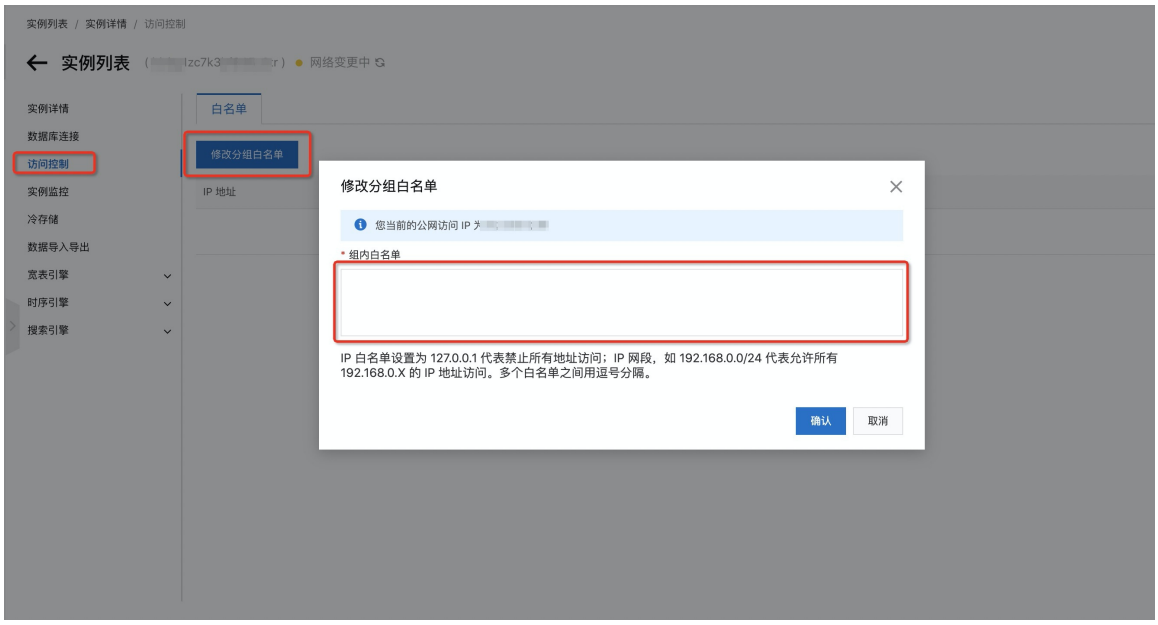
本文将介绍如何在用户的已有应用环境中部署 Telegraf 监控代理，并对接公有云时序引擎实例使之接收 Telegraf 的上报数据。

事前准备

1. 确保用户已有的待检测应用环境（以下简称”应用环境”）已经连接上公网，并获得了公网IP。
2. 根据需要申请公有云的时序引擎实例，并为它申请公网地址，并在网络白名单中填入上述应用环境的公网IP。
   - 时序引擎实例管理界面中申请公网地址的入口。



   - 网络白名单配置界面。



3. 时序引擎公网地址申请后，确认从应用环境可以连接到该时序引擎实例的公网域名。时序引擎实例域名以及端口会在公网地址申请成功后显示在下述位置。



安装Telegraf

请结合应用环境所属的平台种类，参见Telegraf产品文档选择适合自身平台的安装方式。

配置Telegraf连接时序引擎

1. 修改 Telegraf 的配置文件 `telegraf.conf` ，找到其中的**OUTPUT PLUGINS**部分。
2. 关闭掉 output.influxdb的插件，启用output.opentsdb插件。

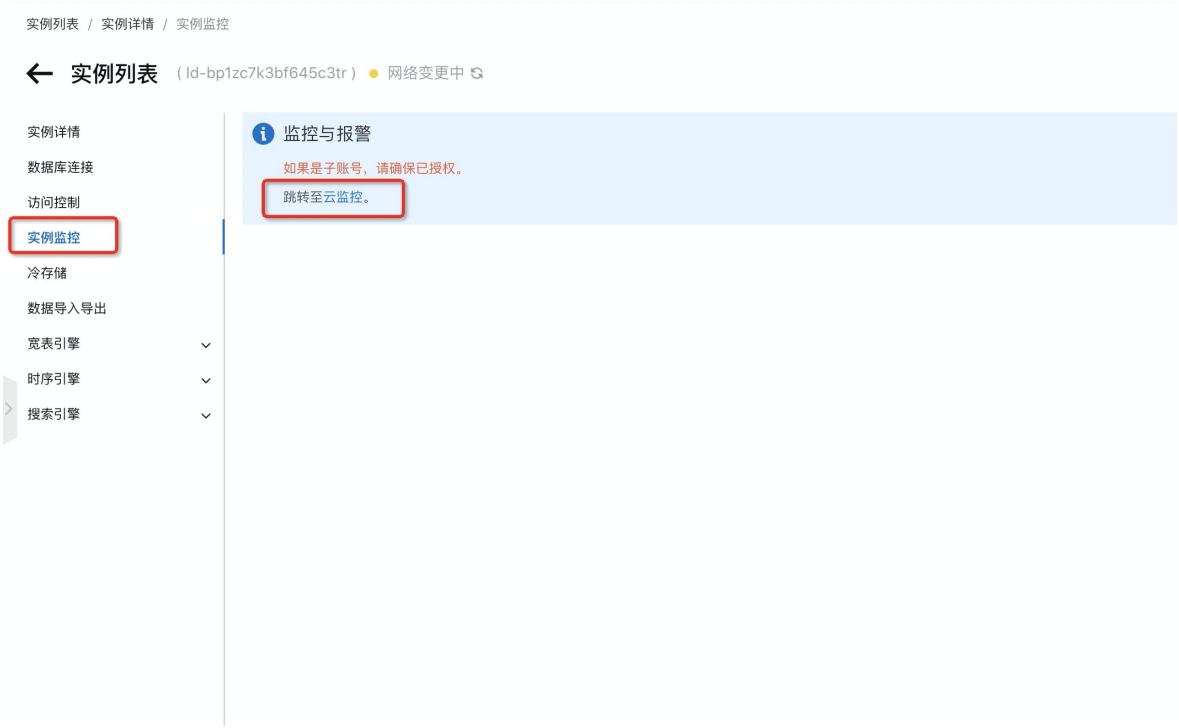
```
......
#[[outputs.influxdb]]
~（下略）~
[[outputs.opentsdb]]
......
```

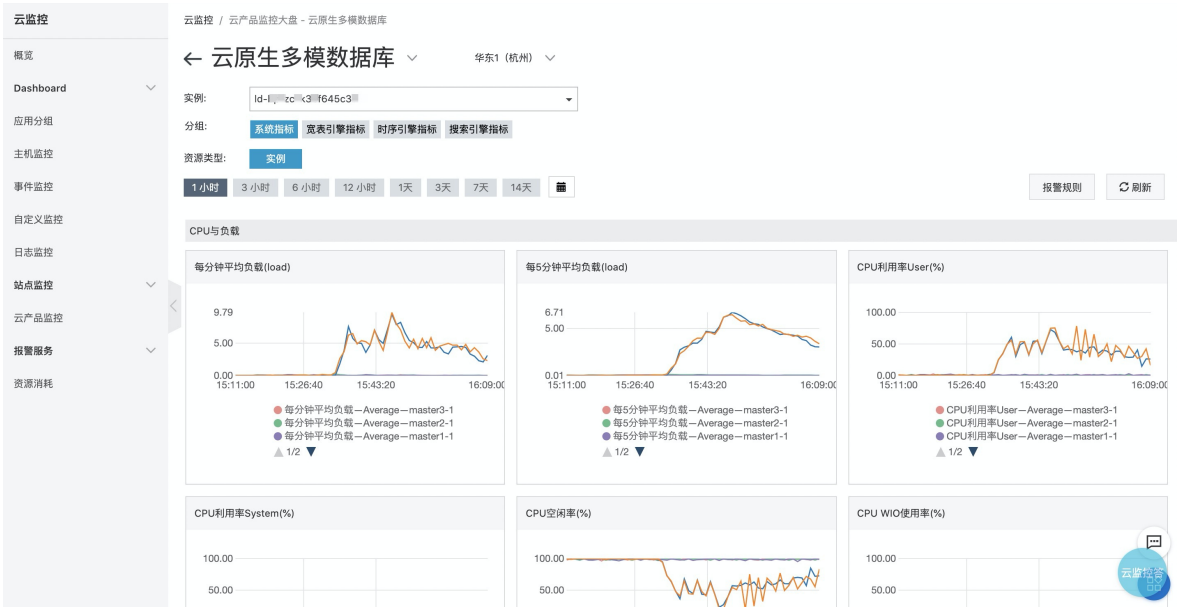
3. 配置[[outputs.opentsdb]]的相关设置。
   - 配置时序引擎实例的域名和端口。填入类似以下的内容。

```
host = "http://{时序引擎实例ID}.hitsdb.rds.aliyuncs.com"
port = 3242
```

> **说明** host必须加上“http://”前缀。3242 是公有云时序引擎实例在公网暴露的默认端口。

   - 配置数据上报时的HTTP接口。可通过打开http_path = "/api/put"注释来实现。同时可以通过设置http_batch_size来指定上报数据时一次批量写入的数据点数。
   - 配置Telegraf写入指标时的分隔符。可通过设置separator实现指标的分隔符。但需注意分隔符只能选用Lindorm时序引擎所支持的单值数据写入。
   - 关于其他可配置项的设置，请参见Telegraf 在线文档。
4. 启动Telegraf，开启数据上报。 数据上报开始后，可以从时序引擎实例的管控平台的监控页面，看到数据的写入情况。监控页面的入口为 **阿里云时序引擎 > 管理控制台 > 实例列表 > 实例监控**。





8.8.2. 通过Grafana访问Lindorm时序引擎

云原生多模数据库Lindorm时序引擎可以通过Grafana服务的Alibaba Cloud Lindorm数据源插件进行访问。本文介绍通过Grafana服务连接时序引擎并可视化分析数据。

背景信息

Grafana服务是用于展示大型测量数据的开源可视化工具，在互联网应用分析中应用范围十分广泛，并且在工业监控、气象监控、家居自动化和过程管理等领域也有着较广泛的用户基础。

准备工作

部署方式	前提条件	部署要求

部署方式	前提条件	部署要求
自建Grafana服务（客户端部署在ECS上）	• 确保ECS实例与Lindorm实例属于同一专有网络。 • 确保ECS实例已开通公网访问。如果您未分配公网IP地址，可以申请弹性公网IP，具体操作，请参见申请弹性公网IP。 • 将ECS实例的内网IP地址添加至Lindorm实例的白名单中。具体操作，请参见设置白名单。 • ECS实例已添加端口为3000的安全组规则。具体操作，请参见添加安全组规则。	• 客户端的操作系统版本选择CentOS 7.3 64位及以上版本。 • 客户端的硬件资源需要单核2 G内存以上，磁盘剩余空间至少有10 G。
阿里云Grafana服务	阿里云Grafana服务目前正在公测，如果您想体验Grafana服务，可以在Grafana服务页面申请开通。	无

操作步骤

以下操作步骤是自建Grafana服务（客户端部署在ECS上）。您也可以通过阿里云Grafana服务创建Grafana托管服务，具体操作请参见创建工作区。

1. 安装Grafana服务（如已安装，可跳过）。
   i. 登录ECS实例。
   ii. 执行下述命令下载Grafana服务：

   ```
   wget https://dl.grafana.com/enterprise/release/grafana-enterprise-8.2.2-1.x86_64.rpm
   ```

   > **说明** 本文以Grafana 8.2.2-1版本为例演示操作流程，您也可以安装其他版本。具体操作，请参见Grafana官网。

   iii. 执行下述命令安装Grafana服务：

   ```
   sudo yum install grafana-enterprise-8.2.2-1.x86_64.rpm
   ```

   > **说明** 您也可以参考官网安装Grafana服务。

   iv. 执行下述命令启动Grafana服务：

   ```
   sudo service grafana-server start
   ```

2. 安装Alibaba Cloud Lindorm数据源插件。
   i. 执行下述命令下载数据源插件，数据源插件的链接请单击下载。

   ```
   wget https://tsdbtools.oss-cn-hangzhou.aliyuncs.com/grafana-alibabacloud-lindorm-datasource-latest.zip
   ```

   ii. 执行下述命令解压插件到默认目录，Linux的默认目录为*/var/lib/grafana/plugins/*。

   ```
   unzip grafana-alibabacloud-lindorm-datasource-latest.zip
   ```

   iii. 执行如下命令修改*grafana.ini*文件信息，Linux默认文件位置*/etc/grafana/grafana.ini*。

   ```
   sudo nano -w  /etc/grafana/grafana.ini
   ```

   iv. 在**[plugins]**区域中将 `;allow_loading_unsigned_plugins=` 修改为 `allow_loading_unsigned_plugins = alibabacloud-lindorm-datasource` 。修改后按下*CTRL+O*保存并回车退出编辑页面。

   > **注意** 修改*grafana.ini*文件时，需要删除 `;allow_loading_unsigned_plugins` 前面的半角分号（;）。

   v. 执行如下命令重启Grafana服务：

   ```
   sudo service grafana-server restart
   ```

3. 将Grafana服务与Lindorm实例连接。

i. 登录Grafana服务并进入Grafana主页。具体操作请参见登录Grafana。

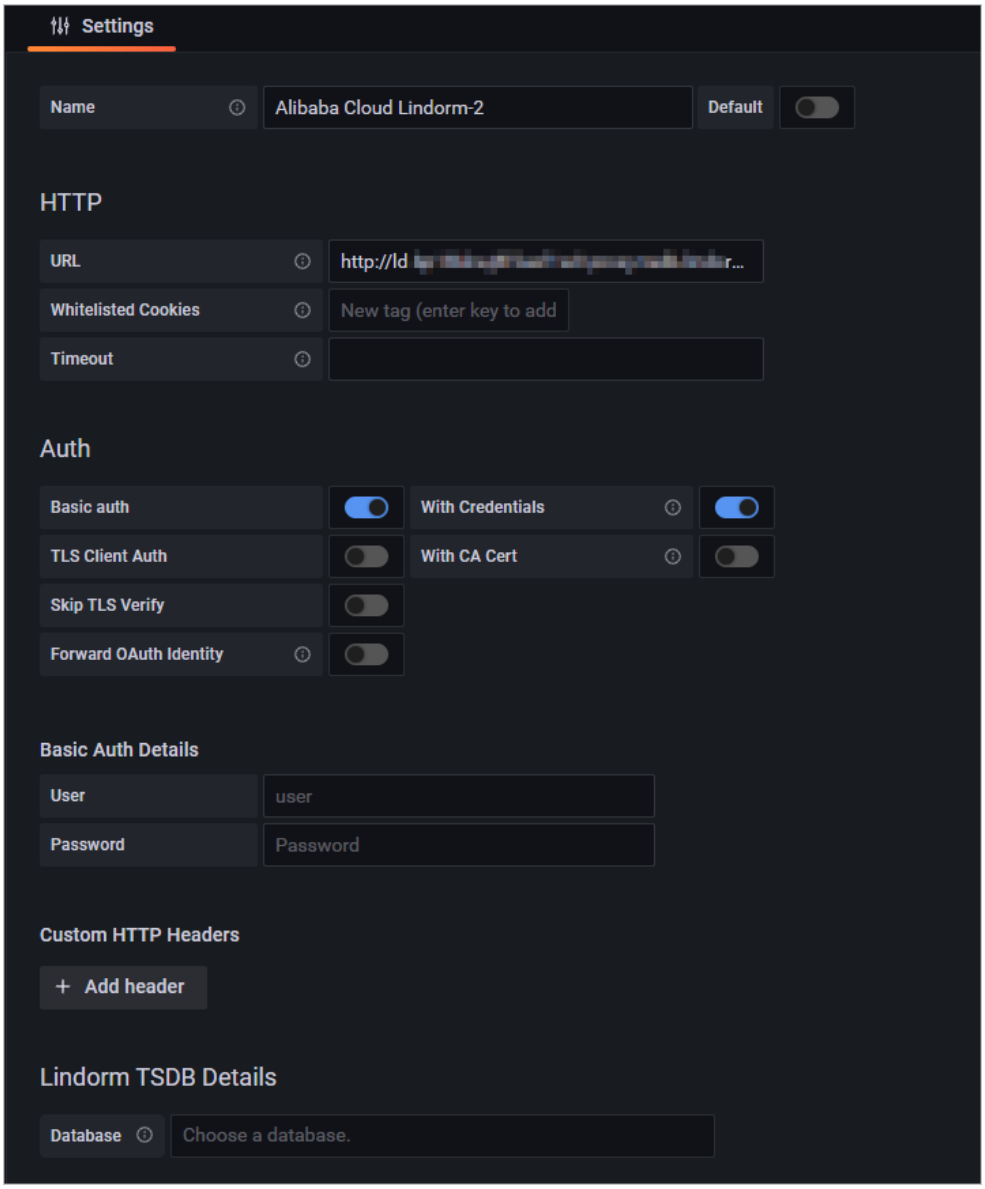
> **说明** 如果Grafana服务部署在阿里云Grafana服务，您可以通过阿里云Grafana服务登录Grafana主页，具体操作请参见管理工作区。

ii. 单击左侧的  图标，在列表中选择**Data Sources**。

iii. 进入**Configuration**页面，单击**Add data source**。

iv. 在**Add data source**页面的搜索框中输入*Alibaba Cloud Lindorm*，找到目标数据源并单击**Select**。

v. 在Settings页签中配置以下参数，其他参数选择默认值。

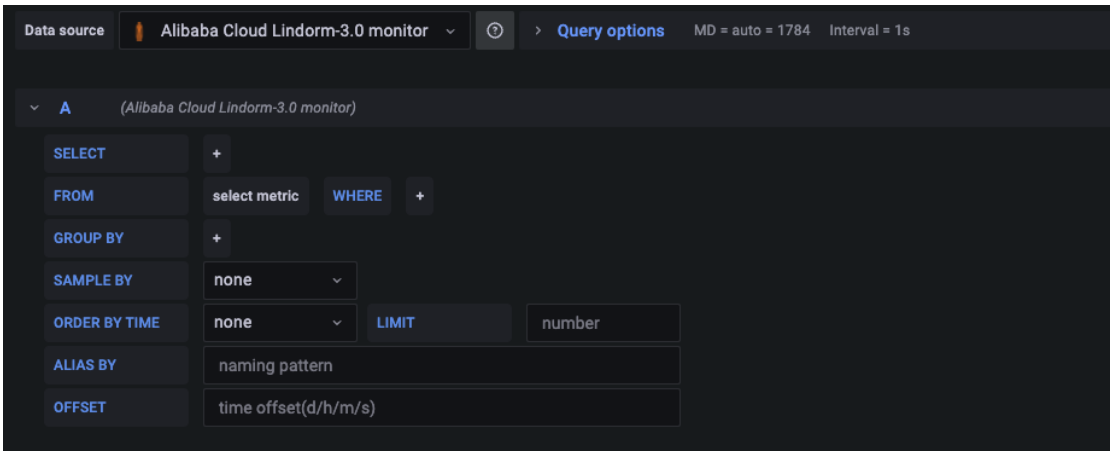


配置项	参数	描述
Name		数据源名称，可自定义。
HTTP	URL	Lindorm时序引擎的连接地址，获取方法请参见操作步骤。
Auth	Basic auth	如果需要使用鉴权认证请打开Basic auth。
	With Credentials	如果需要使用鉴权认证请打开With Credentials。
Basic Auth Details	User	如果打开Basic auth需要填写Lindorm实例的用户名。
	Password	如果打开Basic auth需要填写Lindorm实例的密码。
Lindorm TSDB Details	Database	填写需要访问的Lindorm实例中的数据库。

vi. 单击Save & test。

4. 连接Lindorm实例后，在Grafana服务中创建并使用dashboard，具体操作您可以参考官网文档。

5. 在Query页签中通过TSQL查询数据。
   i. 在Data source列表中选择目标数据源名称。
   ii. TSQL查询操作有两种方式，包括使用编辑框查询和使用TSQL语句查询。
   - 使用编辑框查询：在A区域下的选择框选择需要查询的数据，可以完成简单的查询操作，关键字说明如下表。



关键字	说明
SELECT	选择需要查询的字段名。
FROM	选择需要查询的表名。
WHERE	添加查询条件。
GROUP BY	添加聚合条件。
SAMPLE BY	选择降采样时间间隔，具体请参见降采样查询。
ORDER BY TIME	选择排列顺序。
LIMIT	输入查询返回最多的数据数量。
ALIAS BY	重命名时间线标签。
OFFSET	设置时间偏移量，方便您将不同时间段的数据进行对比。

   - 使用TSQL语句查询：单击A区域右上角的，可以通过输入TSQL语句完成复杂的查询操作（例如嵌套查询）。

```
SELECT time,host,value FROM cpu_user WHERE instanceId = '$instanceId' AND host similar to 'tsdb' a
nd time >= $from AND time <= $to
```

> **说明** TSQL语句使用请参见SELECT语法。
> - time >= $from AND time <= $to 表示时间条件，会自动替换为页面中选择的时间范围。
> - time 为必选项，否则不会出现分析结果图。
> - SELECT语句中通过tag方式对不同时间线进行分组聚合。
> - 通过ALIAS BY重命名时间线标签，例如使用 $tag_host 进行区分时间线。
> - 通过OFFSET设置时间偏移量，方便您将不同时间段的数据进行对比。

iii. TSQL语句中的 `$interval` 表示降采样时间间隔，语句示例如下：

```
SELECT region,time,avg(temperature) FROM sensor WHERE time >= $from AND time <= $to SAMPLE BY $interval
```

> 说明 单击Query options，通过Min interval和Max data points参数来设置Interval参数大小。

8.8.3. InfluxDB®行协议写入

本文介绍通过InfluxDB®行协议将数据写入Lindorm时序引擎的方法和示例。

背景信息

Lindorm时序引擎兼容InfluxDB®的行协议（Line Protocol）格式。对于通过行协议发送数据的应用或设备，都可以将数据点写入Lindorm时序引擎中。

支持的数据类型

行协议写入语法的数据类型包括Float、Integer、String、Boolean和Timestamp。数据类型说明参见下表。

数据类型	说明
Float	默认解析类型，例如：1、1.0、1.e+10、1.E+10。
Integer	支持64位整数，通过后缀i来识别，例如：12i、101i。
String	字符串类型，使用双引号（“”）表示，例如：“hello world”。
Boolean	取值包括True和False。 • True支持的语法包括t、T、true、True、TRUE。 • False支持的语法包括f、F、false、False、FALSE。
Timestamp	时间戳。Timestamp目前协议上支持ns，数据存储时转换为ms存储。如果您需要修改时间戳精度，可通过**precision**来设置。

请求路径

- 请求路径：/api/v2/write
- 请求方法：POST

发送行协议写入请求时，需要将数据按行协议写入格式编写后作为POST请求的请求体填入，参数说明如下：

参数	是否必选	说明
precision	否	默认值ns，时间戳精度，取值包括ns、us、ms、s、m、h。
db	否	数据写入的数据库名。

行协议写入格式

行协议写入格式和参数说明如下：

```
<table_name>[,<tag_key>=<tag_value>[,<tag_key>=<tag_value>]] <field_key>=<field_value>[,<field_key>=<field_value>] [<timestamp>]
```

参数	是否必选	说明
table_name	是	数据写入的表名。
tag_set	否	数据标签，可用于搜索。
field_set	是	数据字段。

参数	是否必选	说明
timestamp	否	时间戳，默认为当前时间，默认单位为ns。 说明 如果您需要修改时间戳精度，可通过precision来设置。

说明
- 一次写入多条数据使用反斜线n（\n）分隔。
- 通过行协议将数据写入Lindorm时序引擎时，时序引擎的Schema约束行为属于弱约束。

示例
行协议写入的语句示例如下。

```
POST /api/v2/write?precision=ms&db=default
sensor,device_id=F07A1260,region=north-cn temperature=12.1,humidity=45 1619076780000
sensor,device_id=F07A1260,region=north-cn temperature=13.2,humidity=47 1619076790000
sensor,device_id=F07A1260,region=north-cn temperature=10.6,humidity=46 1619076800000
sensor,device_id=F07A1260,region=north-cn temperature=18.1,humidity=44 1619076780000
sensor,device_id=F07A1260,region=north-cn temperature=19.7,humidity=44 1619076790000
```

行协议写入成功后返回状态码204。

8.8.4. 对接Prometheus
Lindorm时序引擎提供remote write/remote read接口实现作为Prometheus的远端存储。同时原生支持PromQL访问接口以提供更加高效的数据访问。

写入数据

获取数据库连接地址
1. 登录Lindorm控制台。
2. 在**实例列表**单击目标实例ID，进入**实例详情**页面。
3. 单击**数据库连接>时序引擎**进入**时序引擎**页面。
4. 在**时序引擎**页面查看**私网**地址或**公网**地址。



remote read和remote write的连接地址分别为：

http://ld-****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/prom_write

http://ld-****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/prom_read

配置Prometheus远端存储

打开Prometheus.yaml配置文件，并在文件末尾增加以下内容，将**remote_write**和**remote_read**链接替换为上一步获取的地址。

```
remote_write:
  - url: "http://ld-****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/prom_write"
remote_read:
  - url: "http://ld-****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242/api/prom_read"
    read_recent: true
```

使用PromQL读取数据

Lindorm时序引擎提供PromQL的原生支持，例如可以使用Grafana的PromQL插件来访问Lindorm时序引擎。PromQL访问链接地址获取方式如下：

1. 登录Lindorm控制台。
2. 在**实例列表**单击目标实例ID，进入**实例详情**页面。
3. 单击**数据库连接>时序引擎**进入**时序引擎**页面。
4. 在**时序引擎**页面点击PromQL兼容地址查看**私网**地址或**公网**地址。



8.8.5. 使用Flink写入时序引擎

本文介绍如何在实时计算Flink上创建时序结果表并写入至云原生多模数据库Lindorm时序引擎。

前提条件

- 已购买实时计算Flink版或者已有自建Flink，实时计算Flink版的购买请参见快速入门。

  > **说明** 实时计算Flink需要VVR 4.0.13及以上版本，VVR 4.0.13版本是基于Apache Flink 1.13。

- 云原生多模数据库Lindorm实例已开通时序引擎功能。
- Lindorm时序引擎版本为3.4.7及以上版本，查看方法请参见时序引擎版本说明。
- 为了保证网络的连通性，确保云原生多模数据库Lindorm实例和实时计算Flink使用相同的专有网络。

  > **说明** 如果您需要通过公网使用实时计算Flink写入时序引擎，请确保实时计算Flink具备公网访问能力并配置Lindorm白名单，请参见Flink全托管集群如何访问公网。

背景信息

时序引擎SINK是一种将时序结果表写入SINK连接器的插件，支持将多种数据源的数据高效写入至Lindorm时序引擎并进行业务分析。使用时序引擎SINK插件，需要先获取时序引擎SINK插件，再将JAR包上传至实时计算Flink版控制台，上传方法请参见Connector使用。

语法

在实时计算Flink上创建时序结果表，创建语法代码如下：

```
CREATE TEMPORARY TABLE tsdb_sink(
  `timestamp` BIGINT,
  tag_tagname VARCHAR,
  field_fieldname1 DOUBLE,
  field_fieldname2 VARCHAR,
  field_fieldname3 BIGINT,
  field_fieldname4 BOOLEAN
  -- table VARCHAR（可选字段）
)
with (
    'connector' = 'lindormtsdb',
    'url'='http://ld-bp159jt4eivt3****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242',
    'table'='mytable',
    'schemaPolicy'='weak',
    'sync'='false',
    'debug'='false'
);
```

参数说明

默认格式

字段名	数据类型	是否必选	说明
timestamp	BIGINT	是	单位为毫秒。 **说明** 字段值为13位时间戳，如果是10位时间戳，写入会自动转换为13位。
tag_tagname	VARCHAR	是	指定时序数据标签（Tag）。tag_表示前缀（不能省略），tagname表示时序数据标签名称。 **说明** tag_tagname为一列或者多列。
field_fieldname	DOUBLE、VARCHAR、BIGINT、BOOLEAN	是	指定时序数据量测值（Field）。field_表示前缀（不能省略），fieldname表示时序数据量测值名称。 **说明** field_fieldname为一列或者多列。
table	VARCHAR	否	指定时序数据表。 • 如果写入一张时序数据表，建议在WITH参数中配置。 • 如果写入多张表可以在该字段中自定义表名。

WITH参数

参数	是否必选	说明
connector	是	固定值lindormtsdb，指定时序引擎SINK插件。

参数	是否必选	说明
url	是	Lindorm时序引擎的连接地址，获取方法请参见查看连接地址。
table	否	指定时序数据表。 • 如果写入一张时序数据表，建议配置此参数。 • 如果写入多张表可以在table字段中自定义表名。
username	否	默认情况未启用时序引擎的用户认证与权限校验，访问时序引擎时无需输入用户名和密码。但是为了数据安全，建议开启时序引擎的用户认证与权限校验，开启后访问时序引擎都需要输入用户名和密码。通过SQL创建用户和修改用户密码请参见CREATE USER和ALTER USER。
password	否	
defaultDatabase	否	读写数据库，默认值为default。
schemaPolicy	否	表的Schema约束策略，可以设置为： • Strong：默认值，强约束，时序引擎会严格依据预先定义的表结构对写入数据的表名、字段名、类型进行校验。 • Weak：弱约束，写入数据的表不存在时引擎不会报错，而是会自动创建对应的表。 • None：无约束，无法直接通过SQL查询写入的数据。 **说明** 设置为Strong或者None时需要手动建表。更多信息请参见关于时序数据的Schema约束。
ignoreErrorData	否	是否忽略写入错误。 • false：默认值，不忽略，如果遇到错误就跳出写入操作。 • true：忽略，如果遇到错误就忽略继续写入操作。
maxRetries	否	写入时遇到服务端内部错误或者网络错误时最大重试次数，默认值为3。
batchSize	否	写入时每次向服务端发送的数据量，默认值为500个数据点。
connectTimeoutMs	否	HTTP连接超时时间，默认值为90000。
debug	否	是否开启debug模式，用来打印详细数据点日志。 • false：默认值，不开启。 • true：开启。
sync	否	是否同步写入，建议使用false。 • false：异步写入，默认值，写入效率高。 • true：同步写入。

示例

以datagen_source随机数据生成器为例，将生成的数据写入Lindorm时序表tsdb_sink中。示例代码如下：

```
CREATE TEMPORARY TABLE datagen_source (
  id INTEGER,
  score DOUBLE,
  name STRING
)
WITH (
  'connector' = 'datagen'
);
CREATE TEMPORARY TABLE tsdb_sink(
  tag_tagk VARCHAR,
  field_score DOUBLE,
  field_name STRING,
  `timestamp` BIGINT
)
WITH (
    'connector' = 'lindormtsdb',
    'url'='http://ld-bp159jt4eivt3****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242',
    'table'= 'mytable'
);
INSERT INTO tsdb_sink
SELECT
  CAST(id as STRING) as tag_tagk,
  score as field_score,
  name as field_name,
  UNIX_TIMESTAMP(now()) * 1000  as `timestamp`
FROM datagen_source;
```

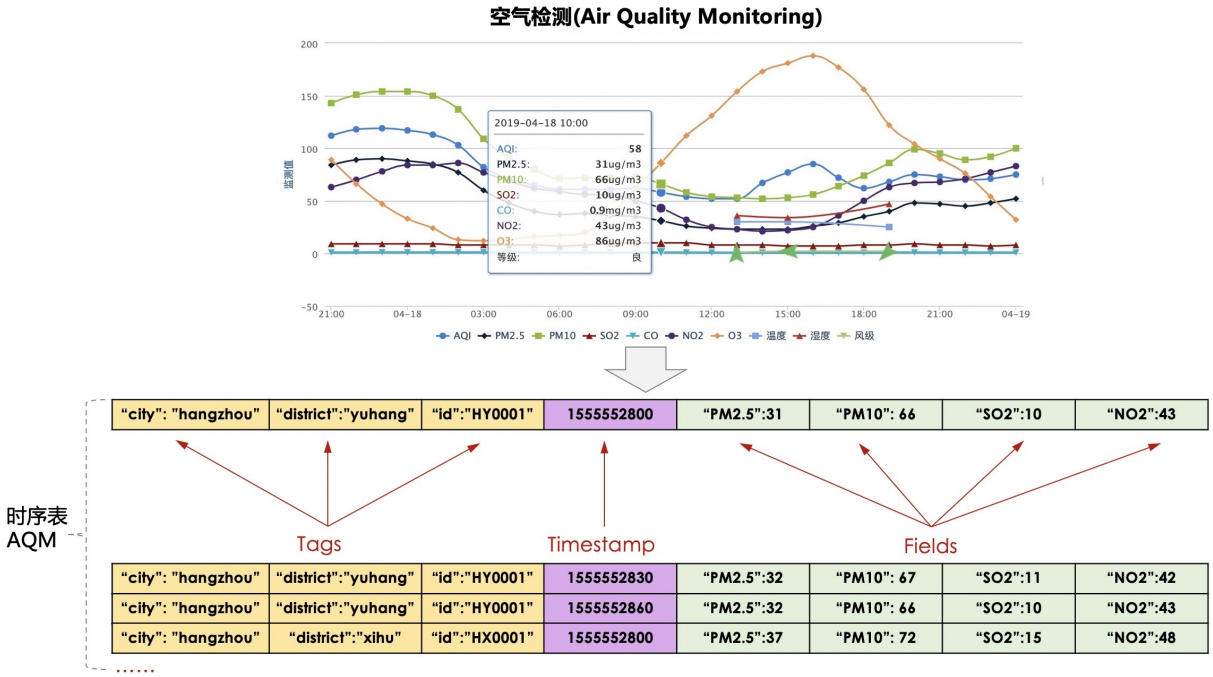
8.9. 最佳实践

8.9.1. 如何设计时序数据表

时序数据表可以为您提高查询和存储性能，本文介绍如何根据时序数据的来源和场景建立时序数据表。

时序数据示例

时序数据建模至关重要，需要仔细分析数据来源的特性和查询的场景，建立合理的数据表，以便达到最佳的存储和查询性能。以下图的空气检测数据为例：



每个时间点对应的数据记录可以详细分为几个部分：

- 表（Table）：代表一系列同类时序数据的集合。
- 标签（Tags）：表明指标项监测针对的具体对象属性。其中一个标签（Tag）由一个标签键（Key）和一个对应的标签值（Value）组成。时序引擎默认会为每一个标签的健-值对都建立标签与时间序列之间的索引。在特定场合下，标签也可能会有不同的叫法，例如可能被称为*Labels*或者*Dimensions*。
- 时间戳（Timestamp）：表示该数据记录对应的生成时间。
- 字段（Field）：一条数据记录可以有多个字段值，表示指标的不同方面。时序引擎不会为字段建立索引。

表设计的最佳实践

基于以上的数据模型，我们便可以将同一类时序数据的度量（如上图中的AQM）设计为表名，并按照SQL的CREATE TABLE语法进行表设计。
以上文的空气质量监测的时序模型为例，可以采用的表定义如下所示。

```
CREATE TABLE aqm (
    city        VARCHAR TAG,
    district    VARCHAR TAG,
    id          VARCHAR TAG,
    time        TIMESTAMP,
    pm2_5       DOUBLE,
    pm10        DOUBLE,
    so2         DOUBLE,
    no2         DOUBLE,
    PRIMARY KEY(id)
);
```

在此定义基础上，随后向表中写数据便可通过INSERT语句来进行：

```
INSERT INTO aqm (city, district, id, time, pm2_5, pm10, so2, no2)
VALUES ('hangzhou', 'yuhang', 'HY00001', '2019-04-18 10:00:00', 31.0, 66.0, 10.0, 43.0);

INSERT INTO aqm (city, district, id, time, pm2_5, pm10, so2, no2)
VALUES ('hangzhou', 'yuhang', 'HY00001', '2019-04-18 10:01:00', 31.2, 66.0, 10.5, 43.1);
```

也可以通过下述INSERT语句实现批量写入数据。

```
INSERT INTO aqm (city, district, id, time, pm2_5, pm10, so2, no2)
VALUES ('hangzhou', 'yuhang', 'HY00001', '2019-04-18 10:02:00', 31.3, 66.0, 10.0, 42.9),
('hangzhou', 'yuhang', 'HY00001', '2019-04-18 10:03:00', 31.2, 66.4, 10.3, 43.0);
```

> **说明**
> 关键字TAG是时序引擎对SQL的语法扩展，用于将表中的部分列标示为“标签”。为避免在时序数据建表的过程中纠结于应该将哪些字段应定义为TAG，建议尝试遵循以下原则：
>
> - 能够标示数据采集源属性的字段定义为TAG，特别是能够唯一标示数据源的属性。
> - 应避免使用进程ID，时间关联属性等易变值作为标签，即便这类属性的类型是字符串，仍然建议将其定义为Field。如果使用此类易变属性作为标签，将会导致时间线数量急剧膨胀，并带来时间线索引大小剧增，反而不利于查询。

PRIMARY KEY设计的最佳实践

数据库会根据PRIMARY KEY进行存储分片与查询优化，指定PRIMARY KEY的查询请求将会非常高效，强烈建议建表时指定。
通常建议选择数据源的唯一标识作为PRIMARY KEY，场景示例如下：

- 物联网、工业互联网场景，可以采用设备ID作为PRIMARY KEY。
- 车联网场景，可以采用车辆唯一标识来作为PRIMARY KEY。
- 监控场景，可以采用APP ID或者host:port等标识作为PRIMARY KEY。

以上述时序数据表aqm的示例，如果业务存在大量基于ID的查询，则指定ID作为PRIMARY KEY是最佳选择。

```
CREATE TABLE aqm (
    city        VARCHAR TAG,
    district    VARCHAR TAG,
    id          VARCHAR TAG,
    time        TIMESTAMP,
    pm2_5       DOUBLE,
    pm10        DOUBLE,
    so2         DOUBLE,
    no2         DOUBLE,
    PRIMARY KEY (id)
);
```

> 注意
> 和传统的关系型数据库不同，在时序数据表中定义为PRIMARY KEY的列并不要求列的值遵循唯一性约束，但是需要出现在PRIMARY KEY列表中的列名必须是标签列。

数据类型选择的最佳实践

时序引擎当前支持的数据类型可参见数据类型。对于时序数据表的创建，各种列的数据类型选择的建议如下：

- 标签(Tag)列
  声明为TAG的列固定为VARCHAR类型，无法更改。
- 时间戳(Timestamp)列
  在存储引擎时间戳实际上会按照BIGINT类型统一存为Epoch时间戳以便实现更优的时间戳压缩。但用户可结合对时间戳的计算需求自行选择在建表时使用BIGINT类型或TIMESTAMP类型。
- 字段(Field)列
  理论上字段列可选择使用所有数据类型。但出于数据压缩和查询计算的角度考虑，应尽量避免使用VARCHAR类型作为字段列的类型。

8.9.2. 如何高效地写入数据

本文介绍通过JDBC向时序引擎中写入数据的方法。

基于JDBC开发高效数据写入的应用

对于使用JDBC进行应用开发的场景，为了达到高效数据写入的目的，推荐使用JDBC中的 `PreparedStatement` 实现INSERT语句的批量写入。具体步骤如下：

1. 使用JDBC创建 `Connection` 对象。
   通过JDBC建立连接的方法，详细可参见通过JDBC Driver连接使用（推荐）。
2. 创建 `PreparedStatement` 对象并指定带参数的INSERT语句。
   以"如何为时序数据建表"章节中介绍的示例为基础，创建 `PreparedStatement` 的方法如下所示：

   ```
   StringBuilder builder = new StringBuilder();
   builder.append("INSERT INTO aqm (city, district, id, time, pm2_5, pm10, so2, no2) ");
   builder.append("VALUES (?, ?, ?, ?, ?, ?, ?, ?)");

   PreparedStatement preparedStmt = connection.prepareStatement(builder.toString());
   ```

3. 对 `PreparedStatement` 对象进行批量的输入参数绑定，并实际执行写入。
   对于上一步的INSERT语句，绑参的示例代码如下：

```
    for (int i = 0; i < batchSize; i++) {
        // 按INSERT语句中的列参数指定顺序依次绑参
        // 下述代码中的 city, district, id, ts, pm25等都是用于绑参的局部变量
        preparedStmt.setString(1, city);
        preparedStmt.setString(2, district);
        preparedStmt.setString(3, id);
        preparedStmt.setTimestamp(4, ts);
        preparedStmt.setDouble(5, pm25);
        preparedStmt.setDouble(6, pm10);
        preparedStmt.setDouble(7, so2);
        preparedStmt.setDouble(8, no2);

        // 加入批次
        preparedStmt.addBatch();
    }

    // 执行一个批次的写入
    int[] results = preparedStmt.executeBatch();

    // 从results中查询实际写入的数据记录数。以下略
}
```

说明

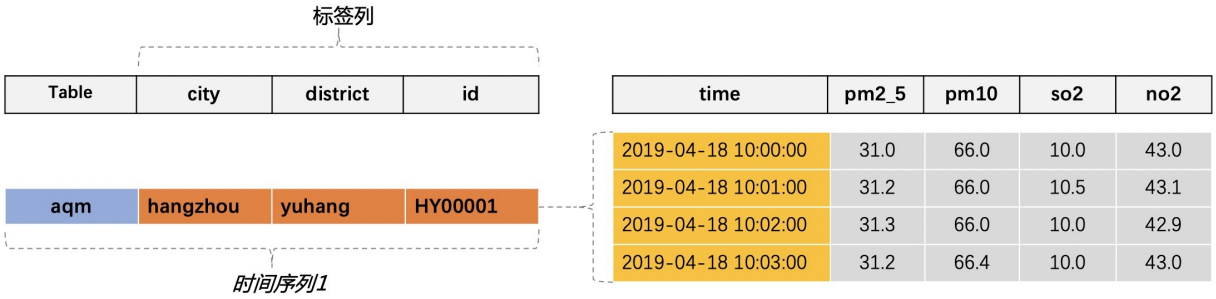
- 时序引擎的JDBC中的 `PreparedStatement` 只支持无命名参数（即通过占位符“?”指定参数），因此在实际调用 `PreparedStatement` 的绑参API时，需要确保 `setXXX()` 方法中指定的参数index所对应的参数值与INSERT语句中占位符所对应列一致。
- `PreparedStatement` 在执行 `executeBatch()` 前绑定的参数批次并不是越多越好，其性能表现往往与表定义相关。根据实际测试的结果，如果表定义中仅有1个Field，并且TAG列的个数不超过5个时，`PreparedStatement` 执行的一个批次的个数为5000（即上述代码示例中的 `batchSize` 变量大小）性能最优。

8.9.3. 如何高效地查询时序数据

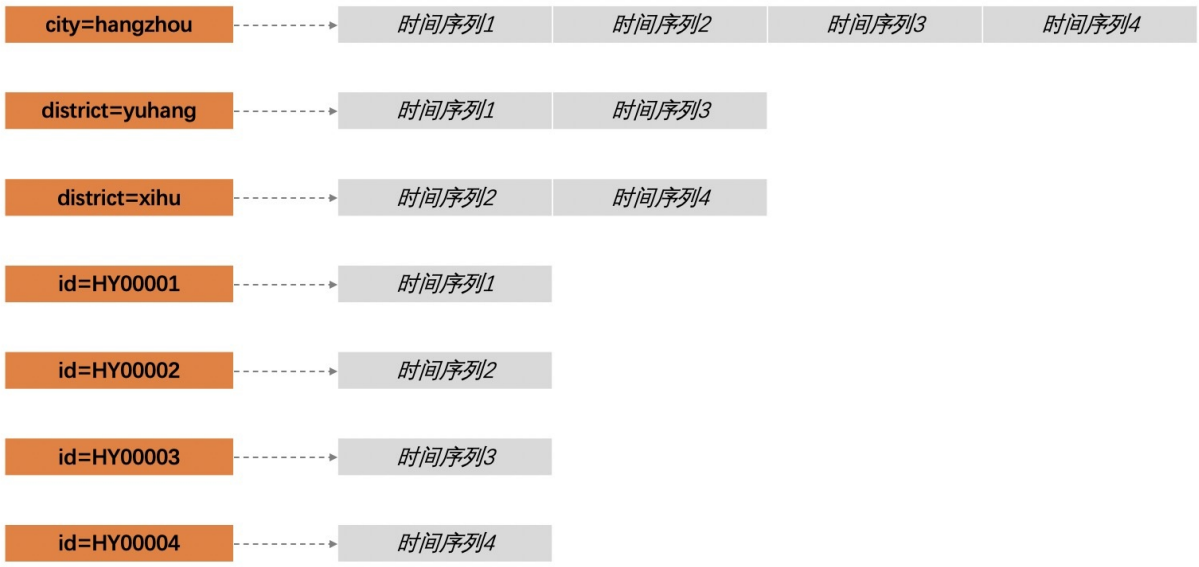
本文介绍了时序表的索引机制，以及查询时序表的最佳实践。

背景信息

时序数据的模型请参见如何设计时序数据表，数据是按照时间序列来进行组织的。所以在时序数据表中，所有标记TAG的列（标签列）会被建立为索引列，来表示每一条数据具体所属于哪个时间序列，从而显示数据的来源。
以时序数据表为例，具体请参见如何设计时序数据表，这些数据都属于同一个时间序列，如下图所示。



- 如果标签列的值发生变化，则数据将属于不同的时间序列。通常情况下，一个时间序列可以标记一个产生时序数据的数据源。
- 时序引擎会根据每个**标签列**建立索引，其索引形态是倒排索引，以标签列的列名和值作为索引键来索引所有拥有该键值对的时间序列，用于从时间序列维度快速定位某个标签对应的数据范围。随着数据的不断写入，时序数据表的倒排索引将会呈现以下形态。



- 对于时间戳列，时序引擎会默认建立数据块范围索引，确保在海量时序数据中快速定位到需要查询的数据范围。但对于普通的Field列，时序引擎不会对数据建立索引。

使用建议

结合时序引擎的内部索引机制，对Lindorm时序引擎的时序表进行查询时需要考虑以下几点：

- 建议在查询的条件语句中增加标签列的等值过滤条件语句和时间戳列的过滤条件语句，避免条件语句中只有Field列过滤语句。
- 查询的条件语句中如果包含多个标签列过滤条件，且这几个标签列映射的时间线范围存在包含关系，建议只需要保留选择区分度最高的标签列的过滤条件（例如上图中的id列相对city列选择区分度更高）。
- 查询条件覆盖的时间范围过大可能会导致扫描过多的数据，进而导致查询速度过慢，在这种情况下建议您可以缩小时间戳列的过滤范围来提高查询性能。

常见的时序查询场景和查询操作

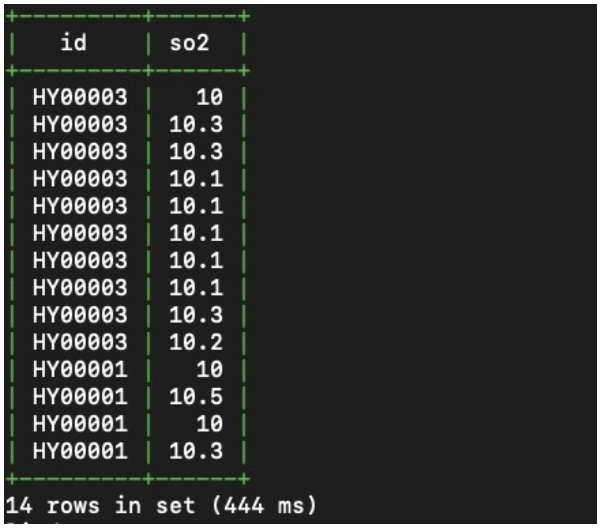
> 注意
> 如果需要复现以下查询场景的结果，请下载相应的SQL脚本语句填写样例数据。

查询时间范围内的原始点

使用以下语句查询余杭区内设备从2019-04-18 10:00:00至2019-04-18 10:30:00上报的全部SO2监控指标。

```
SELECT id, so2 FROM aqm WHERE district='yuhang' AND time >= '2019-04-18 10:00:00' AND time < '2019-04-18 10:30:00';
```

查询结果如下。



id	so2
HY00003	10
HY00003	10.3
HY00003	10.3
HY00003	10.1
HY00003	10.1
HY00003	10.1
HY00003	10.1
HY00003	10.1
HY00003	10.3
HY00003	10.2
HY00001	10
HY00001	10.5
HY00001	10
HY00001	10.3

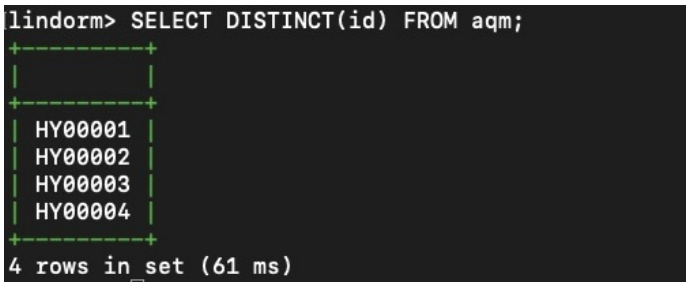
14 rows in set (444 ms)

查询时序数据表中特定标签列的所有标签值

当将许多同类设备上报的数据接入到一张时序数据表中之后，有时会需要查询接入设备的某个标签的标签值。例如下述SQL就可以查询时序数据表 `aqm` 中接入的所有空气质量监控设备的设备 `id` ：

```
SELECT DISTINCT(id) FROM aqm;
```

查询结果如下。



```
lindorm> SELECT DISTINCT(id) FROM aqm;
+---------+
|         |
+---------+
| HY00001 |
| HY00002 |
| HY00003 |
| HY00004 |
+---------+
4 rows in set (61 ms)
```

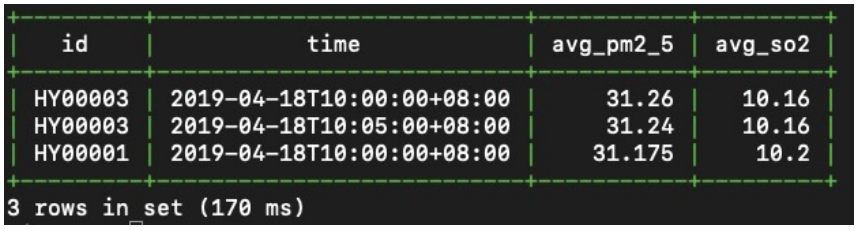
> **说明**
> 对于标签列的查询在存储引擎层面默认会基于标签索引进行查询，因此无需担心这样的查询会走全表扫描而产生性能问题。但若已接入的设备量非常多时，即使是走标签索引其查询耗时也可能会较长。

查询时间精度的降采样

查询余杭区内设备从2019-04-18 10:00:00至2019-04-18 10:30:00上报的PM2.5和SO2监控指标，按5分钟为粒度求平均值。

```
SELECT id, time, avg(pm2_5) AS avg_pm2_5, avg(so2) AS avg_so2 FROM aqm WHERE district='yuhang' AND time >= '
2019-04-18 10:00:00' AND time < '2019-04-18 10:30:00' SAMPLE BY 5m;
```

查询结果如下。



id	time	avg_pm2_5	avg_so2
HY00003	2019-04-18T10:00:00+08:00	31.26	10.16
HY00003	2019-04-18T10:05:00+08:00	31.24	10.16
HY00001	2019-04-18T10:00:00+08:00	31.175	10.2

3 rows in set (170 ms)

> **说明**
> 降采样的实质是以时间序列为单位在时间维度的聚合。关于降采样查询的详细说明请参见 降采样。

查询跨设备的聚合数据

查询余杭区内设备从2019-04-18 10:00:00至2019-04-18 10:30:00上报的PM2.5和SO2监控指标，查询按照5分钟粒度的最大平均值。

```
SELECT max(avg_pm2_5) AS max_avg_pm25, max(avg_so2) AS max_avg_so2 FROM (SELECT district, id, time, avg(pm2_5) AS avg_pm2_5, avg(so2) AS avg_so2 FROM aqm WHERE district='yuhang' AND time >= '2019-04-18 10:00:00' AND time < '2019-04-18 10:30:00' SAMPLE BY 5m) GROUP BY district;
```

查询结果如下。

```
+--------------+--------------+
| max_avg_pm25 | max_avg_so2 |
+--------------+--------------+
|        31.26 |         10.2 |
+--------------+--------------+
1 rows in set (103 ms)
```

8.10. 性能白皮书

8.10.1. 测试环境与工具

本文主要介绍本次测试的测试环境和工具使用方式。

测试环境要求

- 测试使用的操作系统为Alibaba Cloud Linux 3.2104 64位。
- 压测端和Lindorm时序引擎在同一个VPC网络中，使用VPC地址进行连接。
- 本测试中压测端使用的机器规格为ecs.c6e.26xlarge。建议使用32C128G以上配置的机器。

Lindorm时序引擎规格

分别测试了4组Lindorm时序引擎的规格，存储类型均为标准型云存储。

- 集群1：4核16GB * 3节点
- 集群2：8核32GB *3节点
- 集群3：16核64GB *3节点
- 集群4：32核128GB *3节点

> 说明
> - 上述测试实例规格均使用Lindorm产品的公开售卖规格。
> - 测试实例的存储类型选用的是标准型云存储，选用性能型云存储可以获得更好的性能。
> - 测试实例的时序引擎版本为3.4.9。

测试工具介绍

测试工具基于时序数据库领域应用最广的压测工具influxdb-comparisons，在此基础上进行了Lindorm的适配。
该测试中数据模型采用压测工具中的DevOps模型。这个数据模型的场景是：一组服务器需要定期报告系统和应用程序的指标，比如每10秒在一台服务器上的9个子系统（CPU、内存、磁盘、磁盘I/O、内核、网络、Redis、PostgreSQL和Nginx）上采样101个测量值，上报到时序数据库。

测试工具下载

通过下载压测工具，上传到压测客户端机器并解压后即可进行测试。

> 说明
> 关于该工具的使用方法介绍可参见influxdb-comparisons，整体的使用方式和思路和它类似。

写入测试

- 生成写入数据

```
# ${scale} ：生成数据的设备数量
# ${data_file} ：生成的数据文件名
# ${batch_size} ：batch的大小
GOGC=500 ./bulk_data_gen -format lindormtsdbbatch -use-case devops -seed 100 -sampling-interval 10s -scale-v
ar ${scale} -batchSize ${batchsize}  > ${data_file}
```

- 进行写入

```
# ${data_file} ：上一步生成的文件名
# ${add} ：Lindorm时序引擎的地址，形式为ip:port， 例如 ld-8vb458k154q7xxxx-proxy-tsdb.lindorm.rds.aliyuncs.com:8
242
# ${workers} ：写入并发大小
GOGC=500 ./bulk_load_lindormtsdb_batch -file ${data_file} -url ${add} -workers ${workers} -time-limit 10m
```

查询测试

- 生成查询

```
# ${query_type} ：查询的类型，具体参考
查询测试结果

# ${scale} ：scale的大小
# ${queryfile} ：生成的包含查询的文件
GOGC=500 ./bulk_query_gen -seed 123 -queries 100000 -format lindormtsdb -use-case devops -query-type ${query
_type} -scale-var ${scale} > ${queryfile}
```

- 进行查询

```
# ${workers} ：查询的并发数
# ${add} ：Lindorm时序引擎的地址，形式为ip:port， 例如 ld-8vb458k154q7xxxx-proxy-tsdb.lindorm.rds.aliyuncs.com:8
242
# ${queryfile} ：上一步生成的包含查询语句的文件
GOGC=500 ./query_benchmarker_lindormtsdb -workers ${workers} -url ${add} -file ${queryfile}
```

8.10.2. 写入测试结果

本章节介绍写入测试的结果对比。本测试是在发挥数据最佳性能，并在没有任何查询的情况下测得的结果。在不同时间线规模的业务场景，或同时存在查询的场景下，写入性能可能会下降，建议在使用前结合自己的业务场景进行测试评估，本测试结果仅供参考。

写入数据样例

每10秒在一台服务器的9个子系统（CPU、内存、磁盘、磁盘I/O、内核、网络、Redis、PostgreSQL和Nginx）上采样101个测量点，上报到Lindorm时序引擎。
每个子系统在Lindorm时序引擎中作为一张Table，Table、TAG、Field的定义参考名词解释。

- 建表语句如下，其中hostname作为每个设备的唯一标识。

```
CREATE table cpu(hostname VARCHAR primary TAG,region VARCHAR TAG,datacenter VARCHAR TAG,rack VARCHAR TAG,os
VARCHAR TAG,arch VARCHAR TAG,team VARCHAR TAG,service VARCHAR TAG,service_version VARCHAR TAG,service_enviro
nment VARCHAR TAG,time BIGINT,usage_user double,usage_system double,usage_idle double,usage_nice double,usag
e_iowait double,usage_irq double,usage_softirq double,usage_steal double,usage_guest double,usage_guest_nice
double);

CREATE table mem(hostname VARCHAR primary TAG, region VARCHAR TAG, datacenter VARCHAR TAG, rack VARCHAR TAG,
os VARCHAR TAG, arch VARCHAR TAG, team VARCHAR TAG, service VARCHAR TAG, service_version VARCHAR TAG, servic
e_environment VARCHAR TAG, time BIGINT,total BIGINT, available BIGINT, used BIGINT, `free` BIGINT, cached BI
GINT, buffered BIGINT, used_percent double, available_percent double, buffered_percent double);

CREATE table disk(hostname VARCHAR primary TAG, region VARCHAR TAG, datacenter VARCHAR TAG, rack VARCHAR TAG
, os VARCHAR TAG, arch VARCHAR TAG, team VARCHAR TAG, service VARCHAR TAG, service_version VARCHAR TAG, serv
ice_environment VARCHAR TAG, path VARCHAR TAG, fstype VARCHAR TAG,time BIGINT,total BIGINT, `free` BIGINT, u
sed BIGINT, used_percent BIGINT, inodes_total BIGINT, inodes_free BIGINT, inodes_used BIGINT);

CREATE table diskio(hostname VARCHAR primary TAG, region VARCHAR TAG, datacenter VARCHAR TAG, rack VARCHAR T
AG, os VARCHAR TAG, arch VARCHAR TAG, team VARCHAR TAG, service VARCHAR TAG, service_version VARCHAR TAG, se
rvice_environment VARCHAR TAG, serial VARCHAR TAG,time BIGINT,`reads` BIGINT, writes BIGINT, read_bytes BIGI
NT, write_bytes BIGINT, read_time BIGINT, write_time BIGINT, io_time BIGINT);

CREATE table kernel(hostname VARCHAR primary TAG, region VARCHAR TAG, datacenter VARCHAR TAG, rack VARCHAR T
AG, os VARCHAR TAG, arch VARCHAR TAG, team VARCHAR TAG, service VARCHAR TAG, service_version VARCHAR TAG, se
rvice_environment VARCHAR TAG,time BIGINT,boot_time BIGINT, interrupts BIGINT, context_switches BIGINT, proc
esses_forked BIGINT, disk_pages_in BIGINT, disk_pages_out BIGINT);

CREATE table net(hostname VARCHAR primary TAG, region VARCHAR TAG, datacenter VARCHAR TAG, rack VARCHAR TAG,
os VARCHAR TAG, arch VARCHAR TAG, team VARCHAR TAG, service VARCHAR TAG, service_version VARCHAR TAG, servic
e_environment VARCHAR TAG, interface VARCHAR TAG,time BIGINT, bytes_sent BIGINT, bytes_recv BIGINT, packets_
sent BIGINT, packets_recv BIGINT, err_in BIGINT, err_out BIGINT, drop_in BIGINT, drop_out BIGINT );

CREATE table redis(hostname VARCHAR primary TAG, region VARCHAR TAG, datacenter VARCHAR TAG, rack VARCHAR TA
G, os VARCHAR TAG, arch VARCHAR TAG, team VARCHAR TAG, service VARCHAR TAG, service_version VARCHAR TAG, ser
vice_environment VARCHAR TAG, port VARCHAR TAG, server VARCHAR TAG,time BIGINT, uptime_in_seconds BIGINT, to
tal_connections_received BIGINT, expired_keys BIGINT, evicted_keys BIGINT, keyspace_hits BIGINT, keyspace_mi
sses BIGINT, instantaneous_ops_per_sec BIGINT, instantaneous_input_kbps BIGINT, instantaneous_output_kbps BI
GINT, connected_clients BIGINT, used_memory BIGINT, used_memory_rss BIGINT, used_memory_peak BIGINT, used_me
mory_lua BIGINT, rdb_changes_since_last_save BIGINT, sync_full BIGINT, sync_partial_ok BIGINT, sync_partial_
err BIGINT, pubsub_channels BIGINT, pubsub_patterns BIGINT, latest_fork_usec BIGINT, connected_slaves BIGINT
, master_repl_offset BIGINT, repl_backlog_active BIGINT, repl_backlog_size BIGINT, repl_backlog_histlen BIGI
NT, mem_fragmentation_ratio BIGINT, used_cpu_sys BIGINT, used_cpu_user BIGINT, used_cpu_sys_children BIGINT,
used_cpu_user_children BIGINT);

CREATE table postgresl(hostname VARCHAR primary TAG, region VARCHAR TAG, datacenter VARCHAR TAG, rack VARCHA
R TAG, os VARCHAR TAG, arch VARCHAR TAG, team VARCHAR TAG, service VARCHAR TAG, service_version VARCHAR TAG,
service_environment VARCHAR TAG, time BIGINT,numbackends BIGINT, xact_commit BIGINT, xact_rollback BIGINT, b
lks_read BIGINT, blks_hit BIGINT, tup_returned BIGINT, tup_fetched BIGINT, tup_inserted BIGINT, tup_updated
BIGINT, tup_deleted BIGINT, conflicts BIGINT, temp_files BIGINT, temp_bytes BIGINT, deadlocks BIGINT, blk_re
ad_time BIGINT, blk_write_time BIGINT );

CREATE table nginx(hostname VARCHAR primary TAG, region VARCHAR TAG, datacenter VARCHAR TAG, rack VARCHAR TA
G, os VARCHAR TAG, arch VARCHAR TAG, team VARCHAR TAG, service VARCHAR TAG, service_version VARCHAR TAG, ser
vice_environment VARCHAR TAG, port VARCHAR TAG, server VARCHAR TAG,time BIGINT,accepts BIGINT, active BIGINT
, handled BIGINT, reading BIGINT, requests BIGINT, waiting BIGINT, writing BIGINT );
```

- 向每张表插入数据的样例如下

```
INSERT INTO cpu(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environment, ti
me ,usage_user,usage_system,usage_idle,usage_nice,usage_iowait,usage_irq,usage_softirq,usage_steal,usage_gue
st,usage_guest_nice) VALUES ('host_0','ap-northeast-1','ap-northeast-1a','72','Ubuntu16.10','x86','CHI','10'
,'0','test',1514764800000,60.4660287979619540,94.0509088045012476,66.4560053218490481,43.7714187186980155,42
.4637497071265670,68.6823072867109374,6.5637019217476222,15.6519254732791246,9.6969518914484567,30.091186058
5287059);
INSERT INTO diskio(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environment,
serial, time ,`reads`,writes,read_bytes,write_bytes,read_time,write_time,io_time) VALUES ('host_0','ap-north
east-1','ap-northeast-1a','72','Ubuntu16.10','x86','CHI','10','0','test','694-511-162',1514764800000,0,0,3,0
,0,7,0);
INSERT INTO disk(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environment,pa
th,fstype, time ,total,`free`,used,used_percent,inodes_total,inodes_free,inodes_used) VALUES ('host_0','ap-n
ortheast-1','ap-northeast-1a','72','Ubuntu16.10','x86','CHI','10','0','test','/dev/sda9','ext4',151476480000
0,1099511627776,549755813888,549755813888,50,268435456,134217728,134217728);
INSERT INTO kernel(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environment,
time ,boot_time,interrupts,context_switches,processes_forked,disk_pages_in,disk_pages_out) VALUES ('host_0',
'ap-northeast-1','ap-northeast-1a','72','Ubuntu16.10','x86','CHI','10','0','test',1514764800000,233,0,1,0,0,
0);
INSERT INTO mem(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environment, ti
me ,total,available,used,`free`,cached,buffered,used_percent,available_percent,buffered_percent) VALUES ('ho
st_0','ap-northeast-1','ap-northeast-1a','72','Ubuntu16.10','x86','CHI','10','0','test',1514764800000,858993
4592,6072208808,2517725783,5833292948,1877356426,2517725783,29.3101857336815748,70.6898142663184217,78.14469
47407235797);
INSERT INTO net(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environment,int
erface, time ,bytes_sent,bytes_recv,packets_sent,packets_recv,err_in,err_out,drop_in,drop_out) VALUES ('host
_0','ap-northeast-1','ap-northeast-1a','72','Ubuntu16.10','x86','CHI','10','0','test','eth3',1514764800000,0
,0,0,2,0,0,0,0);
INSERT INTO nginx(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environment,p
ort,server, time ,accepts,active,handled,reading,requests,waiting,writing) VALUES ('host_0','ap-northeast-1'
,'ap-northeast-1a','72','Ubuntu16.10','x86','CHI','10','0','test','12552','nginx_65466',1514764800000,0,0,11
,0,0,0,0);
INSERT INTO postgresql(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environme
nt, time ,numbackends,xact_commit,xact_rollback,blks_read,blks_hit,tup_returned,tup_fetched,tup_inserted,tup
_updated,tup_deleted,conflicts,temp_files,temp_bytes,deadlocks,blk_read_time,blk_write_time) VALUES ('host_0
','ap-northeast-1','ap-northeast-1a','72','Ubuntu16.10','x86','CHI','10','0','test',1514764800000,0,0,0,0,3,
0,0,0,0,0,0,0,12,0,0,0);
INSERT INTO redis(hostname,region,datacenter,rack,os,arch,team,service,service_version,service_environment,p
ort,server, time ,uptime_in_seconds,total_connections_received,expired_keys,evicted_keys,keyspace_hits,keysp
ace_misses,instantaneous_ops_per_sec,instantaneous_input_kbps,instantaneous_output_kbps,connected_clients,us
ed_memory,used_memory_rss,used_memory_peak,used_memory_lua,rdb_changes_since_last_save,sync_full,sync_partia
l_ok,sync_partial_err,pubsub_channels,pubsub_patterns,latest_fork_usec,connected_slaves,master_repl_offset,r
epl_backlog_active,repl_backlog_size,repl_backlog_histlen,mem_fragmentation_ratio,used_cpu_sys,used_cpu_user
,used_cpu_sys_children,used_cpu_user_children) VALUES ('host_0','ap-northeast-1','ap-northeast-1a','72','Ubu
ntu16.10','x86','CHI','10','0','test','19071','redis_86258',1514764800000,0,0,0,5,0,0,0,0,0,0,8589934592,858
9934592,8589934592,8589934592,0,0,0,0,36,0,0,0,0,0,0,0,0,0,16,0,0);
```

- 本测试中，使用SQL语句进行写入，为了使SQL写入具有更高的性能，会对同一张表进行batch，先prepare然后再绑定参数，进行批量写入。写入方式参考 如何高效地写入数据

写入数据量

- 设备数量10000
- 每个设备每10秒上报101个测量值
- 共上报15天数据

测试性能指标说明

介绍该测试中写入的性能指标。

- tps： 向数据库中执行写入时，平均每秒钟写入的测量点数。在Lindorm时序引擎中，也就是每秒写入的Field数量，参考名词解释
- worker：写入的并发数量
- batch：每次写入的batch的行数

- max_cpu: CPU的峰值百分比

测试结果

> 注意
> - 写入方式均是攒批写入（即一次写入请求传入一批数据点的方式）。
> - 测试写入的同时，时序引擎没有执行任何查询。

图 1. batch为500的情况下，各个规格不同并发数量下的写入TPS

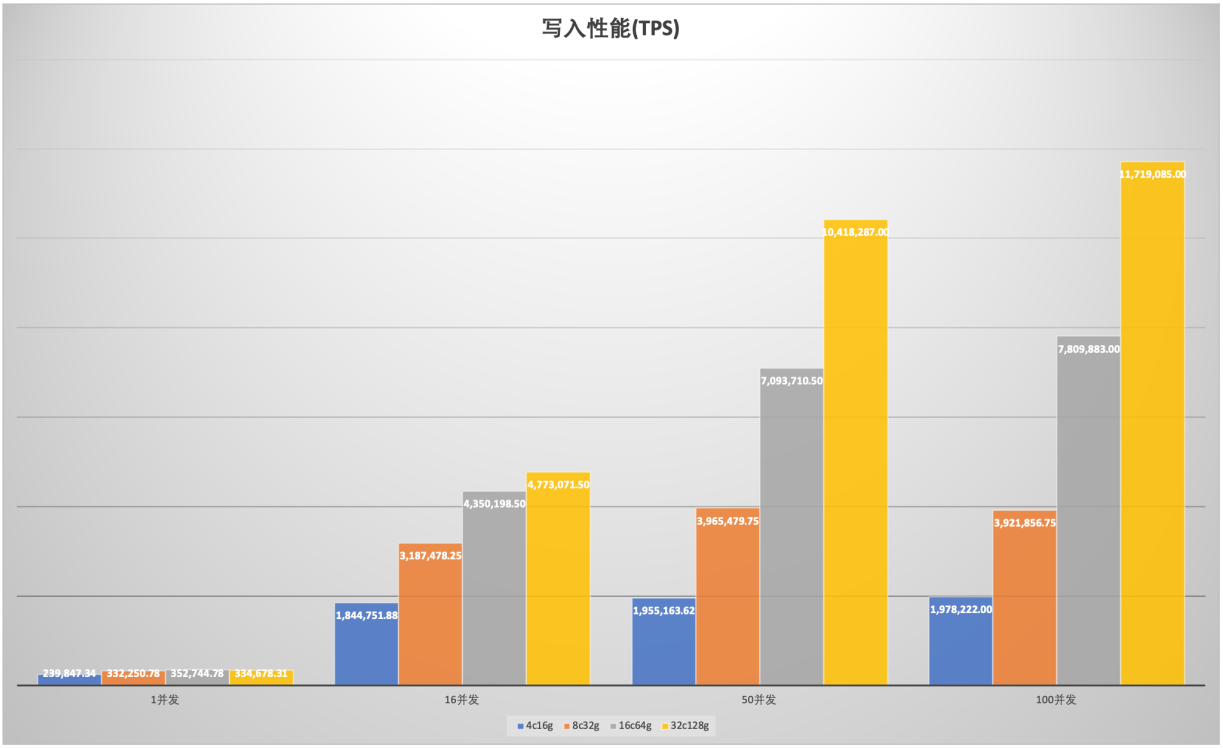


表 1. 集群1（4核16GB * 3节点）

batch	worker	tps	max_cpu
1	1	4,846.48	36.68
1	16	36,862.15	78.50
1	50	31,653.44	73.46
1	100	31,521.71	74.32
1	200	31,651.03	73.03
50	1	126,462.85	65.43
50	16	460,032.75	79.89
50	50	457,791.78	81.50
50	100	457,956.53	82.69
50	200	434,573.47	81.18
100	1	168,643.80	74.14

batch	worker	tps	max_cpu
100	16	468,008.25	84.09
100	50	470,608.31	84.34
100	100	451,384.44	83.32
100	200	457,740.22	84.61
200	1	205,046.31	74.77
200	16	480,309.56	84.74
200	50	489,903.34	86.73
200	100	484,745.44	86.77
200	200	475,824.97	86.55
500	1	239,847.34	74.76
500	16	511,989.50	87.86
500	50	544,544.75	88.23
500	100	543,131.56	88.12
500	200	528,027.12	88.57

表 2. 集群2（8核32GB * 3节点）

batch	worker	tps	max_cpu
1	1	3,601.72	19.88
1	16	46,701.05	46.97
1	50	69,892.66	59.77
1	100	70,219.33	60.32
1	200	70,187.81	60.54
50	1	114,062.01	22.88
50	16	1,123,739.88	64.66
50	50	1,416,314.00	70.99
50	100	1,421,701.75	70.94
50	200	1,422,040.12	71.56
100	1	183,456.98	22.38
100	16	1,651,046.25	65.09
100	50	2,029,514.75	74.16
100	100	2,040,670.38	73.39
100	200	2,025,066.12	73.98

batch	worker	tps	max_cpu
200	1	254,914.23	24.27
200	16	2,172,662.25	71.47
200	50	2,670,999.25	76.47
200	100	2,674,582.25	76.95
200	200	2,693,531.50	76.41
500	1	332,250.78	23.86
500	16	2,820,651.50	72.56
500	50	3,429,375.00	80.98
500	100	3,442,593.75	80.62
500	200	3,440,201.50	81.12

表 3. 集群3（16核64GB * 3节点）

batch	worker	tps	max_cpu
1	1	3,897.79	8.97
1	16	58,217.44	27.30
1	50	127,110.89	50.49
1	100	165,754.09	62.31
1	200	202,844.20	70.72
50	1	136,378.39	11.77
50	16	1,634,203.88	41.92
50	50	2,773,785.75	58.96
50	100	3,363,458.25	67.87
50	200	3,703,033.00	74.14
100	1	198,375.67	10.35
100	16	2,494,268.00	45.86
100	50	4,007,320.25	60.87
100	100	4,753,680.50	69.29
100	200	5,095,771.00	75.17
200	1	278,253.53	10.57
200	16	3,368,596.50	45.48
200	50	5,214,060.50	64.57
200	100	6,040,166.50	72.35

batch	worker	tps	max_cpu
200	200	6,283,312.00	77.07
500	1	352,744.78	10.80
500	16	4,281,761.50	47.17
500	50	6,544,214.00	71.73
500	100	7,267,295.50	77.15
500	200	7,290,116.00	83.91

表 4. 集群4（32核128GB * 3节点）

batch	worker	tps	max_cpu
1	1	3,405.32	4.52
1	16	51,460.87	11.04
1	50	134,289.62	27.32
1	100	201,014.75	40.45
1	200	255,692.84	51.99
50	1	113,644.64	5.33
50	16	1,596,669.88	19.13
50	50	3,676,491.50	38.75
50	100	5,217,282.50	50.84
50	200	6,345,112.00	62.49
100	1	188,352.08	5.05
100	16	2,624,622.50	21.15
100	50	5,740,561.50	40.49
100	100	7,521,672.00	55.85
100	200	8,507,855.00	61.68
200	1	249,571.77	5.05
200	16	3,637,803.50	21.23
200	50	8,141,380.50	45.39
200	100	10,289,145.00	57.85
200	200	10,462,525.00	60.48
500	1	334,678.31	5.47
500	16	4,657,772.50	24.23
500	50	10,098,200.00	46.90

batch	worker	tps	max_cpu
500	100	12,405,648.00	64.57
500	200	12,136,903.00	66.01

8.10.3. 查询测试结果

本章节介绍查询测试的结果对比。本测试是在发挥数据最佳性能，并在没有任何数据写入的情况下测得的结果。在不同时间线规模的业务场景，或同时存在大量写入的场景下，或查询扫描的时间线和时间点更多时，查询性能可能会下降，建议在使用前结合自己的业务场景进行测试评估，本测试结果仅供参考。

查询的基础数据量

- 设备数100
- 每个设备每10秒上报101个测量值
- 共上报1天的数据

查询样例

分别测试多种查询，queryType表示查询类型，分别介绍如下

- 1-host-1-hr-raw: 查询1个小时数据，1个设备，查找所有的cpu.usage_user的值

```
select time, usage_user from cpu where (hostname = 'host_17') and time >=1514837782646 and time < 1514841382
646 ;
```

- 1-host-1-hr: 查询1个小时数据，1个设备，查找每分钟cpu.usage_user的最大值

```
select time, max(usage_user) as max_usage_user from cpu where (hostname = 'host_17') and time >=151483778264
6 and time < 1514841382646 sample by 1m
```

- 1-host-12-hr-raw: 查询12个小时数据，1个设备，查找所有的cpu.usage_user的值

```
select time, usage_user from cpu where (hostname = 'host_0') and time >=1514787382646 and time < 15148305826
46;
```

- 1-host-12-hr: 查询12小时数据，1个设备，查找每分钟cpu.usage_user的最大值

```
select time, max(usage_user) as max_usage_user from cpu where (hostname = 'host_0') and time >=1514787382646
and time < 1514830582646 sample by 1m ;
```

- 1-host-latest: 随机查找1个设备的cpu.usage_user的最新值

```
select  time, latest(usage_user) as latest_usage_user from cpu where (hostname = 'host_50') sample by 0
```

- 8-host-latest: 随机查找8个设备的cpu.usage_user的最新值

```
select  time, latest(usage_user) as latest_usage_user from cpu where (hostname = 'host_64' or hostname = 'ho
st_80' or hostname = 'host_21' or hostname = 'host_5' or hostname = 'host_79' or hostname = 'host_47' or hos
tname = 'host_25' or hostname = 'host_18') sample by 0
```

测试性能指标说明

介绍该测试中查询的性能指标。

- queryType： 参考上一节查询样例中的介绍
- worker: 查询的并发数量
- qps：每秒执行完成的查询SQL数量
- max_cpu: CPU的峰值百分比
- max_mem: 内存的峰值百分比

测试结果

> 注意
> 下述查询的性能测试结果均是在时序引擎无其他工作负载的情况下单独测得。

图 1. 并发为200的情况下，各个规格的3种查询下的QPS

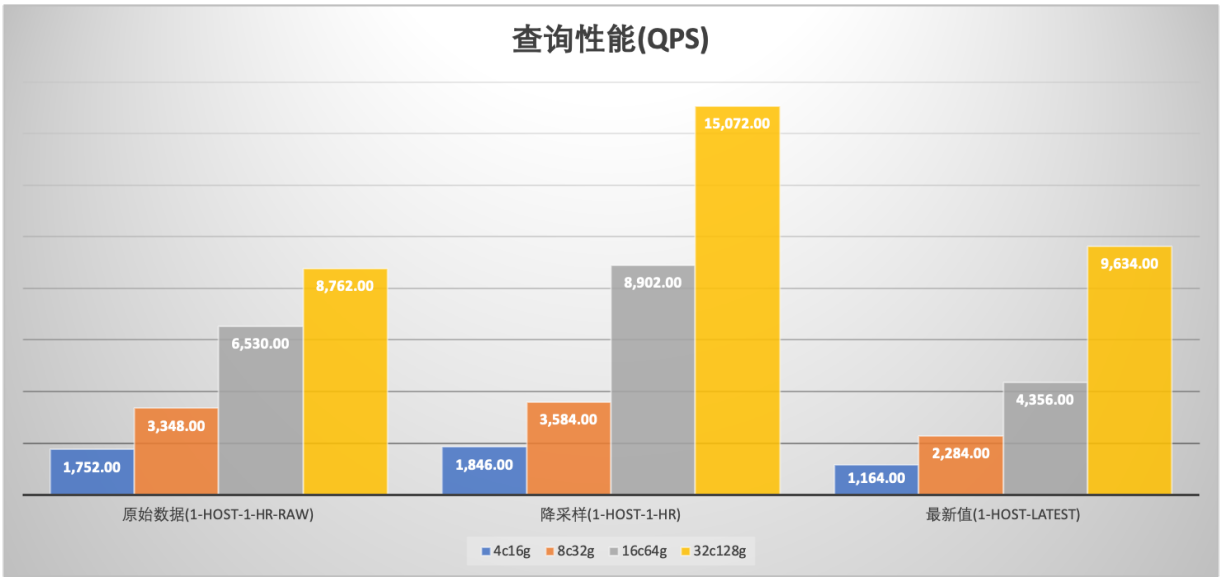


表 1. 集群1（4核16GB * 3节点）

queryType	worker	qps	max_cpu	max_mem
1-host-1-hr	16	1,420.96	86.95	48.08
1-host-1-hr	50	1,865.50	83.45	49.32
1-host-1-hr	100	1,887.00	81.71	48.95
1-host-1-hr	200	1,846.00	82.99	48.87
1-host-1-hr-raw	16	1,324.80	79.54	50.34
1-host-1-hr-raw	50	1,722.00	86.42	50.96
1-host-1-hr-raw	100	1,737.00	86.36	50.92
1-host-1-hr-raw	200	1,752.00	87.09	50.88
1-host-12-hr	16	497.44	86.77	48.81
1-host-12-hr	50	555.00	95.05	48.81
1-host-12-hr	100	571.00	94.97	48.90
1-host-12-hr	200	584.00	94.77	49.11
1-host-12-hr-raw	16	286.56	82.20	51.04
1-host-12-hr-raw	50	328.00	95.34	51.14
1-host-12-hr-raw	100	330.00	96.51	51.21
1-host-12-hr-raw	200	334.00	96.70	51.56

queryType	worker	qps	max_cpu	max_mem
1-host-latest	16	1,025.12	81.79	49.34
1-host-latest	50	1,146.00	93.36	49.42
1-host-latest	100	1,144.00	92.41	49.36
1-host-latest	200	1,164.00	92.89	49.44
8-host-latest	16	175.20	96.65	49.64
8-host-latest	50	180.00	97.01	49.95
8-host-latest	100	181.00	96.59	50.05
8-host-latest	200	182.00	96.73	50.15

表 2. 集群2（8核32GB * 3节点）

queryType	worker	qps	max_cpu	max_mem
1-host-1-hr	16	2,426.56	79.45	31.83
1-host-1-hr	50	3,363.50	71.58	33.50
1-host-1-hr	100	3,530.00	73.80	35.44
1-host-1-hr	200	3,584.00	70.12	35.43
1-host-1-hr-raw	16	2,181.92	52.67	36.10
1-host-1-hr-raw	50	3,231.50	69.77	36.20
1-host-1-hr-raw	100	3,371.00	72.88	36.19
1-host-1-hr-raw	200	3,348.00	69.80	35.08
1-host-12-hr	16	877.44	59.57	35.66
1-host-12-hr	50	1,132.50	82.37	35.51
1-host-12-hr	100	1,145.00	84.10	35.55
1-host-12-hr	200	1,152.00	84.69	35.62
1-host-12-hr-raw	16	445.44	58.55	35.29
1-host-12-hr-raw	50	641.00	84.98	35.31
1-host-12-hr-raw	100	668.00	90.79	35.35
1-host-12-hr-raw	200	670.00	91.98	35.42
1-host-latest	16	1,785.44	51.20	35.80
1-host-latest	50	2,273.50	64.86	35.92
1-host-latest	100	2,280.00	64.97	35.82
1-host-latest	200	2,284.00	65.01	35.85
8-host-latest	16	439.04	85.04	35.98

queryType	worker	qps	max_cpu	max_mem
8-host-latest	50	471.50	91.71	36.00
8-host-latest	100	472.00	92.42	36.08
8-host-latest	200	470.00	93.10	36.09

表 3. 集群3（16核64GB * 3节点）

queryType	worker	qps	max_cpu	max_mem
1-host-1-hr	16	3,287.84	49.77	25.83
1-host-1-hr	50	6,543.50	56.06	26.45
1-host-1-hr	100	7,713.00	59.81	26.48
1-host-1-hr	200	8,902.00	75.23	26.47
1-host-1-hr-raw	16	2,857.60	32.42	26.89
1-host-1-hr-raw	50	5,173.00	50.44	26.96
1-host-1-hr-raw	100	6,177.00	56.88	26.95
1-host-1-hr-raw	200	6,530.00	59.20	26.98
1-host-12-hr	16	1,148.48	38.69	26.52
1-host-12-hr	50	1,920.50	71.18	26.57
1-host-12-hr	100	2,181.00	82.79	26.62
1-host-12-hr	200	2,398.00	91.44	26.64
1-host-12-hr-raw	16	539.20	33.78	27.04
1-host-12-hr-raw	50	704.50	46.87	27.12
1-host-12-hr-raw	100	612.00	41.76	27.17
1-host-12-hr-raw	200	694.00	46.89	27.24
1-host-latest	16	2,141.60	40.24	26.69
1-host-latest	50	3,602.00	77.84	26.69
1-host-latest	100	4,098.00	91.87	26.72
1-host-latest	200	4,356.00	94.94	26.71
8-host-latest	16	541.28	81.87	26.77
8-host-latest	50	599.50	97.52	26.77
8-host-latest	100	648.00	97.85	26.88
8-host-latest	200	832.00	97.40	26.94

表 4. 集群4（32核128GB * 3节点）

queryType	worker	qps	max_cpu	max_mem
1-host-1-hr	16	3,543.04	24.25	28.33
1-host-1-hr	50	9,351.00	34.14	28.89
1-host-1-hr	100	12,567.00	46.42	28.92
1-host-1-hr	200	15,072.00	57.27	28.94
1-host-1-hr-raw	16	2,811.20	15.41	29.11
1-host-1-hr-raw	50	6,086.50	27.85	29.13
1-host-1-hr-raw	100	7,881.00	36.44	29.14
1-host-1-hr-raw	200	8,762.00	40.97	29.17
1-host-12-hr	16	1,394.24	18.85	28.91
1-host-12-hr	50	3,317.50	49.89	28.96
1-host-12-hr	100	4,160.00	67.13	28.96
1-host-12-hr	200	4,808.00	79.02	28.96
1-host-12-hr-raw	16	542.56	17.31	29.22
1-host-12-hr-raw	50	826.50	24.96	29.24
1-host-12-hr-raw	100	669.00	23.55	29.28
1-host-12-hr-raw	200	772.00	24.86	29.32
1-host-latest	16	2,600.00	16.04	28.97
1-host-latest	50	6,590.00	46.11	28.98
1-host-latest	100	8,516.00	65.60	28.99
1-host-latest	200	9,634.00	75.93	28.98
8-host-latest	16	949.12	55.91	29.03
8-host-latest	50	1,525.50	83.49	29.04
8-host-latest	100	1,710.00	93.16	29.06
8-host-latest	200	1,758.00	95.54	29.09

9.搜索引擎

9.1. 引擎简介

Lindorm搜索引擎提供面向海量数据的在线搜索服务，具备高效的索引读写能力、更低成本的数据存储，并支持快速的弹性扩展，广泛应用于站内搜索、订单查询、日志分析、监控APM等场景。

核心能力

- **低成本**
  支持多种高效压缩算法，有效降低索引存储成本，同时借助冷热分离机制，可以将历史数据转储到更低廉的存储介质上。
- **云原生弹性**
  基于云原生存储计算分离架构设计，可快速弹性扩展副本，高效应对流量高峰。
- **高可用**
  分布式高可用设计，集群无单点问题，同时能够智能识别异常流量，通过快速隔离与降级确保系统稳定。
- **生态丰富**
  支持标准SQL语句访问。

场景案例

- **任意多个字段组合查询**
  例如物流订单场景，需要存储大量轨迹物流信息，需要基于多个字段检索查询订单详情。
- **地理位置检索**
  例如交通监控场景，保存大量过车记录，同时会根据车辆牌照、位置信息等条件组合检索查询感兴趣的记录。
- **文本检索**
  例如商品信息检索场景，一般会保存大量商品文本信息，希望通过某个指定短语检索查询符合条件的商品。

服务开通

如何开通搜索引擎，具体请参见开通指南。

使用指南

搜索引擎支持标准SQL访问，同时支持Web式操作，具体操作请参见管理索引表。

9.2. 连接并使用搜索引擎

9.2.1. 通过Search Shell连接并使用搜索引擎

Search Shell是云原生多模数据库Lindorm自带的数据库管理工具，您可以在本地或ECS上安装Search Shell工具，然后通过Search Shell连接Lindorm实例。本文介绍在ECS上通过Search Shell连接Lindorm的方法。

前提条件

- 已安装Java环境，建议安装JDK 1.8及以上版本。
- 已开通Lindorm搜索引擎，具体操作请参见开通指南。
- 已将客户端的IP地址加入到Lindorm实例的白名单中，具体操作请参见设置白名单。
- 已获取Lindorm实例的私网地址，具体操作请参见查看连接地址。

操作步骤

执行下述操作前，请先切换至非Root的用户。

1. 安装Search Shell（如已安装，可跳过）。
   i. 登录ECS实例，执行下述命令下载Search Shell工具。
   ```
   wget https://hbaseuepublic.oss-cn-beijing.aliyuncs.com/lindorm-search-cli.tar.gz
   ```
   ii. 执行下述命令解压Search Shell工具。
   ```
   tar -xzvf lindorm-search-cli.tar.gz
   ```
2. 修改*bin/search.in.sh*文件中的 `ZK_HOST` 地址，您可以通过云原生多模数据库Lindorm控制台获取 `host:port` 。

```
ZK_HOST="host:port"   //host:port为Lindorm搜索引擎的Solr兼容地址的私网地址。
```

获取路径如下图所示。



Search Shell访问示例代码

执行下述命令进入*lindorm-search-cli/bin*目录。

```
cd lindorm-search-cli/bin
```

执行以下代码查看帮助命令。

```
./search-cli
```

使用示例。

- 执行以下代码创建索引表。

  ```
  ./search-cli create_collection -c testIndex -n _indexer_default -shards 2
  ```

  > **说明** 索引名为 `testIndex`，使用默认配置集 `_indexer_default`，分片数设置为 `2`。

- 执行以下代码查看索引表。

  ```
  ./search-cli list_collections
  ```

- 执行以下代码下载配置集。

  ```
  ./search-cli zk downconfig -d . -n _indexer_default
  ```

  > **说明** `_indexer_default` 是Search Shell工具提供的默认配置集，执行上述命令后，*bin*目录会自动创建一个名称为 `conf` 的子文件夹，子文件夹存储的是 `_indexer_default` 的配置集合。

- 上传配置集。

  ```
  ./search-cli zk upconfig -d conf -n myConf
  ```

  > **说明** 可以将配置集修改为自定义的配置集名，如 `myConf`。

- 查看配置集。

  ```
  ./search-cli zk ls /configs
  ```

- 创建基于自定义配置集的索引表。

  ```
  ./search-cli create_collection -c myIndex -n myConf -shards 2
  ```

9.2.2. 通过SQL连接并使用搜索引擎

Lindorm搜索引擎支持SQL JDBC访问，本文介绍如何使用开源的Solr JDBC访问Lindorm搜索引擎。

前提条件

- 已开通Lindorm搜索引擎，具体操作请参见开通指南。
- 已将客户端的IP地址加入到Lindorm实例的白名单中，具体操作请参见设置白名单。
- 已获取**Lindorm搜索SQL地址**，具体操作请参见查看连接地址。

添加Maven依赖

在您的Java应用 `pom.xml` 中添加如下依赖：

```
<dependency>
  <groupId>com.aliyun.lindorm</groupId>
  <artifactId>lindorm-all-client</artifactId>
  <version>2.0.5</version>
</dependency>
```

创建索引

可以通过**集群管理**页面创建索引表，具体操作请参见管理索引表。

连接代码示例

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;
public class SearchSqlDemo {
    public static void main(String[] args) {
        Connection pconn = null;
        Statement stmt = null;
        try {
            //加载搜索引擎Driver
            Class.forName("com.aliyun.lindorm.search.client.Driver");
            //配置Lindorm搜索SQL地址
            String url = "jdbc:lindorm:search:url=http://ld-xxxx:30070";
            Properties props = new Properties();
            props.put("user", "root");
            props.put("password", "root");
            //建立连接
            pconn = DriverManager.getConnection(url, props);
            //创建Statement
            stmt = pconn.createStatement();
            //建表
            stmt.execute("create table if not exists test(c1 int, c2 varchar, primary key(c1))");
            //单条写入
            stmt.execute("upsert into test(c1,c2) values(1,'深圳')");
            stmt.execute("upsert into test(c1,c2) values(2,'上海')");
            //批量写入
            stmt.execute("upsert into test(c1,c2) values(3,'北京'),(4,'广州'),(5,'杭州')");
            //查询
            ResultSet rs = stmt.executeQuery("select * from test order by c1 ");
            System.out.println("#####before delete:");
            while (rs.next()) {
                System.out.println("c1=" + rs.getInt("c1") + ",c2=" + rs.getString("c2"));
            }
            //删除记录
            stmt.execute("delete from test where c1 >3");
            //查询
            rs = stmt.executeQuery("select * from test order by c1 ");
            System.out.println("#####after delete:");
            while (rs.next()) {
                System.out.println("c1=" + rs.getInt("c1") + ",c2=" + rs.getString("c2"));
            }
            //删表
            stmt.execute("drop table if exists test");
        } catch (Throwable e) {
            e.printStackTrace();
        } finally {
            //清理Statement和连接
            //注意：在实际环境中尽量通过连接池来避免频繁创建和释放连接，以便提升性能。
            try {
                if (stmt != null) {
                    stmt.close();
                }
                if (pconn != null) {
                    pconn.close();
                }
            } catch (Throwable e) {
                e.printStackTrace();
            }
        }
    }
}
```

9.3. 用户指南

9.3.1. 开通指南

本章节介绍搜索引擎开通的操作步骤。

前提条件

在购买搜索引擎服务之前，您需要仔细评估实例的资源容量。为了帮助您进行资源规划，我们给出一些通用的建议参考，具体请参见集群容量规划。

购买实例时开通搜索索引

1. 进入Lindorm控制台页面，单击**创建**。
2. 选择搜索引擎的规格。以搜索为主要场景的业务，建议选择大规格的节点，节点个数至少2个，实例规格和节点数都可以后续变更。如果对规格选型有疑问，可以咨询专家服务群或提工单，咨询方法请参见技术支持。



购买实例后开通搜索索引

如果已经开通宽表引擎，需要在宽表基础上增加搜索能力，可以开通搜索索引功能。

1. 选择左侧导航栏的**搜索索引**，并单击**立即开通**按钮。
2. 选择搜索引擎的规格和LTS（原BDS）规格。
3. 单击**立即购买**。

查看开通进度

购买完成后，在Lindorm控制台可以看到实例的状态为**创建中**，当实例状态变为**运行中**时，可以使用该实例。

9.3.2. 查看连接地址

使用连接地址访问云原生多模数据库Lindorm，确保连接的可用性。您可以在控制台查看各引擎的连接地址。

操作步骤

1. 登录Lindorm管理控制台。
2. 在顶部菜单栏的左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID。
4. 在左侧导航栏中选择**数据库连接**。
5. 单击**搜索引擎**页签，可以查看Lindorm实例搜索引擎的访问地址。



9.3.3. 使用示例

搜索引擎支持多语言访问，并且完全兼容开源Solr API，本文介绍使用Solr Java API开发一个简单的读写应用访问搜索引擎。

前提条件

- 已创建Lindorm实例并开通搜索引擎，具体操作请参见开通指南。
- 已获取Lindorm实例搜索引擎的连接地址，具体操作请参见查看连接地址。

操作步骤

1. 配置客户端SDK依赖。
   Lindorm搜索引擎完全兼容开源Solr协议，可直接依赖开源客户端版本，Maven配置如下：

```
<dependency>
  <groupId>org.apache.solr</groupId>
  <artifactId>solr-solrj</artifactId>
  <version>7.7.3</version>
</dependency>
```

2. 创建索引表。
   可通过**集群管理**页面创建索引表，具体操作请参见管理索引表。

代码示例

下面给出一个简单的代码样例：创建连接，写入3行数据，再进行查询。

- 创建连接

```
/**
//公网访问时，本地调试采用HttpSolrClient模式
//前提：通过控制台"访问控制"添加本机IP到白名单
String httpUrl = "http://***:8983/solr/"
HttpSolrClient solrClient = new HttpSolrClient.Builder(httpUrl).build();
**/
// 内网访问建议采用CloudSolrClient模式
String zkHost = "zk1:2181,zk2:2181,zk3:2181/solr"
CloudSolrClient solrClient = new CloudSolrClient.Builder(Collections.singletonList(zkHost), Optional.empty
()).build(); //CloudSolrClient是线程安全的，应用多线程可以共享一个对象
```

- 写入数据

```
// 构造数据
List<SolrInputDocument> docs = new ArrayList<>();
SolrInputDocument doc = new SolrInputDocument();
doc.addField("id", "1");
doc.addField("name_s", "zhangsan");
doc.addField("age_i", 18);
docs.add(doc);
doc = new SolrInputDocument();
doc.addField("id", "2");
doc.addField("name_s", "lisi");
doc.addField("age_i", 20);
docs.add(doc);
doc = new SolrInputDocument();
doc.addField("id", "3");
doc.addField("name_s", "wangwu");
doc.addField("age_i", 35);
docs.add(doc);
// 写入数据
solrClient.add(docs);
```

- 查询

```
// 查询数据
SolrQuery solrQuery = new SolrQuery("name_s:zhangsan");
QueryResponse response = solrClient.query("your_index", solrQuery);
SolrDocumentList documentList = response.getResults();
for(SolrDocument doc : documentList){
    String id = (String)doc.getFieldValue("id");
    //do something
}
solrClient.close();
```

更多样例代码，请参见Java Demo。

9.3.4. 集群容量规划

Lindorm搜索引擎是由多个节点组成的分布式集群，在购买服务之前，需要仔细评估集群的资源容量。为了帮助您进行集群规划，我们给出一些通用的建议参考。

存储容量评估

存储容量评估主要考虑以下几点：

1. 副本数量：默认建议副本为0，搜索引擎基于分布式共享存储，节点宕机后，数据可以自动迁移到其它节点上提供服务，对于部分可靠性要求高的场景，可考虑设置副本数量为1。
2. 索引数据膨胀：数据一般膨胀在20%。
3. 搜索引擎预留：记录事务log、定期的compaction数据合并等，建议预留20%。
4. 操作系统预留：系统默认预留5%的存储空间。
5. 系统安全阈值：为保障整个集群的稳定运行，建议预留20%的存储空间，存储使用率到80%时会自动触发短信告警。

因此，实际的存储空间可以通过下面公式估算：

```
实际空间 = 源数据 * 1.9
```

计算资源评估

不同的业务场景在写入和查询方面的复杂度、比重各有不同，建议先基于存储容量评估选择合适的资源，再进行业务测试确认计算资源是否足够。通常的建议如下：

- 至少选择2个节点，避免出现单节点故障。
- 建议选择较高规格的节点，例如选择16C 64GB的实例规格。
- 如果实际测试发现计算资源不符合，扩容时，建议优先进行纵向扩容，例如把节点规格从4C 16GB的规格升级到8C 32GB，然后再考虑横向扩容增加节点个数。

分片数量评估

每个索引被分成多个分片，数据写入时会基于文档ID自动哈希打散到不同的分片中，在设置分片个数时，通常的建议如下：

- 建议单个分片大小保持在20GB~50GB之间。
- 建议分片数量为节点个数的整数倍，例如实例有2个节点，则创建的索引分片设置为2。
- 日志、订单等有时间属性的场景，建议使用系统自带的分库分表（Alias功能），持续滚动产生新的索引，并且周期性删除旧的索引。

9.3.5. 配置冷存储

云原生多模数据库Lindorm搜索引擎支持配置表级别的冷存储功能，可以将表数据存储至冷存储介质中，从而降低存储成本。本文介绍搜索引擎配置冷存储的具体操作。

前提条件

已下载并安装Search Shell，具体操作请参见通过Search Shell连接并使用搜索引擎。

应用场景

当某张表基本上不需要写入，并且只存在少量查询时，可以将表的属性设置为冷存储。设置为冷存储的表执行查询操作时查询结果返回会比较慢。

操作步骤

1. 开通冷存储，具体操作请参见开通冷存储。
2. 设置目标表的冷存储属性，`collection_name`为目标表名。
   i. 执行`RELOAD`操作，避免影响冷存储数据转移耗时。
   ```
   ./search-cli reload_collection -c collection_name
   ```
   ii. 执行以下语句设置目标表的冷存储属性。
   ```
   ./search-cli modify_collection -c collection_name -s COLD
   ```
3. 执行以下语句将目标表的数据迁移至冷存储介质中。
   ```
   ./search-cli major_compaction -c collection_name
   ```
4. 迁移操作完成后，执行以下语句释放热存储空间。由于将目标表的数据迁移至冷存储介质中的操作是异步操作，释放前建议等待30~60分钟。
   ```
   ./search-cli reload_collection -c collection_name
   ```

9.4. 集群管理

9.4.1. 登录集群管理系统

搜索引擎提供Web方式管理集群。本文介绍通过Web方式登录搜索引擎的集群管理系统。

前提条件

- 已创建Lindorm实例，具体操作请参见创建实例。
- 已将本机的外网IP添加至白名单。具体操作请参见设置白名单。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏中，选择**搜索引擎 > 集群管理**。
5. 在**UI访问**区域中单击**ClusterManager**登录集群管理系统。

9.4.2. 管理索引表

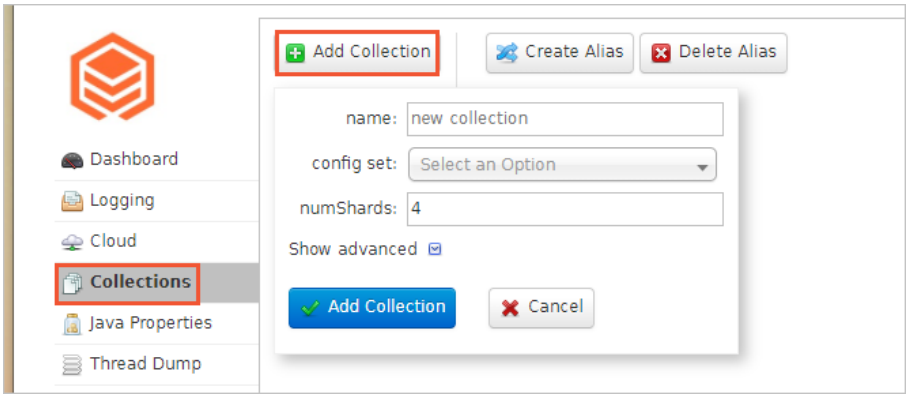
本文介绍通过集群管理系统管理索引表。

名词解释

- Collection：索引表。
- Shard：索引表分片。
- Replica：索引表分片的副本。

创建索引表

1. 登录搜索引擎的集群管理系统。
2. 在左侧导航栏，单击**Collections**。
3. 单击**Add Collection**。



4. 主要参数配置说明如下表。

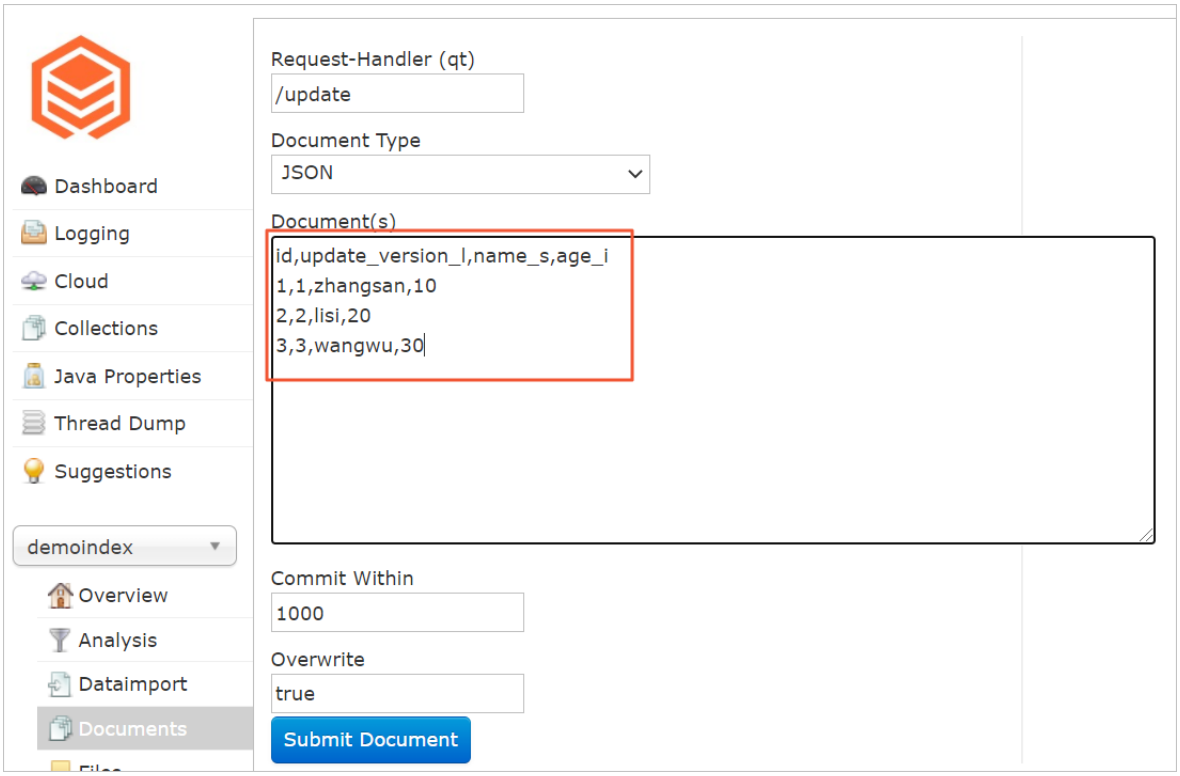
参数名称	参数说明
name	索引表的名称。
config set	索引表的配置，默认配置名称为*_indexer_default*。
numShards	索引表的分片个数，建议设置为节点个数的整数倍。比如搜索引擎有2个节点，numShards可以设置为2或4，同时请注意MaxShardsPerNode的限制。
replicationFactor	每个索引表的分片副本个数，默认值为1。
maxShardsPerNode	设置每个节点最多有几个该索引的分片，默认为1。 **说明** 参数配置需要满足maxShardsPerNode×节点个数≥numShards×replicationFactor，否则无法创建索引表。
autoAddReplica	当分片所在的节点异常时，是否需要在其它节点恢复该分片，建议设置为True。

5. 单击**Add Collection**。

写入数据

1. 在左侧导航栏的**Collection Selector**下拉列表中选择创建的索引表名称。
2. 选择**Documents**，进入写数据页面。
3. 在**Document(s)**区域中写入以下代码。

```
id,update_version_l,name_s,age_i
1,1,zhangsan,10
2,2,lisi,20
3,3,wangwu,30
```

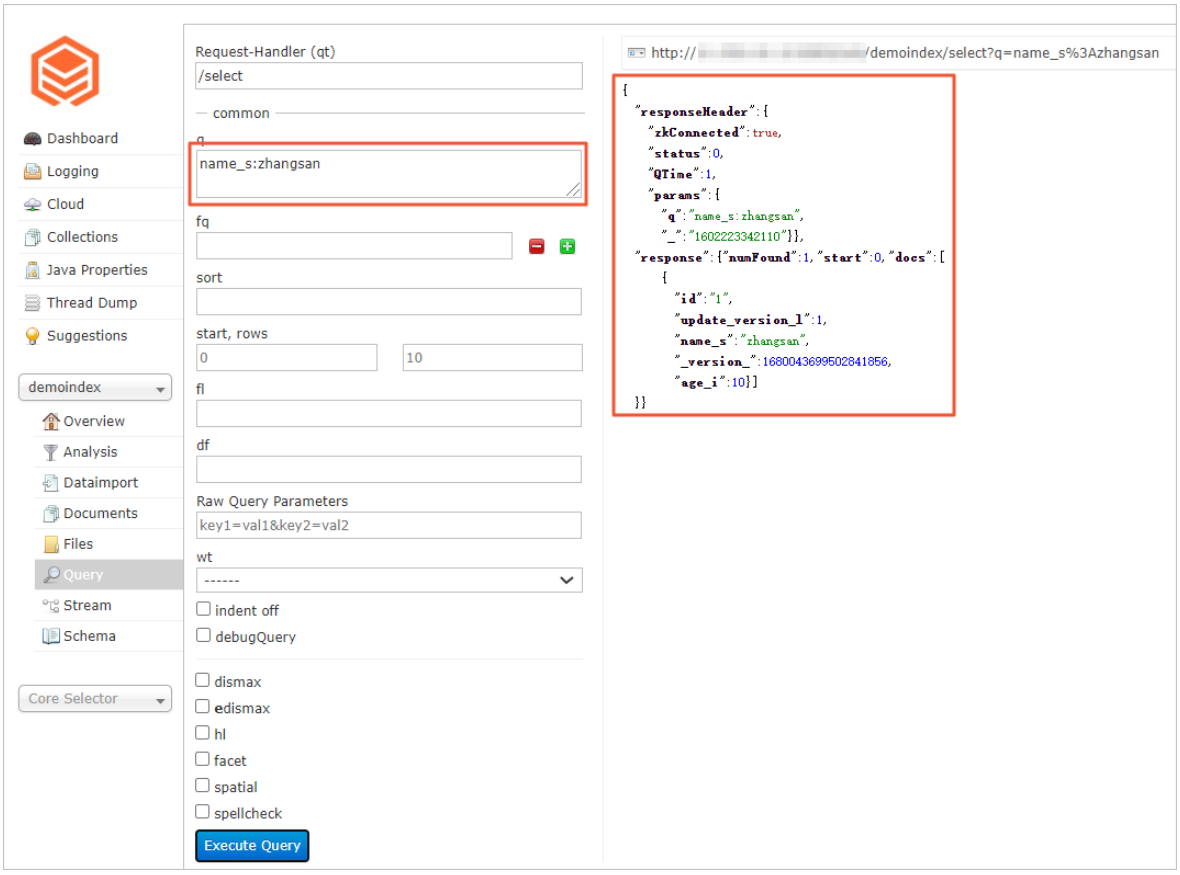


说明 代码示例表示写入三行数据，每行数据有四列。`id` 和 `update_version_l` 为系统定义列，必须写。name_s的后缀_s表示该列为字符串类型，age_i的后缀_i表示该列为整数类型。

4. 单击**Submit Document**。

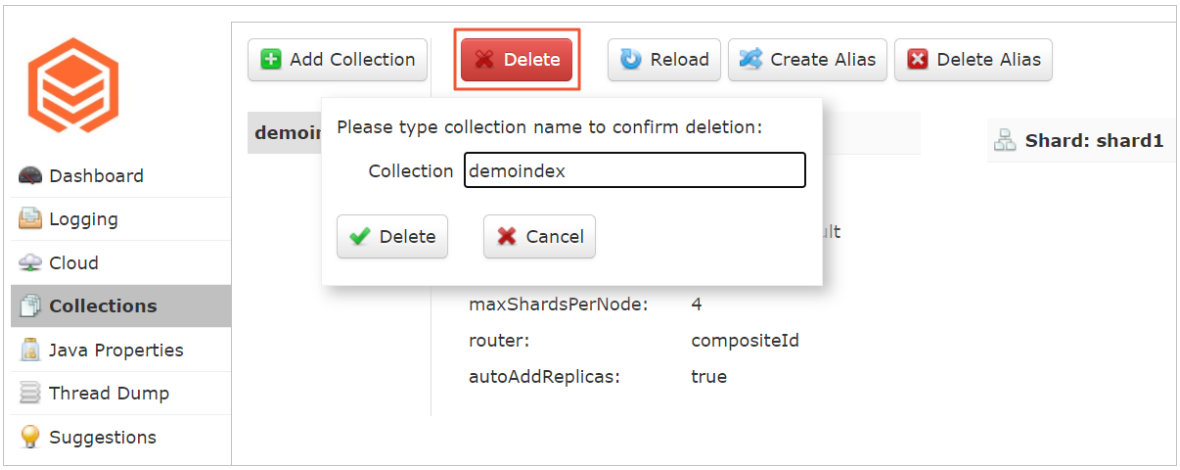
查询索引数据

1. 在左侧导航栏中选择**Query**。
2. 在**q**区域输入查询的条件。
3. 单击**Execute Query**，查询结果如下图所示。



删除索引

1. 在左侧导航栏中选择**Collections**。
2. 选择目标索引表名称，单击**Delete**。
3. 在**Collection**文本框中输入目标索引表名称，如下图所示。



4. 单击**Delete**。

9.5. 性能白皮书

9.5.1. 测试环境

本次测试针对开源自建Solr搜索引擎与云Lindorm搜索引擎进行了基础场景的读写性能对比。测试使用的工具为自研的benchmark工具，如需了解该工具的详细设计，可通过Lindorm钉钉值班或者提工单联系我们。

环境配置

- 网络类型为VPC网络，保证客户端与服务端处于同一个服务可用区。
- 所有测试均在华东2（上海）地域的可用区G完成。
- 开源自建Solr使用社区7.7.3版本，存储基于HDFS。
- Lindorm Search的版本为7.7.7，存储基于Lindorm Store。

服务端配置

名称	内容
Core节点配置	16C32G(ecs.c5.4xlarge)
Core节点数量	2
Core节点磁盘类型	云盘SSD
Core节点单节点磁盘大小	400GB

压力端配置

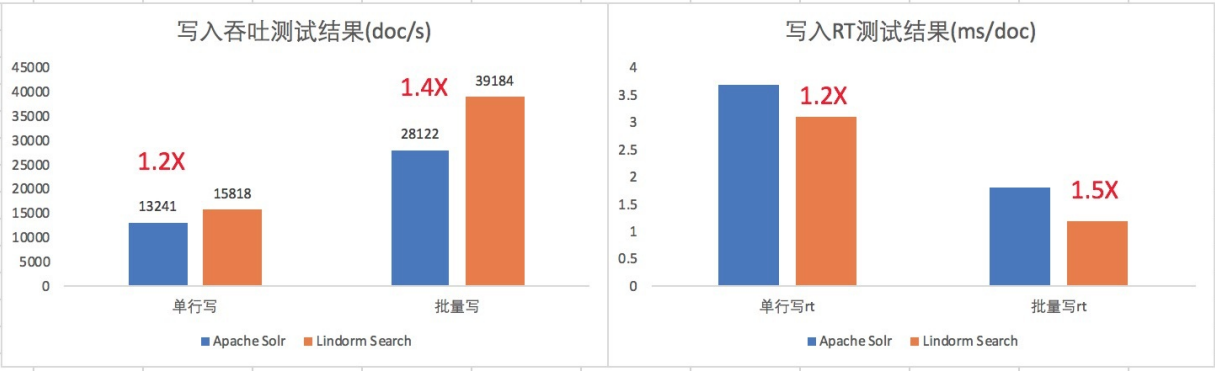
名称	内容
Core节点配置	16C32G(ecs.c5.4xlarge)
Core节点数量	1

9.5.2. 测试结果

本章节介绍吞吐量对比和平均rt对比的测试结果。

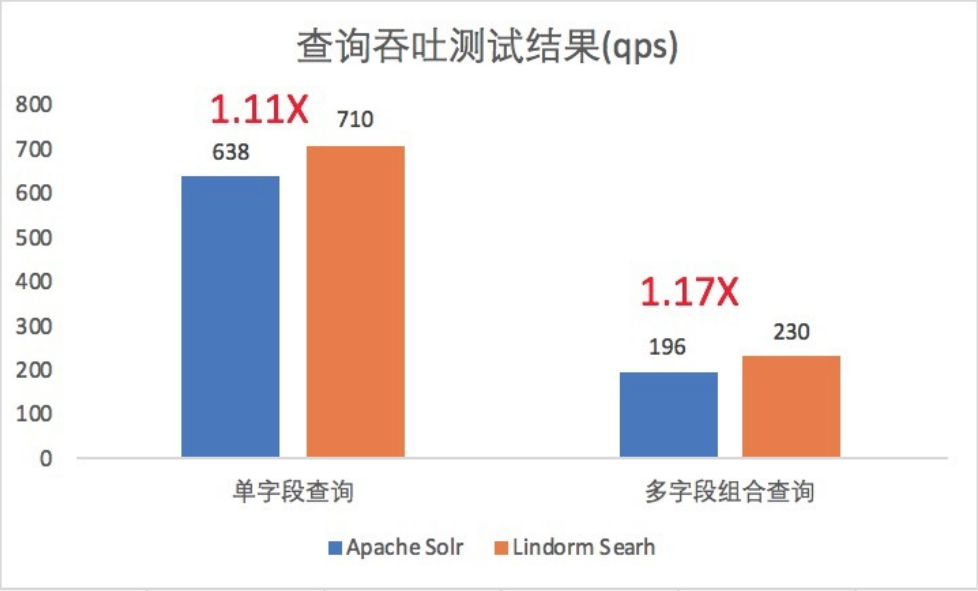
写入吞吐和写入RT测试结果

索引表：2个shard，副本为1。写入20个field，10个long型，10个string型，单行平均长度200字节。



查询吞吐测试结果

索引表：1个shard，副本为1.按照上面的写入模型，5亿条基础数据，分别执行单字段查询、多字段组合查询。



9.6. 开发指南

9.6.1. 更新配置集

本文介绍如何更新云原生多模数据库Lindorm搜索引擎的配置集。

前提条件

下载并安装Shell，具体请参见通过Search Shell连接并使用搜索引擎。

下载默认的配置集模板

执行以下命令下载默认的配置集模板 `_indexer_default` ，在其基础上进行编辑，添加业务自定义的配置。

```
cd lindorm-search-cli/bin
./search-cli zk ls /configs          // 查询已有的配置集列表
./search-cli zk downconfig -d . -n _indexer_default     // 下载配置集_indexer_default到当前目录
```

执行上述命令成功后，在当前目录下会生成一个*conf*的目录，其中有两个重要的文件： `managed-schema` 和 `solrconfig.xml` 。

创建新的配置集

以下展示一个简单示例：

1. 打开 `managed-schema` 文件。
2. 增加两个新的索引列定义，增加内容如下：

```
<field name="name" type="string" indexed="true" stored="true" required="false" multiValued="false" />
<field name="age" type="pint" indexed="true" stored="true" docValues="true"  multiValued="false" />
```

```
<uniqueKey>id</uniqueKey>

<!--
  id, _version_, _text_, _root_, update_version_l
  系统保留的字段名，不要修改
-->
<field name="id" type="string" indexed="true" stored="true" required="true" multiValued="false" />
<field name="_version_" type="plong" indexed="false" stored="false"/>
<field name="_root_" type="string" indexed="true" stored="false" docValues="false" />
<field name="_text_" type="text_general" indexed="true" stored="false" multiValued="true"/>
<field name="update_version_l" type="plong" indexed="true" stored="true"/>
<field name="type1" type="string" indexed="false" stored="true" docValues="false" />
<field name="type2" type="string" indexed="true" stored="true" docValues="false" />
<!--
  TODO: 需要您定义的业务字段列表：
    1、整形的建议使用type=pint
    2、长整形的建议使用type=plong
    3、字符类型建议使用type=string
    4、文本类型建议使用type=text_general，IK中文分词建议使用type=text_ik
    5、字段名称格式：建议全部由字母组成，区分大小写，不能以数字开头
    6、stored="false"，如果仅仅是查询，不需要存储
    7、indexed="false"，如果仅仅是存储，不需要查询
    8、建议只增加field，不要轻易修改已经定义的field，可能会引起索引数据损坏，无法恢复
    9、更多可参考：http://lucene.apache.org/solr/guide/documents-fields-and-schema-design.html
    例如：
    <field name="myfield" type="string" indexed="true" stored="false" multiValued="false" />
 -->
<field name="name" type="string" indexed="true" stored="true" required="false" multiValued="false" />
<field name="age" type="pint" indexed="true" stored="true" docValues="true"  multiValued="false" />
<!--
  动态定义字段，方便业务使用，不需要按照上面的格式一个一个的定义字段。
  例如，
  当您写入myfield_i时，会自动识别为*_i
  当您写入myfield_l时，会自动识别为*_l
  当您写入myfield_s时，会自动识别为*_s
-->
<dynamicField name="*_i"  type="pint"    indexed="true"  stored="true"/>
<dynamicField name="*_is" type="pints"    indexed="true"  stored="true"/>
```

> **说明**
>
> - `name` 是string类型，`age` 是基本int类型（pint代表int，plong代表long），两个列都需要建立索引 `indexed=true`，并且都需要存储原始数据 `stored=true`。
> - 每增加一个新列，都需要在文件中定义好，当需要增加非常多的列时，定义起来会比较复杂。此时，可以使用Search服务提供的动态列能力，参考 `managed-schema` 中的 `dynamicField` 定义，有了它之后不需要额外定义每个列，只需要在写入数据时指定的列名称后缀与定义保持一致即可。
> - 例如：`name_s` 可以自动匹配 `*_s`。`age_i` 可以自动匹配 `*_i`。

3. 上传配置集。
   修改完后，可以上传自定义的配置集（建议每个collection对应一个配置集），执行如下命令：

   ```
   ./search-cli zk upconfig -d conf/ -n myconf
   ```

4. 在集群管理系统查看配置集是否上传成功，登录集群管理系统请参见登录集群管理系统。
   选择**Cloud > Tree > /cnfigs**，查看配置集 `myconf` 是否上传成功。

相关建议

- 建议使用dynamicField功能，不单独定义每个索引列，避免频繁修改 `managed-schema` 文件。
- 每个索引需要有自己的配置集，不建议多个索引共享配置集。
- 如果需要自己定义配置，请下载 `_indexer_default` 配置集后，在此基础上进行修改。

9.6.2. 索引数据可见性

写入Search服务中的数据，只有等到 `commit` 后才是可见的，何时执行commit是可以配置的，当前有两种commit方式：soft commit、hard commit。

solrconfig.xml中的配置

创建索引时需要指定配置集 `config set` ，配置集中的 `solrconfig.xml` 文件可以用来控制索引数据的可见性。如何获取 `solrconfig.xml` ，请参见更新配置集。

- soft commit

```
<autoSoftCommit>
  <maxTime>${solr.autoSoftCommit.maxTime:15000}</maxTime>
</autoSoftCommit>
```

写入的数据多久后可以查询，默认是15秒。

- hard commit

```
<autoCommit>
  <maxTime>${solr.autoCommit.maxTime:30000}</maxTime>
  <openSearcher>false</openSearcher>
</autoCommit>
```

写入的数据多久后刷新到磁盘中，默认是30秒。

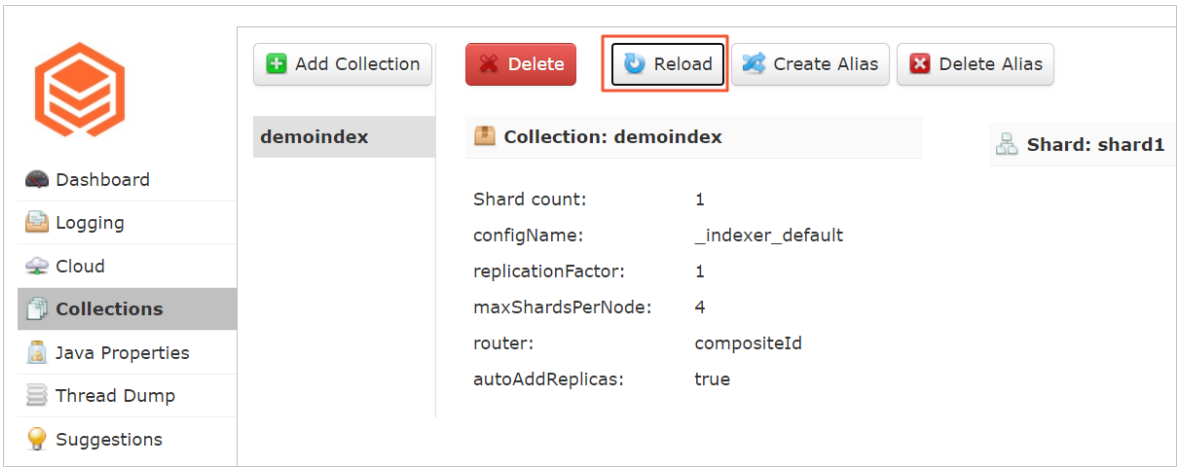
> 说明
> - soft commit的时间要小于hard commit时间。
> - 一般不建议配置较小的时间，这会导致服务端频繁刷新和open searcher，影响写入性能。采用默认值即可。

如何生效

1. 下载需要修改的配置集，请参见更新配置集。
2. 修改 `soft commit` 和 `hard commit` 的时间。
3. 更新配置集。

```
./solr zk upconfig -d conf/ -n myconf
```

4. 在**Collections**页面，选中目标Collection，单击**Reload**。



客户端参数

如果不想修改服务端的solrconfig.xml文件，在单独写数据到Search时可以显示调用commit来达到数据可见的目的。

- commitWithinMs

```
// 第二个参数commitWithinMs可以控制当前写入的数据多久后可以查询。
public UpdateResponse add(SolrInputDocument doc, int commitWithinMs) throws SolrServerException, IOExcepti
on;
//例如：代表10秒后数据可查询。
cloudSolrClient.add(doc, 1000);
```

- commit API

```
// 写完数据后，显示调用commit接口，让服务端在多久后执行commit，保证数据可查询。
public UpdateResponse commit(boolean waitFlush, boolean waitSearcher, boolean softCommit) throws SolrServe
rException, IOException;
//例如：服务端立即执行soft commit。
cloudSolrClient.commit(false, false, true);
```

参考文档

请参见官方文档。

9.6.3. 分库分表（Alias功能）

本文介绍分库分表功能的使用场景和使用方法。

背景信息

- 表变更业务逻辑中设置了访问某个表A，突然有一天需要修改为表B，此时只能修改配置进行线上变更。
- 分库分表
  业务大部分场景只访问最近一周的数据，可以每隔一周新建一张表来存储，这样可以确保高效的查询热数据。在这个场景中需要自己来维护表的创建和删除，带来一定的业务复杂性。

适用场景

- 时间序列场景
  业务数据具有明显的时间特性，可以基于时间来创建不同的索引，这样既能降低单个索引的大小，又能提升查询性能。整个过程中业务不需要自己维护索引创建和删除。
- 重建索引场景
  在不影响已有索引查询下，重建新的索引，待索引建完后，指向新的索引访问。整个过程中业务不需要代码变更。

创建Alias指向已有的索引表

```
curl "http://solrhost:8983/solr/admin/collections?action=CREATEALIAS&name=your_alias_name&collections=your_c
ollection_name_A"
```

上面的URL表示：创建一个Alias名为your_allias_name，其指向一个索引表your_collection_name_A。这样业务逻辑中可以只设置访问your_alias_name，内核会自动转发请求到真实的索引表上。假设某一天需要变更索引表名为your_collection_name_B，执行一次更改Alias命令。

修改Alias

```
curl "http://solrhost:8983/solr/admin/collections?
action=ALIASPROP&name=your_alias_name&collections=your_collection_name_B"
```

这样，业务代码上不需要任何变更即可访问新的索引表。

自动分表

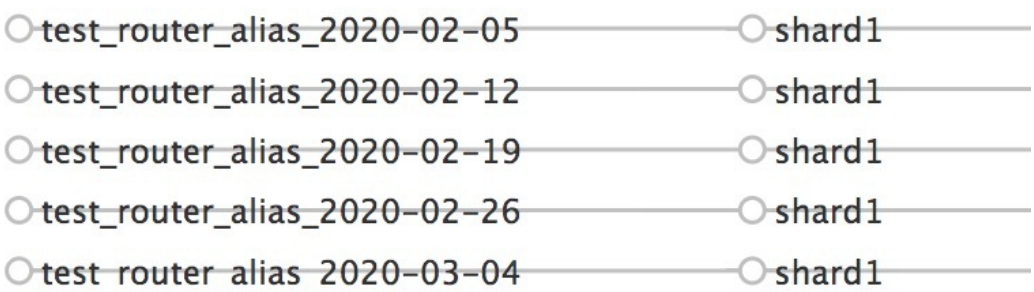
搜索服务支持按照时间字段自动分表，大大简化业务逻辑。下面以具体的示例来介绍：业务要求以周为单位创建索引表，并且能够自动删除旧的索引表。

```
curl "http://solrhost:8983/solr/admin/collections?action=CREATEALIAS&name=test_router_alias&router.start=NOW
-30DAYS/DAY&router.autoDeleteAge=/DAY-90DAYS&router.field=your_timestamp_l&router.name=time&router.interval=
%2B7DAY&router.maxFutureMs=8640000000&create-collection.collection.configName=_indexer_default&create-collec
tion.numShards=2"
```

参数	值	说明
router.start	NOW-30DAYS/DAY	第一个collection创建的时间点，样例中给出的NOW-30DAYS/DAY代表以30天前开始新建索引
router.interval	+7DAY	间隔多久创建新的索引表，样例中给出的是每隔7天新建一个索引表
router.autoDeleteAge	/DAY-90DAYS	自动淘汰多久前的索引表，样例中给出的是淘汰90天前的索引表，其值必须大于router.start
router.field	your_timestamp_l	分表的时间字段，默认业务中需要携带这个字段，并指定时间值，例如当前的系统时间戳System.currentTimeMillis()
router.maxFutureMs	8640000000	代表最大容忍写入的时间字段your_date_dt与当前时间的差值，防止写入过大的时间字段或者过小的时间值，样例中给出的是100天，即不能写入一条数据的时间比当前时间大100天或小100天
collection.collection.configName	_indexer_default	代表创建的索引表依赖的配置集，可以设置为自己的配置集名称，参考配置集更新
create-collection.numShards	2	创建的索引表shard个数，默认为2

上面的业务含义，以30天前（假设今天是3月7日）开始创建索引，每隔7天新建一个索引，your_timestamp_l，并且它的值与当前时间在100天以内，周期性的删除90天过期的索引。

效果如下



> 说明
> - 业务必须带有时间字段，可以为Date类型，也可以为Long类型。
> - 查询时，默认是查询全部索引表。此时，可以单独指定查询某个索引表。需要通过API或者URL获取到所有collection列表，然后提取其中的时间字段来判断真实访问的collection，可参见下面的代码样例。

删除Alias

普通的Alias可以直接通过下面的命令删除。

```
curl "http://solrhost:8983/solr/admin/collections?action=DELETEALIAS&name=your_alias_name"
```

具有自动分表的Alias删除，还需要主动删除关联的collection。

1. 取得关联Alias的collection列表。

   ```
   curl "http://solrhost:8983/solr/admin/collections?action=LIST"
   ```

   其中collection名称以 `test_router_alias` 开头的都是关联该Alias的collection。

2. 删除Alias。

   ```
   curl "http://solrhost:8983/solr/admin/collections?action=DELETEALIAS&name=test_router_alias"
   ```

3. 删除所有collection

   ```
   curl "http://solrhost:8983/solr/admin/collections?action=DELETE&name=collection_name"
   ```

参考文档

CREATEALIAS：CREATEALIAS

LIST：LIST

如何指定long型时间进行查询

```
import org.apache.solr.client.solrj.SolrQuery;
import org.apache.solr.client.solrj.impl.CloudSolrClient;
import org.apache.solr.client.solrj.impl.ClusterStateProvider;
import org.apache.solr.client.solrj.response.QueryResponse;
import org.apache.solr.common.SolrDocument;
import org.apache.solr.common.util.StrUtils;
import java.time.Instant;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeFormatterBuilder;
import java.time.temporal.ChronoField;
import java.util.AbstractMap;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Optional;
public class SolrDemo {
  private static final DateTimeFormatter DATE_TIME_FORMATTER = new DateTimeFormatterBuilder()
      .append(DateTimeFormatter.ISO_LOCAL_DATE).appendPattern("[_HH[_mm[_ss]]]")
      .parseDefaulting(ChronoField.HOUR_OF_DAY, 0)
      .parseDefaulting(ChronoField.MINUTE_OF_HOUR, 0)
      .parseDefaulting(ChronoField.SECOND_OF_MINUTE, 0)
      .toFormatter(Locale.ROOT).withZone(ZoneOffset.UTC);
  private static final String zkHost = "localhost:2181/solr";
  private CloudSolrClient cloudSolrClient;
  private ClusterStateProvider clusterStateProvider;
  public SolrDemo() {
    cloudSolrClient = new CloudSolrClient.Builder(
        Collections.singletonList(zkHost), Optional.empty()).build();
    cloudSolrClient.connect();
    clusterStateProvider = cloudSolrClient.getClusterStateProvider();
  }
  public void close() throws Exception {
    if (null != cloudSolrClient) {
```

```
      cloudSolrClient.close();
    }
  }
  private List<String> findCollection(String aliasName, long start, long end) {
    List<String> collections = new ArrayList<>();
    if (start > end) {
      return collections;
    }
    //基于[start, end]寻找合适的collection
    if (clusterStateProvider.getState(aliasName) == null) {
      // 拿到当前aliasName对应的所有collection列表
      // test_router_alias_2020-03-04, test_router_alias_2020-02-26, test_router_alias_2020-02-19, test_rout
er_alias_2020-02-12, test_router_alias_2020-02-05
      List<String> aliasedCollections = clusterStateProvider.resolveAlias(aliasName);
      // 从collection名称中提取出时间日期
      // 2020-03-04T00:00:00Z=test_router_alias_2020-03-04,
      // 2020-02-26T00:00:00Z=test_router_alias_2020-02-26,
      // 2020-02-19T00:00:00Z=test_router_alias_2020-02-19,
      // 2020-02-12T00:00:00Z=test_router_alias_2020-02-12,
      // 2020-02-05T00:00:00Z=test_router_alias_2020-02-05
      List<Map.Entry<Instant, String>> collectionsInstant = new ArrayList<>(aliasedCollections.size());
      for (String collectionName : aliasedCollections) {
        String dateTimePart = collectionName.substring(aliasName.length() + 1);
        Instant instant = DATE_TIME_FORMATTER.parse(dateTimePart, Instant::from);
        collectionsInstant.add(new AbstractMap.SimpleImmutableEntry<>(instant, collectionName));
      }
      // 根据查询时间找出合理的collection
      Instant startI = Instant.ofEpochMilli(start);
      Instant endI = Instant.ofEpochMilli(end);
      for (Map.Entry<Instant, String> entry : collectionsInstant) {
        Instant colStartTime = entry.getKey();
        if (!endI.isBefore(colStartTime)) {
          collections.add(entry.getValue());
          System.out.println("find collection: " + entry.getValue());
          if (!startI.isBefore(colStartTime)) {
            break;
          }
        }
      }
    } else {
      collections.add(aliasName);
    }
    System.out.println("query " + collections);
    return collections;
  }
  public void run() throws Exception {
    try {
      // [2020-03-07 2020-03-10]
      long start = 1583538686312L;
      long end = 1583797886000L;
      String aliasName = "test_router_alias";
      String collections = StrUtils.join(findCollection(aliasName, start, end), ',');
      QueryResponse res = cloudSolrClient.query(collections, new SolrQuery("*:*"));
      for (SolrDocument sd : res.getResults()) {
        System.out.println(sd.get("id") + " " + sd.get("gmtCreate_l"));
      }
    } finally {
      cloudSolrClient.close();
    }
  }
  public static void main(String[] args) throws Exception {
    SolrDemo solrDemo = new SolrDemo();
    solrDemo.run();
    solrDemo.close();
```

```
  }
}
```

9.6.4. 预排序

在数据量非常大的情况下，排序不仅影响查询速度，而且占用系统资源。但如果本身存储的数据已经按照某个字段预先排序，检索性能会有明显提升，在大数据量上的对比尤为明显。因此，使用预排序功能够大幅度提升数据检索性能，优化搜索引擎的服务能力。本章节介绍使用预排序功能的操作步骤。

操作步骤

1. 修改solrconfig.xml中的MergePolicy，具体请参见Customizing Merge Policies。
2. 查询时，指定参数segmentTerminateEarly=true即可，请参见以下demo演示：

```
<mergePolicyFactory class="org.apache.solr.index.SortingMergePolicyFactory">
  <str name="sort">timestamp desc</str>
  <str name="wrapped.prefix">inner</str>
  <str name="inner.class">org.apache.solr.index.TieredMergePolicyFactory</str>
  <int name="inner.maxMergeAtOnce">10</int>
  <int name="inner.segmentsPerTier">10</int>
</mergePolicyFactory>
```

此时插入数据将会按照timestamp字段进行预先倒序排序，执行查询如下：

```
curl "http://localhost:8983/solr/testcollection/query?q=*:*&sort=timestamp+desc&rows=10&segmentTerminateEarly=true"
```

参数上加上 `segmentTerminateEarly=true` 后，显示效果会比没有设置预排序的快很多，尤其排序数据量到达T级别之后，效果更为明显。

> 说明
> - 查询时，指定的sort必须与配置的MergePolicy中指定的一致，否则无效。
> - 查询时需要指定segmentTerminateEarly参数，否则会进行全排。
> - 使用了这个预排序返回的结果中，“numFound”是不准确的。

9.6.5. Lindorm搜索SQL语法手册

9.6.5.1. 名词解释

本文介绍Lindorm搜索SQL语法的常见术语。

名词	对应Lindorm中的含义	描述
Table	Table	表，构建表时需要定义表的Schema结构，包括表名、表的列名和列的类型。
PRIMARY KEY	PRIMARY KEY	主键，用于标识一行数据，可以唯一确定一行数据的分布位置。

9.6.5.2. 数据类型

本文介绍Lindorm搜索SQL支持的数据类型。

数据类型	对应的Java类型	说明
BOOLEAN	java.lang.Boolean	长度为1字节。 • 0表示false。 • 1表示true。

数据类型	对应的Java类型	说明
TINYINT	java.lang.Byte	长度为1字节。
SMALLINT	java.lang.Short	长度为2字节。
INTEGER	java.lang.Integer	长度为4字节。
BIGINT	java.lang.Long	长度为8字节。
FLOAT	java.lang.Float	长度为4字节，符号位反转。
DOUBLE	java.lang.Double	长度为8字节，符号位反转。
VARCHAR/CHAR	java.lang.String	变长字符串类型，支持中文。

9.6.5.3. 函数说明

9.6.5.3.1. Count函数

Count函数用于计算给定列的非空值，可以对查询返回的行进行计数。

示例

```
select count(*) from test;
select count(*) from test where c1 = 1;
select count(c1) from test where c1 = 1;
```

9.6.5.3.2. Sum函数

Sum函数用于求和，计算指定列的查询返回值之和。

示例

```
select sum(c1) from test;
```

9.6.5.3.3. Avg函数

Avg函数用于计算指定列查询返回值的平均值。

示例

```
//计算test表格中c1列的平均值。
SELECT avg(c1) FROM test;
```

9.6.5.3.4. Max函数和Min函数

Max函数和Min函数用于计算指定列的查询返回的最大值和最小值。

示例

```
SELECT max(c1) FROM test;
SELECT min(c1) FROM test;
```

9.6.5.3.5. As函数

As是将请求的结果以别名方式命名。

示例

```
select count(*) as countRow from test;
select c1 as testC from test where c1 = '2';
```

9.6.5.4. 使用限制说明

本文介绍Lindorm搜索SQL语法的使用限制。

支持的条件表达式

- OR或者AND
- LIKE或者NOT LIKE
- IS NULL或者NOT NULL
- BETWEEN XX AND XX
- 支持数量符号：=、>、<、>=、<=、<>
- IN或者NOT IN
- 支持GROUP BY分组

```
SELECT c1, Sum(c2) FROM dt WHERE p2 IN ('a', 'b') GROUP BY c1;
```

- 支持ORDER BY排序

```
SELECT * FROM dt WHERE p2 IN ('a', 'b') ORDER BY c1;
SELECT * FROM dt WHERE p2 IN ('a', 'b') ORDER BY c1 desc;
```

使用限制

- 不支持DCL（Data Control Language，数据控制语句）。
- 长度限制。
  - 单个主键列最大长度为2 KB。
  - 所有主键列长度之和不超过30 KB。
  - 单个非主键列最大长度不超过2 MB。
- 不支持表达式聚合，例如：`SUM(c1+1)`。
- WHERE条件中不支持表达式，例如：`WHERE (a*10) >100`。
- 仅支持单表查询，不支持多表JOIN、SUBQUERY、UNION等查询操作。

9.6.5.5. DDL

9.6.5.5.1. DESCRIBE/SHOW

DESCRIBE语法可以查看指定表的详细信息。SHOW语法可以查看所有的Schema（DataBase）。

示例

- 执行以下语句查看所有的Schema（DataBase）。

```
SHOW databases;
```

- 在对应的Schema下，执行以下语句可以查看所有Table，默认Default Schema下的Table。

```
SHOW tables;
```

- 执行以下语句查看指定表`test`的详细信息，包括表的列信息和列类型。

```
DESCRIBE test;
```

9.6.5.5.2. CREATE TABLE

CREATE TABLE语法用来创建Lindorm搜索表。

语法

```
CREATE TABLE [ IF NOT EXISTS ] table_name
     '('
      column_definition
      ( ',' column_definition )*
       [ ',' PRIMARY KEY '(' primary_key ')' ]
      ')'
     [ PARTITION BY RANGE TIME '(' time_partition_key=value ')' ]
     [ PARTITION BY HASH '(' hash_partition_key (',' hash_partition_key)* ')' ]
     [ PARTITIONS intValue ]
     WITH [ table_options ];
column_definition      ::=  column_name data_type
primary_key            ::=  column_name [ ',' column_name ]
time_partition_key     ::=  column_name
hash_partition_key     ::=  column_name ['(' salt_factor=intValue ')']
```

> **说明** 创建表语句支持 `IF NOT EXISTS`，其中主键由一个或多个列组合而成。

参数

`table_options` 参数说明如下表：

参数	说明
RANGE_TIME_PARTITION_START	表示创建索引表操作前几天开始创建分区。适用于有历史数据的场景，如果历史数据的时间戳比开始分区的时间还要小，则会报错。
RANGE_TIME_PARTITION_INTERVAL	表示间隔几天创建新分区。例如：`RANGE_TIME_PARTITION_INTERVAL='7'` 表示每隔一周创建一个新分区。
RANGE_TIME_PARTITION_TTL	表示保留几天的分区数据。例如：`RANGE_TIME_PARTITION_TTL='180'` 表示保留半年的分区数据，历史分区数据会被自动清理掉。
RANGE_TIME_PARTITION_MAX_OVERLAP	如果写入的数据时间点是将来的时间，这个参数表示写入的数据时间点与当前时刻的最大时间间隔，单位为天。
RANGE_TIME_PARTITION_FIELD_TIMEUNIT	表示业务指定的时间分区字段单位，默认单位为毫秒（ms）。 • 时间分区字段单位设置为秒（s），数字长度为10位。 • 时间分区字段单位设置为毫秒（ms），数字长度为13位。

数据分区参数说明：

- `PARTITION BY RANGE TIME`：表示数据按照时间范围分区。
- `PARTITION BY HASH`：表示数据按照HASH分区。
- `PARTITIONS intValue`：自定义数据的分区数量。

示例

- 创建表dt。

  ```
  CREATE TABLE dt (
    p1 integer, p2 varchar, c1 integer, c2 integer, constraint pk PRIMARY KEY(p1,p2));
  ```

- 对于时间序列的数据，可以按照时间范围分区，例如按照周或者月进行分区，同一时间范围内的数据将会聚集存储，并且历史分区数据会被自动清理掉。

- 按照业务的时间列 `orderTime` 分区，从30天前开始，每7天自动分区，默认保留90天。

```
CREATE TABLE dt_partition (
  storeId integer, goodsId varchar, goodsPrice integer, orderTime bigint, constraint pk primary key(stor
eId))
  PARTITION BY RANGE TIME(orderTime) PARTITIONS 16
  WITH (RANGE_TIME_PARTITION_START='30', RANGE_TIME_PARTITION_INTERVAL='7', RANGE_TIME_PARTITION_TTL='90
', RANGE_TIME_PARTITION_MAX_OVERLAP='10');
```

- 按照业务的时间列 `orderTime` 分区，从半年前开始，每1个月自动分区，默认保留半年，时间分区字段单位设置为秒。

```
CREATE TABLE dt_partition (
  storeId integer, goodsId varchar, goodsPrice integer, orderTime bigint, constraint pk primary key(stor
eId))
  PARTITION BY RANGE TIME(orderTime) PARTITIONS 16
  WITH (RANGE_TIME_PARTITION_START='180', RANGE_TIME_PARTITION_INTERVAL='30', RANGE_TIME_PARTITION_TTL='
180', RANGE_TIME_PARTITION_MAX_OVERLAP='10', RANGE_TIME_PARTITION_FIELD_TIMEUNIT='s');
```

- HASH分区将数据进行散列存储，从而避免出现数据的热点问题。在数据写入量较大的场景中可以很好的实现数据均衡，搜索引擎默认按照主键进行HASH分区，同时也支持自定义的业务分区键。使用一级HASH分区时，如果自定义的HASH键存在热点问题，则可能导致大量数据写入同一个分区中，影响查询和写入性能，建议考虑多级分区，对数据进一步打散。
  - 创建搜索表，按照storeId列进行HASH分区（也就是一级HASH分区），分区数量为16。

```
CREATE TABLE dt_partition (
  storeId integer, goodsId varchar, goodsPrice integer, constraint pk primary key(storeId))
  PARTITION BY HASH(storeId)
  PARTITIONS 16;
```

  - 创建搜索表，按照storeId列和goodsId列组合进行HASH分区（也就是二级HASH分区），加盐因子为2，分区数量为16，最多支持三级分区组合。

```
CREATE TABLE dt_partition (
  storeId integer, goodsId varchar, goodsPrice integer, constraint pk primary key(storeId))
  PARTITION BY HASH(storeId(salt_factor=2),goodsId)
  PARTITIONS 16;
```

> **注意**
>
> - 加盐因子（ `salt_factor` ）是进一步散列相同storeId列值的数据，通常设置为一个较小的数值，并且数值与分区数量是两倍的数量关系。如果分区数量为16，当salt_factor大于4时，则无法散列数据。加盐因子的其他值表示如下：
>   - `salt_factor=1` ：表示相同storeId列的数据，不同goodsId列的数据会被散列到分区总数量的一半中。
>   - `salt_factor=2` ：表示相同storeId列的数据，不同goodsId列的数据会被散列到分区总数量的1/4中。
>   - `salt_factor=3` ：表示相同storeId列的数据，不同goodsId列的数据会被散列到分区总数量的1/8中。
> - 多级HASH分区（二级HASH分区或者三级HASH分区）除了可以散列数据，在查询数据场景中也大大提高了查询效率。例如在上述二级HASH分区中，查询数据时同时输入storeId列和goodsId列的过滤信息会确定在一个分区中进行数据检索，减少数据的扫描范围从而提高查询效率。
> - 使用二级HASH分区时，一级分区键（Lindorm搜索表的第一列即storeId列）必须设置加盐因子。使用三级HASH分区时，一级分区键和二级分区键都必须设置加盐因子。
> - 自定义分区键的值有以下限制：
>   - 分区键值不可更改。
>   - 分区键值不能为空。

- 根据业务需求，如果需要将数据按照HASH分区和时间范围分区，可以将两个分区策略组合使用。

```
CREATE TABLE dt_partition (
  storeId integer, goodsId varchar, goodsPrice integer, orderTime bigint, constraint pk primary key(stor
eId))
  PARTITION BY RANGE TIME(orderTime)
  PARTITION BY HASH(storeId) PARTITIONS 16
  WITH (RANGE_TIME_PARTITION_START='180', RANGE_TIME_PARTITION_INTERVAL='30', RANGE_TIME_PARTITION_TTL='
180', RANGE_TIME_PARTITION_MAX_OVERLAP='10');
```

9.6.5.5.3. DROP TABLE

DROP TABLE语法用来删除搜索表。

语法

```
DROP TABLE [ IF EXISTS ] table_name;
```

参数

参数	描述
table_name	需要删除的表名。

示例

```
DROP TABLE dt;
DROP TABLE IF EXISTS dt;
```

9.6.5.6. DML

9.6.5.6.1. SELECT

通过SELECT语法可以查询一个或多个表格中的行数据。

语法

```
 SELECT ( select_clause | '*' )
        FROM table_name
        [ WHERE where_clause ]
        [ ORDER BY ordering_clause ]
        [ LIMIT integer ] | [LIMIT integer, integer]
select_clause    ::=  selector [ AS identifier ] ( ',' selector [ AS identifier ] )
selector         ::=  column_name
                      | term
                      | function_name '(' [ selector ( ',' selector )* ] ')'
                      | COUNT '(' '*' ')'
where_clause     ::=  relation ( AND|OR relation )*
relation         ::=  column_name operator term
                      '(' column_name ( ',' column_name )* ')' operator tuple_literal
operator         ::=  '=' | '<' | '>' | '<=' | '>=' | '!=' | IN
group_by_clause  ::=  column_name, (',' column_name)*
ordering_clause  ::=  column_name [ ASC | DESC ] ( ',' column_name [ ASC | DESC ] )*
```

参数

SELECT语法中部分参数说明如下：

参数	说明
table_name	需要查询数据的表名。

参数	说明
LIMIT	• 参数后跟一个数字，表示最多返回的行数。 • 参数后跟两个数字，并且由英文逗号（,）分隔。例如：`LIMIT 5,20` 表示跳过5行，最多返回20行。

示例

- 简单的查询操作

```
SELECT * FROM dt;             --选择全部列
SELECT c1,c2 FROM dt;                    --选择部分列
SELECT * FROM dt LIMIT 10;
SELECT * FROM dt WHERE c1 = 10;
SELECT * FROM dt WHERE c1 > 10 LIMIT 5, 20;           --跳过5行，最多返回20行
SELECT * FROM dt WHERE c1 > 10 LIMIT 20 OFFSET 5;      --与limit 5,20相同
SELECT c2 FROM dt WHERE c1 > 10;
SELECT c2 AS xx FROM dt WHERE c1 > 10;
```

- 基本的聚合操作

```
SELECT COUNT(*) FROM dt;
SELECT SUM(c2) FROM dt;
SELECT SUM(c2), AVG(c2),MIN(c2),MAX(c2) FROM dt;
```

- WHERE条件中IS NULL的用法

```
SELECT c1, c2 FROM dt WHERE c1>1 OR c2 IS NULL;
```

- 别名用法

```
SELECT COUNT(*) AS countRow FROM t1;
SELECT c1 AS testC FROM t1 WHERE c1 = '2';
```

- IN的用法

```
SELECT * FROM dt WHERE c1 IN ('a', 'b');
```

9.6.5.6.2. UPSERT

UPSERT语法是INSERT与UPDATE的结合体，表示行存在时执行UPDATE语法，不存在时执行INSERT。执行UPSERT的时候必须指定完全的primary key的相关列的信息。UPSERT语法支持批量写入。

语法

```
UPSERT INTO table_name
             '('columns')'
             VALUES '(' values ')'
columns ::=  column_name, columns
values  ::=  term, values,'(' values ')'
```

> **说明** 支持批量写入，指定需要写入的相关列，可以在Values关键字中确定几行数据。

示例

- 单次写入

```
UPSERT INTO dt (p1,p2,c1,c2) VALUES(10, 20, 30, 40);
UPSERT INTO dt (p1,p2,c2) VALUES(10, 20, 40);    -- 仅写入一行的部分列
UPSERT INTO dt (p1,p2,c1) VALUES(10, 20, 30);    -- 再写入同一行的其他列
```

- 批量写入

```
UPSERT INTO dt (p1,p2,c1,c2) VALUES(1,2,3,4), (2,3,4,5), (3,4,5,6);
```

9.6.5.6.3. DELETE

DELETE语法用来删除表中的行数据，可指定过滤条件。

语法

```
DELETE FROM table_name WHERE where_clause;
```

示例

```
DELETE FROM dt WHERE p1 = 10 AND p2 = 20;
DELETE FROM dt WHERE p1 > 10;
```

9.7. 数据生态

9.7.1. 连接Lindorm Search和Grafana

本文介绍如何通过Solr插件将Lindorm Search和Grafana连接起来，使用Grafana可视化工具更好地监控和分析Lindorm Search的数据。

背景信息

Grafana是一个基于Go开发的指标数据图形化展示工具，可以实时从多个插件中采集数据并以图表展示，插件包括InfluxDB/OpenTSDB/MySQL/PosgreSQL/Elasticsearch/Solr，更多信息，请参见插件列表。

步骤1：安装Grafana和插件

1. 下载Grafana。具体操作，请参见下载Grafana。
2. 添加开源Solr插件。具体操作，请参见添加开源插件。
   i. 执行如下命令下载开源Solr插件。
   ```
   git clone https://github.com/pueteam/datasource-plugin-solr
   ```
   ii. 执行如下命令拷贝插件到Grafana目录。
   ```
   cp datasource-plugin-solr grafana-7.0.5/plugins-bundled -R
   ```
   iii. 执行如下命令重启Grafana服务。
   ```
   service grafana-server restart
   ```

步骤2：添加数据源

1. 登录Grafana。
2. 在左侧菜单栏，选择  > Data Sources。
3. 在Data Sources页签，单击Add data source。
4. 在Add data source页面，搜索框中搜索*Solr*，单击Solr对应的Select按钮。
5. 配置数据源。

   主要参数说明如下表所示。

参数	说明
Name	数据源名称。
URL	目前只支持HTTP方式获取，具体操作，请参见获取Solr的URL地址。
SolrCloud Mode	打开SolrCloud Mode开关，其他参数保持默认配置。

6. 单击Save & Test。

步骤3：添加仪表盘

1. 在左侧菜单栏，选择 > Dashboard。
2. 在New Panel面板中，单击Choose Visualization。
3. 配置模板变量。
   i. 在New dashboard页面右上角单击图标。
   ii. 在Type参数中选择standard。
   iii. Grafana中的Collection、Timestamp和Fields参数自动从Solr中获取的。
   iv. Query参数保持默认配置。配置页面如下图所示。



4. 单击Add。

获取Solr的URL地址

1. 登录阿里云Lindorm控制台。请参见Lindorm控制台。
2. 在**实例列表 > 数据库连接**中切换到**搜索引擎**页面，单击**开通外网地址**获取URL地址，如获取URL地址所示。

获取URL地址



10.文件引擎（兼容HDFS）

10.1. 引擎简介

Lindorm文件引擎是面向海量非结构化数据的分布式文件存储服务，与多模引擎共享存储，LindormDFS支撑，充分利用了对象存储的低成本与高可靠、块存储的高性能等优势，并通过高低速分层存储、副本共享去重、智能冷热转存等技术，面向用户提供弹性低成本、100%HDFS协议兼容的云原生大数据存储能力，可使用开源HDFS客户端直接访问。
在计算分析、备份归档、数据导入等场景，Lindorm支持外部系统直接访问多模引擎的底层文件，从而大幅提升数据的读写效率。例如用户可以在离线计算系统直接生成底层数据格式的物理文件，LindormDFS中，以减少对在线服务的影响。

核心优势

- **低成本：**最低至0.12元/GB/月（不同区域可能有变化，具体以购买页为准），未来可进一步下降。
- **文件系统底座：**支持高性能元数据读写操作。支持通过目录批量移动文件。
- **HDFS协议级兼容：**可使用开源HDFS客户端接入，无需担心云厂商锁定（vender lock-in）。兼容Hive、Spark、Impala、Presto、Flink、Yarn、Kylin，EMR、DLA、MaxCompute等开源生态与云上生态。
- **存储计算分离：**存储容量按需购买，计算与存储解耦。无需担心存储计算绑定导致的资源浪费。
- **冷热数据混合：**支持冷热数据混存，以文件或者文件夹为粒度设置冷热，不同性价比存储介质组合降低成本。

10.2. 连接并使用文件引擎

10.2.1. 通过HDFS Shell连接并使用文件引擎

使用HDFS Shell

通过HDFS Shell工具，你可以交互式管理文件引擎的文件，包括文件查询、文件删除、文件权限管理和修改文件名称等，本章介绍最基本的HDFS Shell的使用命令，完整的指令可以通过help命令查看，更多内容请参考 Apache HDFS Shell 介绍 。使用前请先查看 使用开源HDFS客户端访问 进行基本环境的配置。

常用命令

1. 从本地上传数据到文件引擎常用的指令。

```
1.put ： 从本地文件系统中复制单个或多个源路径到目标文件引擎。也支持从标准输入中读取输入写入目标文件引擎。
2.copyFromLocal ： 除了限定源路径是一个本地文件外，和put命令相似。
3.moveFromLocal ： 把文件从本地剪切到文件引擎。
4.appendToFile ： 从本地文件系统追加一个或多个src到目标文件引擎。
```

2. 操作文件引擎数据的常用指令

```
1. cp ： 将文件从源路径复制到目标路径。这个命令允许有多个源路径，此时目标路径必须是一个目录。
2. mv ： 将文件从源路径移动到目标路径。这个命令允许有多个源路径，此时目标路径必须是一个目录。不允许在不同的文件系统间移动文件。
3. chown ： 改变文件的拥有者。使用-R将使改变在目录结构下递归进行。命令的使用者必须是超级用户 。
4. chgrp ： 改变文件所属的组。使用-R将使改变在目录结构下递归进行。命令的使用者必须是文件的所有者或者超级用户。
5. chmod ： 变文件的权限。使用-R将使改变在目录结构下递归进行。命令的使用者必须是文件的所有者或者超级用户。
6. mkdir ： 接受路径指定的uri作为参数，创建这些目录。其行为类似于Unix的mkdir -p，它会创建路径中的各级父目录。
7. du ： 显示目录中所有文件的大小，或者当只指定一个文件时，显示此文件的大小。
8. df ： 查看文件的使用情况 。
9. cat ： 将路径指定文件的内容输出到stdout。
10. rm ： 删除指定的文件。只删除非空目录和文件。
11. ls ： 如果是文件，则按照如下格式返回文件信息：
          文件名 <副本数> 文件大小 修改日期 修改时间 权限 用户ID 组ID
          如果是目录，则返回它直接子文件的一个列表，就像在Unix中一样。目录返回列表的信息如下：
          目录名 <dir> 修改日期 修改时间 权限 用户ID 组ID 。
```

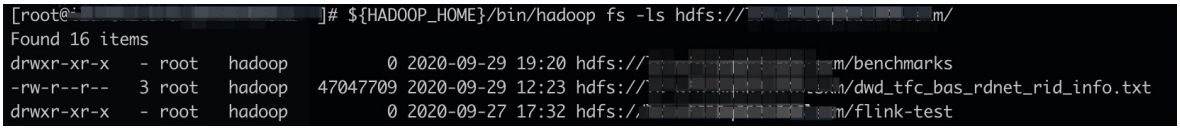
3. 从文件引擎下载数据到本地的常用指令。

```
1. get ： 复制文件到本地文件系统。可用-ignorecrc选项复制CRC校验失败的文件。使用-crc选项复制文件以及CRC信息。
2. getmerge ： 接受一个源目录和一个目标文件作为输入，并且将源目录中所有的文件连接成本地目标文件。
3. copyToLocal ： 除了限定目标路径是一个本地文件外，和get命令类似。
```

示例

1. 查询目录下面的文件，${实例Id}为文件引擎的实例Id，请根据您的实际情况进行修改。

```
${HADOOP_HOME}/bin/hadoop fs -ls hdfs://${实例ID}/
```



2. 上传文件，${实例Id}为文件引擎的实例Id，请根据您的实际情况进行修改。

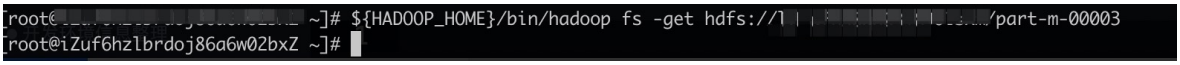
```
${HADOOP_HOME}/bin/hadoop fs -put test.txt hdfs://${实例ID}/
```



3. 下载文件到本地，${实例Id}为文件引擎的实例Id，请根据您的实际情况进行修改。

```
${HADOOP_HOME}/bin/hadoop fs -get hdfs://${实例ID}/xxxxx
```

xxxxx 为你要下载的文件名称。



4. 删除文件，${实例Id}为文件引擎的实例Id，请根据您的实际情况进行修改。

```
${HADOOP_HOME}/bin/hadoop fs -rm hdfs://${实例ID}/xxxxx
```



xxxxx 为你要删除的文件名称，删除文件，会将文件移到Trash目录下面，不会被真正清除；如需彻底清除，则执行 `hadoop fs -expunge` 命令，即清空回收站。

10.2.2. 通过开源HDFS客户端连接并使用文件引擎

本文介绍如何使用开源HDFS客户端访问文件引擎。

前提条件

- 已安装Java环境，JDK版本不能低于1.7。
- 已设置环境变量，假设Java安装位置为 `/opt/install/java` 。

```
JAVA_HOME=/opt/install/java
PATH=/opt/install/java/bin:$PATH
```

- 访问文件引擎的节点需要加入Lindorm白名单，添加方法请参见设置白名单。

下载客户端

您可以从Apache官网下载hadoop2.7.3版本SDK `hadoop-2.7.3.tar.gz` 。

配置Hadoop

1. 下载 `hadoop-2.7.3` 发布包。
2. 执行 `tar -zxvf hadoop-2.7.3.tar.gz` 命令，解压缩SDK包 。
3. 执行 `export HADOOP_HOME=/installDir/hadoop-2.7.3` 命令，设置环境变量。
4. 执行 `cd $HADOOP_HOME` 命令，进入hadoop目录。

5. 修改 `etc/hadoop/hadoop-env.sh` 文件，并添加**准备运行环境**章节中设置的 `JAVA_HOME` ，假设Java安装在 `/opt/install/java` 。

```
# set to the root of your Java installation
export JAVA_HOME=/opt/install/java
```

6. 修改 `etc/hadoop/hdfs-site.xml` 文件， `hdfs-site.xml` 文件中需要修改的内容如下所示，其中实例ID需要替换为真实的实例ID。

```
<configuration>
  <property>
        <name>dfs.nameservices</name>
        <value>${实例ID}</value>
    </property>
    <property>
       <name>dfs.client.failover.proxy.provider.${实例ID}</name>
       <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
    </property>
    <property>
       <name>dfs.ha.automatic-failover.enabled</name>
       <value>true</value>
    </property>
    <property>
       <name>dfs.ha.namenodes.${实例ID}</name>
       <value>nn1,nn2</value>
    </property>
     <property>
       <name>dfs.namenode.rpc-address.${实例ID}.nn1</name>
       <value>${实例ID}-master1-001.lindorm.rds.aliyuncs.com:8020</value>
    </property>
    <property>
       <name>dfs.namenode.rpc-address.${实例ID}.nn2</name>
       <value>${实例ID}-master2-001.lindorm.rds.aliyuncs.com:8020</value>
    </property>
</configuration>
```

> **说明**
> 您可以在Lindorm控制台->文件引擎中单击一键生成配置项按钮自动生成配置文件，详情请参见开通指南。

常用操作示例

1. 上传本地文件。
   - 创建目录。

```
$HADOOP_HOME/bin/hadoop fs -mkdir hdfs://${实例ID}/test
```

   - 准备一个文件，将其上传到文件引擎。

```
echo "test" > test.log
$HADOOP_HOME/bin/hadoop fs -put test.log hdfs://${实例ID}/test
```

2. 查看上传的文件。

```
 $HADOOP_HOME/bin/hadoop fs -ls hdfs://${实例ID}/test
```

3. 下载文件到本地。

```
$HADOOP_HOME/bin/hadoop fs -get hdfs://${实例ID}/test/test.log
```

> 说明
> 其中**实例ID**请替换为真实的实例ID。

10.2.3. 通过HDFS FUSE连接并使用文件引擎

本章介绍如何使用HDFS FUSE访问文件引擎。

准备运行环境

- 安装JDK并设置JAVA_HOME，检查JDK版本：`$JAVA_HOME/bin/java -version`
  - CentOS/RedHat

```
sudo yum install java-1.8.0-openjdk-devel -y
```

  - Ubuntu

```
sudo apt-get update
sudo apt-get install openjdk-8-jdk -y
```

- 安装FUSE库：
  - CentOS/RedHat

```
sudo yum install fuse fuse-devel fuse-libs -y
```

  - Ubuntu

```
sudo apt-get update
sudo apt-get install fuse libfuse-dev -y
```

下载FUSE客户端

当前提供FUSE客户端下载（基于开源hadoop-2.7.7版本）：下载链接

安装FUSE客户端

1. 客户端包解压缩：`tar -zxf ldfs-fuse-2.7.tar.gz && cd ldfs-fuse-2.7`
2. 修改连接地址：获取当前实例ID，例如 ld-uxxxxxxxxxx，修改`etc/hadoop/hdfs-site.xml`文件。示例：

   ```
   sed -i 's/\${INSTANCE_ID}/ld-uxxxxxxxxxx/g' etc/hadoop/hdfs-site.xml
   ```

3. 创建挂载目录：`sudo mkdir /ldfs_fuse && sudo chown $(whoami):$(whoami) /ldfs_fuse`
4. 挂载到目录：`/bin/bash bin/mount_to.sh /ldfs_fuse`
5. 查看FUSE盘：`df -h`
6. 卸载（可选）：`fusermount -u /ldfs_fuse`

> 注意
> 脚本中的实例ID（ld-uxxxxxxxxxx）与挂载路径（/ldfs_fuse）均为参考示例，安装过程中请修改为实际值

常用操作示例

- 写文件：`echo 'test' > /ldfs_fuse/test`
- 读文件：`cat /ldfs_fuse/test`
- 删除文件：`rm -f /ldfs_fuse/test`

> **注意**
> HDFS FUSE挂载后的数据盘仅支持顺序写，不支持随机写

更多操作

- FUSE 默认文件副本数为 3，如需更改默认值，可修改配置文件 `etc/hadoop/hdfs-site.xml` 设置。例如以下配置设置副本数为 2（修改后需要重新挂载生效）：

```
<property>
   <name>dfs.replication</name>
   <value>2</value>
</property>
```

10.3. 用户指南

10.3.1. 如何选择文件引擎规格

Lindorm文件引擎（LindormDFS）是面向海量非结构化数据的分布式文件存储服务，与多模引擎共享存储底座，提供数据库级的安全可靠性。LindormDFS与开源HDFS保持100%通信协议兼容，**使用开源客户端可直接访问**，无缝接入所有开源大数据生态与云计算生态。

存储规格选择

当您选择Lindorm文件引擎作为大数据、数据湖的底层存储时，推荐您选用**容量型云存储**类型。

Lindorm存储以预留空间方式计费，需要确保LindormDFS的预留存储空间超出您的最大使用空间10%以上

> **说明**
> 容量型存储在保持低成本的同时，提供高吞吐顺序读写能力。但随机读能力较弱，适用于大数据计算场景。

文件引擎节点规格选择

文件引擎节点负责承担读写带宽，管理Block数据块，提供HDFS通信协议支持。可以从两个维度确定您的场景需要的文件引擎节点数量：

- 数据量维度：每个文件引擎节点可支撑10TB～50TB存储容量
- 带宽维度：每个文件引擎节点可支撑 100MB/s~200MB/s 数据传输带宽

可以根据您的需求，从数据量维度与带宽维度计算出所需的文件引擎节点数量并取最大值作为您需要购买的文件引擎节点数量。

10.3.2. 开通指南

本章介绍如何开通文件引擎存储服务。

> **说明**
> 文件引擎LindormDFS是Lindorm底层存储，**无需额外购买**即可开通使用。

LindormDFS存储容量在同一Lindorm实例内共享。LindormDFS对HDFS协议级兼容，支持使用开源HDFS客户端进行访问，详情请参见使用开源HDFS客户端访问文档。

控制台开通

1. 登录Lindorm管理控制台。
2. 在**实例列表**页，单击目标实例ID。
3. 在**实例详情**页面，选择左侧导航栏的**文件引擎** 。
4. 单击**立即开通**按钮，开通文件引擎。



访问配置生成

HDFS客户端访问文件引擎需要事先配置两个文件：

- core-site.xml：配置默认的访问前缀
- hdfs-site.xml：配置该访问串对应的nn的信息

配置好以上两个文件后，当使用HDFS客户端访问文件引擎时，控制台可以自动生成文件中的配置信息，如您需要同时访问多个文件引擎，只要将hdfs-site的配置信息添加到您的hdfs-site中，即可通过HDFS连接串进行访问。

操作步骤

1. 登录Lindorm管理控制台。
2. 在**实例列表**页，单击目标实例ID。
3. 在**实例详情**页，单击左侧导航栏的**数据库连接**。
4. 在**数据库连接**页，单击**文件引擎**页签。
5. 单击**一键生成配置项** ，生成hdfs-site和core-site配置信息。



设置白名单

请将需要访问文件引擎的节点加入Lindorm白名单。添加方法请参考设置白名单。

> 说明
> 文件引擎暂不支持公网访问。

> 警告
> 文件引擎LindormDFS是多模引擎数据的共享存储，开通文件存储服务后，多模引擎数据变为可见。使用文件引擎过程中应注意**数据安全**，如因误操作导致的数据丢失等问题需由客户自身承担。多模引擎数据存放路径请查看LindormDFS目录结构说明。

10.3.3. 数据存储目录结构说明

本章介绍文件引擎(LindormDFS)默认的数据存储目录结构。

目录	说明
/${实例ID}-global /${实例ID}_xxx	宽表引擎数据存储目录
/solr	搜索引擎数据存储目录
/tsdb	时序引擎数据存储目录

> **说明**
> - ${实例ID}：lindorm实例ID。
> - xxx：实例所在的区域，如cn-shanghai则表示为上海的实例。

10.3.4. 设置数据冷热存储策略

本文介绍如何在Lindorm文件引擎上设置数据冷热存储策略。

前提条件

Lindorm支持在一套系统中配置多种存储规格，各个存储规格特点请参考存储规格选择。
当您的实例配置为**标准型云存储和容量型存储**或**性能型云存储和容量型云存储**冷热存储组合时，可以运用本文档配置冷热数据存储策略。

设置数据冷热存储策略

- 您可以通过开源HDFS客户端设置数据冷热存储策略，客户端配置请参考使用开源HDFS客户端访问。
- 您可以通过以下命令设置某个目录的冷热数据存储策略，该目录下的所有文件（递归包含）均使用该策略进行存储。

```
$HADOOP_HOME/bin/hdfs storagepolicies -setStoragePolicy -path <PATH> -policy  <POLICY>
```

> **说明**
> POLICY的取值如下：
> - HOT：数据存放在标准型云存储中或性能型云存储中。当您不进行任何配置时，文件引擎默认使用该策略进行数据存储。
> - COLD：数据存放在容量型云存储中。

10.3.5. 透明压缩

本文档主要介绍如何在lindorm文件引擎上开启透明压缩功能。

前提条件

您购买实例时，存储类型必须选择**容量型云存储**。

> **说明** 开启透明压缩功能之后创建的文件才会进行数据压缩，之前创建的文件则不会进行数据压缩。

开启透明压缩

1. 使用开源HDFS客户端访问文件引擎。
2. 执行以下命令开启某个数据目录的透明压缩功能。

   ```
   hdfs dfs -setfattr -n user.lhdfs_compress_enabled /<数据目录的路径>
   ```

   如果您要关闭某个数据目录的透明压缩功能，可以执行以下命令。

   ```
   hdfs dfs -setfattr -x user.lhdfs_compress_enabled /<数据目录的路径>
   ```

3. 执行以下命令查看某个数据目录是否已经成功开启透明压缩功能。

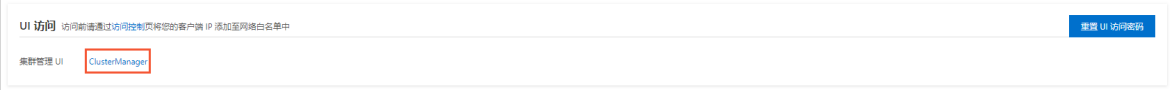
   ```
   hdfs dfs -getfattr -d /<数据目录的路径>
   ```

如果查询结果如下，表示透明压缩功能开启成功。

```
$ hdfs dfs -getfattr -d hdfs://ld-xxxx/tmp
# file: hdfs://ld-xxxx/tmp
user.lhdfs_compress_enabled
```

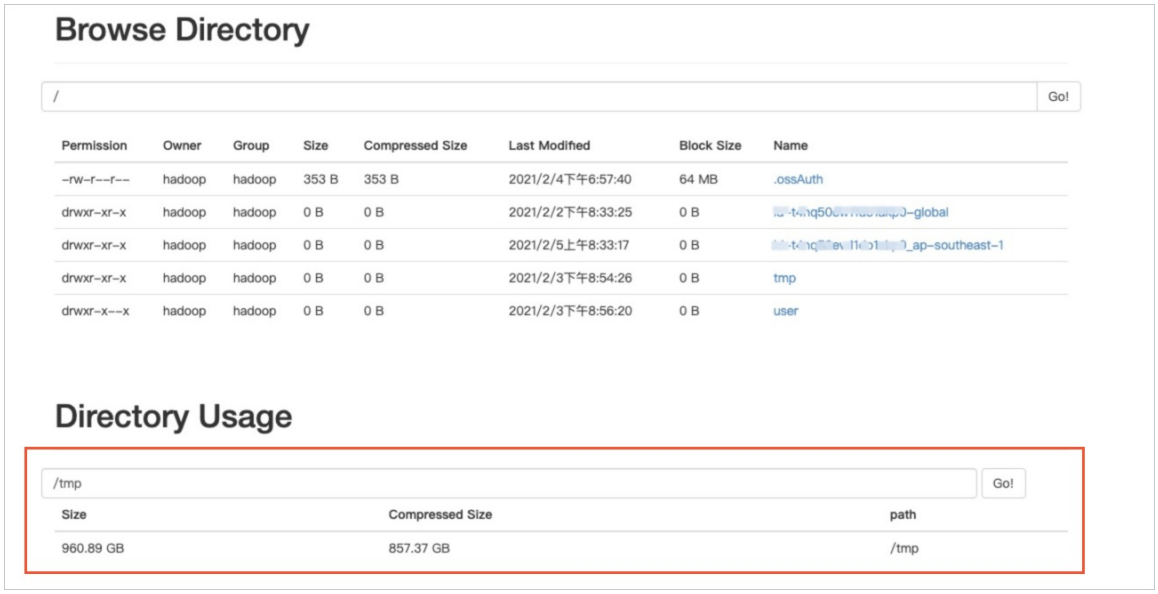
查看透明压缩的效果

1. 登录Lindorm控制台。
2. 在**实例列表**页面，定位到已开启透明压缩功能的实例ID，单击实例ID链接。
3. 在左侧导航栏，单击**文件引擎 > 集群管理**。
4. 在**UI访问**页面，单击**ClusterManager**。



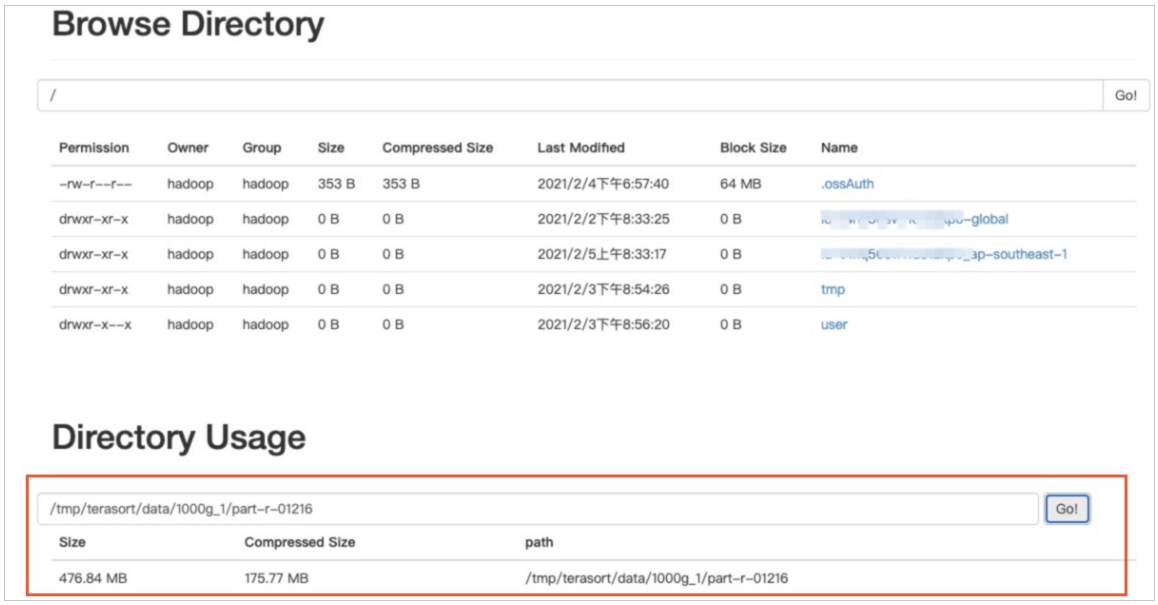
> **说明** 第一次访问集群管理系统，需要重置UI访问密码。

5. 在集群管理系统的菜单栏，选择**Utilities > Browse the file system**。
6. 在**Browse Directory**页面，您可以通过以下方式查看某个数据目录或者文件压缩前后的大小。
   - 查看某个数据目录压缩前后的大小。
     在**Directory Usage**区域的输入框中输入数据目录的路径，然后单击**GO!**进行查询。查询结果如下图所示，您可以看到*/tmp*压缩前的数据大小为960.89 GB，压缩之后的数据大小为857.37 GB。



   - 查看某个文件压缩前后的大小。

在**Directory Usage**区域的输入框中输入文件路径，然后单击**GO!**进行查询。查询结果如下图所示，您可以看到*/tmp/terasort/data/1000g_1/part-r-01216*压缩前的数据大小为476.84 MB，压缩之后的数据大小为175.77 MB。



10.4. 开发指南

10.4.1. 使用Spark访问

本文介绍在文件引擎上搭建及使用Apache Spark的方法。

运行环境准备

- 开通文件引擎，详情请参见开通指南。
- 在计算节点上安装JDK，版本不能低于1.8。
- 在计算节点上安装Scala。
  Scala下载地址：官方链接，其版本要与使用的Apache Spark版本相兼容。
- 下载Apache Hadoop压缩包。
  Apache Hadoop下载地址：官方链接。建议您选用的Apache Hadoop版本不低于2.7.3，本文档中使用的Apache Hadoop版本为Apache Hadoop 2.7.3。
- 下载Apache Spark压缩包。 Apache Spark下载地址：官方链接。选用Apache Spark版本时请注意该版本要与您当前选用的Apache Hadoop版本相兼容，本文中使用的Apache Spark版本为2.4.3。

> **说明**
> 本文档的操作步骤中涉及的安装包版本号、文件夹路径，请根据实际情况进行替换。

Apache Hadoop 配置

1. 解压Apache Hadoop压缩包到指定的目录。

```
tar -zxvf hadoop-2.7.3.tar.gz -C /usr/local/
```

2. 修改 `hadoop-env.sh` 配置文件。

   i. 执行如下命令打开 `hadoop-env.sh` 配置文件。

   ```
   vim /usr/local/hadoop-2.7.3/etc/hadoop/hadoop-env.sh
   ```

   ii. 配置 `JAVA_HOME` 。

   ```
   export JAVA_HOME=${jdk安装目录}
   ```

3. 修改 `core-site.xml` 。

   i. 执行如下命令打开 `core-site.xml` 。

   ```
   vim /usr/local/hadoop-2.7.3/etc/hadoop/core-site.xml
   ```

   ii. 修改 `core-site.xml` 文件，修改的内容如下所示，其中**实例ID**需要替换为实际实例ID。

   ```
   <configuration>
     <property>
        <name>fs.defaultFS</name>
        <value>hdfs://${实例ID}</value>
     </property>
   </configuration>
   ```

4. 修改 `mapred-site.xml` 配置文件。

   i. 执行如下命令打开 `mapred-site.xml` 。

   ```
   vim /usr/local/hadoop-2.7.3/etc/hadoop/mapred-site.xml
   ```

   ii. 修改 `mapred-site.xml` ，修改的内容如下所示。

   ```
   <configuration>
   <property>
         <name>mapreduce.framework.name</name>
         <value>yarn</value>
   </property>
   </configuration>
   ```

   iii. 修改 `yarn-site.xml` 配置文件。

      a. 执行如下命令打开 `yarn-site.xml` 配置文件。

      ```
      vim /usr/local/hadoop-2.7.3/etc/hadoop/yarn-site.xml
      ```

b. 修改 `yarn-site.xml` ，修改的内容如下所示。

```
<configuration>
<property>
  <name>yarn.resourcemanager.hostname</name>
  <value>xxxx</value>
  <!-- 该地址填写集群中yarn的resourcemanager的hostname -->
</property>
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>spark_shuffle,mapreduce_shuffle</value>
    <!-- 如果不配置spark on yarn,此处只设置为mapreduce_shuffle -->
</property>
<property>
    <name>yarn.nodemanager.aux-services.spark_shuffle.class</name>
    <value>org.apache.spark.network.yarn.YarnShuffleService</value>
    <!-- 如果不配置 spark on yarn,不配置该项 -->
</property>
<property>
  <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
  <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
<property>
  <name>yarn.nodemanager.vmem-pmem-ratio</name>
  <value>2.1</value>
</property>
<property>
  <name>yarn.nodemanager.pmem-check-enabled</name>
  <value>false</value>
</property>
<property>
  <name>yarn.nodemanager.vmem-check-enabled</name>
  <value>false</value>
</property>
<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>16384</value>
    <!-- 根据您当前的集群能力进行配置此项 -->
</property>
<property>
  <name>yarn.nodemanager.resource.cpu-vcores</name>
  <value>4</value>
     <!-- 根据您当前的集群能力进行配置此项 -->
</property>
<property>
  <name>yarn.scheduler.maximum-allocation-vcores</name>
  <value>4</value>
    <!-- 根据您当前的集群能力进行配置此项 -->
</property>
<property>
  <name>yarn.scheduler.minimum-allocation-mb</name>
  <value>3584</value>
    <!-- 根据您当前的集群能力进行配置此项 -->
</property>
<property>
  <name>yarn.scheduler.maximum-allocation-mb</name>
  <value>14336</value>
    <!-- 根据您当前的集群能力进行配置此项 -->
</property>
</configuration>
```

iv. 修改 `slaves` 配置文件。

a. 执行如下命令打开 `slaves` 配置文件。

```
vim /usr/local/hadoop-2.7.3/etc/hadoop/slaves
```

b. 修改 `slaves` 配置文件，修改的内容如下所示，假设spark集群有两个节点node1、node2。

```
node1
node2
```

> 说明
> node1和node2为spark集群节点的机器名。

v. 配置环境变量。

a. 执行如下命令打开 `/etc/profile` 配置文件。

```
vim /etc/profile
```

b. 修改 `/etc/profile` ，在文件末尾添加如下内容。

```
export HADOOP_HOME=/usr/local/hadoop-2.7.3
export HADOOP_CLASSPATH=/usr/local/hadoop-2.7.3/etc/hadoop:/usr/local/hadoop-2.7.3/share/hadoop/
common/lib/*:/usr/local/hadoop-2.7.3/share/hadoop/common/*:/usr/local/hadoop-2.7.3/share/hadoop/
hdfs:/usr/local/hadoop-2.7.3/share/hadoop/hdfs/lib/*:/usr/local/hadoop-2.7.3/share/hadoop/hdfs/*
:/usr/local/hadoop-2.7.3/share/hadoop/yarn/lib/*:/usr/local/hadoop-2.7.3/share/hadoop/yarn/*:/us
r/local/hadoop-2.7.3/share/hadoop/mapreduce/lib/*:/usr/local/hadoop-2.7.3/share/hadoop/mapreduce
/*:/usr/local/hadoop-2.7.3/contrib/capacity-scheduler/*.jar
export HADOOP_CONF_DIR=/usr/local/hadoop-2.7.3/etc/hadoop
```

c. 执行如下命令生效配置信息。

```
source /etc/profile
```

5. 执行如下命令将 `${HADOOP_HOME}` 文件夹同步到集群的其他节点。

```
scp -r hadoop-2.7.2/ root@node2:/usr/local/
```

验证Apache Hadoop配置

完成Hadoop配置后，不需要格式化namenode，也不需要使用 `start-dfs.sh` 来启动HDFS相关服务。如需使用yarn服务，只需在resourcemanager节点启动yarn服务，具体验证Hadoop配置是否成功的方法请参见使用开源HDFS客户端访问。

```
/usr/local/hadoop-2.7.3/sbin/start-yarn.sh
```

配置 Apache Spark

本文档以spark on yarn为例进行搭建说明，spark on yarn的官方配置文档请参见在Yarn上使用Spark。

1. 执行如下命令，解压安装包到指定目录。

```
tar -zxvf spark-2.4.3-bin-hadoop2.7.tgz -C /usr/local/
```

2. 修改 `spark-env.sh` 配置文件。

i. 执行如下命令打开 `spark-env.sh` 配置文件。

```
vim /usr/local/spark-2.4.3-bin-hadoop2.7/conf/spark-env.sh
```

ii. 在 `spark-env.sh` 配置文件中，配置如下信息。

```
export JAVA_HOME=java的安装目录
export SCALA_HOME=scala的安装目录
export SPARK_CONF_DIR=/usr/local/spark-2.4.3-bin-hadoop2.7/conf
export HADOOP_HOME=/usr/local/hadoop-2.7.3
export HADOOP_CONF_DIR=/usr/local/hadoop-2.7.3/etc/hadoop
```

3. 拷贝JAR包。
   将Spark配置文件夹中yarn目录下的 `spark-x.x.x-yarn-shuffle.jar` 包拷贝到当前集群所有节点的 `yarn/lib` 目录下。

> **说明**
> 配置spark on yarn模式时，不需要将Spark配置文件夹分发到集群的所有节点，只需要在集群其中一台提交任务的节点上配置即可。

验证Apache Spark

使用Spark读取文件引擎上的文件，进行wordcount计算，并且将计算结果打印并写入文件引擎。

1. 执行以下命令创建测试数据。

   ```
   echo -e "11,22\n11,22\n11,23\n11,22\n11,22" > /tmp/words
   ```

   创建完成后，执行 `cat /tmp/words` 命令查看测试数据是否创建成功。

2. 执行如下命令，在文件引擎上创建文件夹。

   ```
   /usr/local/hadoop-2.7.3/bin/hadoop fs -mkdir -p /test/input
   /usr/local/hadoop-2.7.3/bin/hadoop fs -mkdir -p /test/output
   ```

3. 执行如下命令将测试数据上传至文件引擎的文件夹上。

   ```
   /usr/local/hadoop-2.7.3/bin/hadoop fs -put /tmp/words /test/input
   ```

   上传完成后，执行 `/usr/local/hadoop-2.7.3/bin/hadoop fs -cat /test/input/words` 命令确认测试数据是否上传成功。

4. 执行以下命令启动spark-shell。

   ```
   /usr/local/spark-2.4.3-bin-hadoop2.7/bin/spark-shell --master yarn \
   --deploy-mode client \
   --driver-cores 1  \
   --driver-memory 1G \
   --executor-memory 1G \
   --num-executors 1 \
   ```

5. 执行以下命令运行wordcount程序。

   ```
   scala> val res = sc.textFile("/test/input/words").flatMap(_.split(",")).map((_,1)).reduceByKey(_+_)
   scala> res.collect.foreach(println)
   scala> res.saveAsTextFile("/test/output/res")
   ```

6. 查看结果。

   ```
   /usr/local/hadoop-2.7.3/bin/hadoop fs -cat /test/output/res/part-00000
   ```

10.4.2. 使用Hive访问

本文介绍如何使用Hive访问文件引擎。

准备工作

- 开通文件引擎，详情请参见开通指南。
- 在计算节点上安装JDK，版本不能低于1.8。

- 下载Apache Derby，下载地址为官网地址，本文中使用的Apache Derby 版本为10.13.1.1。
- 下载Hive压缩包，下载地址为官网地址，本文中使用的Apache Hive版本为2.3.7。

配置 Apache Derby

1. 解压Hive压缩包到指定目录。

```
tar -zxvf db-derby-10.13.1.1-bin.tar.gz -C /usr/local/
```

2. 修改 `/etc/profile` 文件，并配置环境变量。

   i. 执行如下命令打开 `/etc/profile` 配置文件。

   ```
   vim /etc/profile
   ```

   ii. 在文件末尾添加如下内容。

   ```
   export DERBY_HOME=/usr/local/db-derby-10.13.1.1-bin
   export CLASSPATH=$CLASSPATH:$DERBY_HOME/lib/derby.jar:$DERBY_HOME/lib/derbytools.jar
   ```

   iii. 创建存储目录。

   ```
   mkdir $DERBY_HOME/data
   ```

   iv. 启动服务。

   ```
   nohup /usr/local/db-derby-10.13.1.1-bin/bin/startNetworkServer &
   ```

配置 Apache Hive

1. 解压Hive压缩包到指定目录。

```
tar -zxvf apache-hive-2.3.7-bin.tar.gz -C /usr/local/
```

2. 修改 `/etc/profile` 文件，配置环境变量。

   i. 执行如下命令打开 `/etc/profile` 置文件。

   ```
   vim /etc/profile
   ```

   ii. 在文件末尾添加如下内容。

   ```
   export HIVE_HOME=/usr/local/apache-hive-2.3.7-bin
   ```

3. 修改 `hive-env.sh` 文件 。

   i. 执行如下命令打开 `hive-env.sh` 文件。

   ```
   vim /usr/local/apache-hive-2.3.7-bin/hive-env.sh
   ```

   ii. 修改 `hive-env.sh` 文件，修改的内容如下所示。

   ```
   # The heap size of the jvm stared by hive shell script can be controlled via
   export HADOOP_HEAPSIZE=1024

   # Set HADOOP_HOME to point to a specific hadoop install directory
   HADOOP_HOME=/usr/local/hadoop-2.7.3

   # Hive Configuration Directory can be controlled by:
   export HIVE_CONF_DIR=/usr/local/apache-hive-2.3.7-bin/conf

   # Folder containing extra ibraries required for hive compilation/execution can be controlled by:
   export HIVE_AUX_JARS_PATH=/usr/local/apache-hive-2.3.7-bin/lib
   ```

4. 修改 `hive-site.xml` 文件。

   i. 执行如下命令打开 `hive-site.xml` 文件。

   ```
   vim  /usr/local/apache-hive-2.3.7-bin/conf/hive-site.xml
   ```

   ii. 修改 `hive-site.xml` 文件，修改的内容如下所示。

   ```
   <configuration>
    <property>
       <name>hive.metastore.warehouse.dir</name>
       <value>/user/hive/warehouse</value>
       <description>location of default database for the warehouse</description>
    </property>
    <property>
       <name>hive.exec.scratchdir</name>
       <value>/tmp/hive</value>
       <description>HDFS root scratch dir for Hive jobs which gets created with write all (733) permiss
   ion. For each connecting user, an HDFS scratch dir: ${hive.exec.scratchdir}/<username> is created, w
   ith ${hive.scratch.dir.permission}.</description>
     </property>
      <property>
      <name>hive.metastore.schema.verification</name>
      <value>false</value>
       <description>
       Enforce metastore schema version consistency.
       True: Verify that version information stored in metastore matches with one from Hive jars.  Also
   disable automatic
             schema migration attempt. Users are required to manully migrate schema after Hive upgrade
   which ensures
             proper metastore schema migration. (Default)
       False: Warn if the version information stored in metastore doesn't match with one from in Hive j
   ars.
       </description>
     </property>
     <property>
       <name>javax.jdo.option.ConnectionURL</name>
       <value>jdbc:derby://127.0.0.1:1527/metastore_db;create=true </value>
       <description>JDBC connect string for a JDBC metastore </description>
     </property>
     <property>
       <name>datanucleus.schema.autoCreateAll</name>
       <value>true</value>
     </property>
   </configuration>
   ```

5. 创建 `jpox.properties` 文件。

   i. 执行如下命令，打开 `jpox.properties` 文件。

   ```
   vim  /usr/local/apache-hive-2.3.7-bin/conf/jpox.properties
   ```

ii. 修改 `jpox.properties` 文件，修改的内容如下所示。

```
javax.jdo.PersistenceManagerFactoryClass =org.jpox.PersistenceManagerFactoryImpl
org.jpox.autoCreateSchema = false
org.jpox.validateTables = false
org.jpox.validateColumns = false
org.jpox.validateConstraints = false
org.jpox.storeManagerType = rdbms
org.jpox.autoCreateSchema = true
org.jpox.autoStartMechanismMode = checked
org.jpox.transactionIsolation = read_committed
javax.jdo.option.DetachAllOnCommit = true
javax.jdo.option.NontransactionalRead = true
javax.jdo.option.ConnectionDriverName = org.apache.derby.jdbc.ClientDriver
javax.jdo.option.ConnectionURL = jdbc:derby://127.0.0.1:1527/metastore_db;create = true
javax.jdo.option.ConnectionUserName = APP
javax.jdo.option.ConnectionPassword = mine
```

6. 创建必要目录。

```
$HADOOP_HOME/bin/hadoop fs -ls /
如果没有发现上面提到的路径，因此需要新建这些目录，并且给它们赋予用户（W）权限。
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse
$HADOOP_HOME/bin/hadoop fs -ls /tmp/hive
$HADOOP_HOME/bin/hadoop fs -chmod 775  /user/hive/warehouse
$HADOOP_HOME/bin/hadoop fs -chmod 775 /tmp/hive
```

7. 修改 `io.tmpdir` 路径。

   同时要修改 `hive-site.xml` 中所有包含 `${system:java.io.tmpdir}` 字段的value即路径，您可以新建一个目录来替换它，例如 `/tmp/hive/iotmp` 。

```
mkdir /usr/local/apache-hive-2.3.7-bin/iotmp
chmod 777 /usr/local/apache-hive-2.3.7-bin/iotmp
```

   最后再修改以下代码中的 `${system:user.name}` 。

```
<property>
    <name>hive.exec.local.scratchdir</name>
    <value>/usr/local/apache-hive-2.3.7-bin/iotmp/${system:user.name}</value>
    <description>Local scratch space for Hive jobs</description>
</property>
```

   修改如下：

```
<property>
    <name>hive.exec.local.scratchdir</name>
    <value>/usr/local/apache-hive-2.3.7-bin/iotmp/${user.name}</value>
    <description>Local scratch space for Hive jobs</description>
</property>
```

8. 初始化Hive服务。

```
 nohup /usr/local/apache-hive-2.3.7-bin/bin/hive --service metastore &
 nohup /usr/local/apache-hive-2.3.7-bin/bin/hive --service hiveserver2 &
```

验证Apache Hive

1. 在hive Shell中创建一张表。

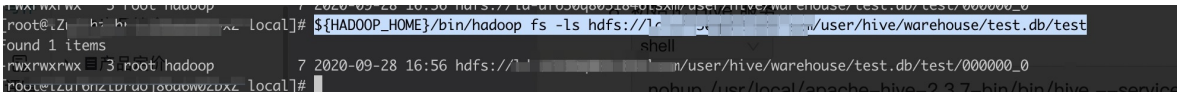
```
create table test (f1 INT, f2 STRING);
```

2. 在表中写入一些数据。

```
insert into test values (1,'2222');
```

3. 查看数据是否已经写入文件引擎。

```
${HADOOP_HOME}/bin/hadoop fs -ls /user/hive/warehouse/test.db/test
```



10.4.3. 使用Flink访问

本文介绍使用自建的Apache Flink访问文件引擎。

准备工作

- 开通文件引擎，详情请参见开通指南 。
- 在计算节点上安装JDK，版本不能低于1.8。
- 在计算节点上安装Scala。
  Scala下载地址为官方链接，其版本要与使用的Apache Spark版本相兼容。
- 下载Apache Hadoop压缩包。
  Apache Hadoop下载地址为官方链接。建议您选用的Apache Hadoop版本不低于2.7.3，本文中使用的Apache Hadoop版本为Apache Hadoop 2.7.3。
- 下载Apache Flink压缩包。
  在Lindorm文件引擎上使用的Flink的版本必须为1.9.0及以上，Apache Flink下载地址为官方链接。本文中使用的Flink版本为官方提供的预编译版本Flink 1.9.0。

> **说明**
> 本文档的操作步骤中涉及的安装包版本号、文件夹路径，请根据实际情况进行替换。

Apache Hadoop 配置

1. 解压压缩包到指定目录。

```
tar -zxvf hadoop-2.7.3.tar.gz -C /usr/local/
```

2. 修改 `hadoop-env.sh` 配置文件。

   i. 执行如下命令打开 `hadoop-env.sh` 配置文件。

   ```
   vim /usr/local/hadoop-2.7.3/etc/hadoop/hadoop-env.sh
   ```

   ii. 配置 `JAVA_HOME` 。

   ```
   export JAVA_HOME=${jdk安装目录}
   ```

3. 修改 `core-site.xml` 。

   i. 执行如下命令打开 `core-site.xml` 。

   ```
   vim /usr/local/hadoop-2.7.3/etc/hadoop/core-site.xml
   ```

   ii. 修改 `core-site.xml` 文件，修改的内容如下所示，其中**实例ID**需要替换为实际实例ID。

   ```
   <configuration>
     <property>
        <name>fs.defaultFS</name>
        <value>hdfs://${实例ID}</value>
     </property>
   </configuration>
   ```

4. 修改 `mapred-site.xml` 配置文件。
   i. 执行如下命令打开 `mapred-site.xml` 。

   ```
   vim /usr/local/hadoop-2.7.3/etc/hadoop/mapred-site.xml
   ```

   ii. 修改 `mapred-site.xml` ，修改的内容如下所示。

   ```
   <configuration>
   <property>
         <name>mapreduce.framework.name</name>
         <value>yarn</value>
   </property>
   </configuration>
   ```

5. 修改 `yarn-site.xml` 配置文件。
   i. 执行如下命令打开 `yarn-site.xml` 配置文件。

   ```
   vim /usr/local/hadoop-2.7.3/etc/hadoop/yarn-site.xml
   ```

ii. 修改 `yarn-site.xml` ，修改的内容如下所示。

```
<configuration>
<property>
  <name>yarn.resourcemanager.hostname</name>
  <value>xxxx</value>
  <!-- 该地址填写集群中yarn的resourcemanager的hostname -->
</property>
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>
<property>
  <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
  <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>16384</value>
    <!-- 根据您当前的集群能力进行配置此项 -->
</property>
<property>
  <name>yarn.nodemanager.resource.cpu-vcores</name>
  <value>4</value>
     <!-- 根据您当前的集群能力进行配置此项 -->
</property>
<property>
  <name>yarn.scheduler.maximum-allocation-vcores</name>
  <value>4</value>
    <!-- 根据您当前的集群能力进行配置此项 -->
</property>
<property>
  <name>yarn.scheduler.minimum-allocation-mb</name>
  <value>3584</value>
    <!-- 根据您当前的集群能力进行配置此项 -->
</property>
<property>
  <name>yarn.scheduler.maximum-allocation-mb</name>
  <value>14336</value>
    <!-- 根据您当前的集群能力进行配置此项 -->
</property>
</configuration>
```

6. 修改 `slaves` 配置文件。

i. 执行如下命令打开 `slaves` 配置文件。

```
vim /usr/local/hadoop-2.7.3/etc/hadoop/slaves
```

ii. 修改 `slaves` ，修改的内容如下所示，假设spark集群有两个节点node1，node2。

```
node1
node2
```

> **说明**
> node1和node2为spark集群节点的机器名。

7. 配置环境变量。

i. 执行如下命令打开 `/etc/profile` 配置文件。

```
vim /etc/profile
```

ii. 修改 `/etc/profile`，在文件末尾添加如下内容。

```
export HADOOP_HOME=/usr/local/hadoop-2.7.3
export HADOOP_CLASSPATH=/usr/local/hadoop-2.7.3/etc/hadoop:/usr/local/hadoop-2.7.3/share/hadoop/common/lib/*:/usr/local/hadoop-2.7.3/share/hadoop/common/*:/usr/local/hadoop-2.7.3/share/hadoop/hdfs:/usr/local/hadoop-2.7.3/share/hadoop/hdfs/lib/*:/usr/local/hadoop-2.7.3/share/hadoop/hdfs/*:/usr/local/hadoop-2.7.3/share/hadoop/yarn/lib/*:/usr/local/hadoop-2.7.3/share/hadoop/yarn/*:/usr/local/hadoop-2.7.3/share/hadoop/mapreduce/lib/*:/usr/local/hadoop-2.7.3/share/hadoop/mapreduce/*:/usr/local/hadoop-2.7.3/contrib/capacity-scheduler/*.jar
export HADOOP_CONF_DIR=/usr/local/hadoop-2.7.3/etc/hadoop
```

iii. 执行如下命令生效配置信息。

8. 执行如下命令将 `${HADOOP_HOME}` 文件夹同步到集群的其他节点。

```
scp -r hadoop-2.7.2/ root@node2:/usr/local/
```

验证Apache Hadoop配置

完成Hadoop配置后，不需要格式化namenode，也不需要使用start-dfs.sh来启动HDFS相关服务。如需使用yarn服务，只需在resourcemanager节点启动yarn服务，具体验证Hadoop是否配置成功的方法请参见使用开源HDFS客户端访问。

配置Apache Flink

执行如下命令解压Flink压缩包到指定目录。

```
tar -zxvf flink-1.9.0-bin-scala_2.11.tgz -C /usr/local/
```

> 说明
>
> - 在使用Apache Flink之前必须在您的集群环境变量中配置 `HADOOP_HOME`，`HADOOP_CLASSPATH` 和 `HADOOP_CONF_DIR`，详情请参见Apache Hadoop 配置步骤7配置环境变量。
> - 如果您需要对Flink进行额外的配置，请参见官方文档配置操作指南。

验证Apache Flink配置

使用Flink自带的 `WordCount.jar` 对Lindorm文件引擎上的数据进行读取，并将计算结果写入Lindorm文件引擎，在测试之前需要先启动yarn服务。

1. 生成测试数据。

   此处使用Apache Hadoop 2.7.3自带的JAR包 `hadoop-mapreduce-examples-2.7.3.jar` 中的randomtextwriter方法在文件引擎上生成测试数据。

   ```
   /usr/local/hadoop-2.7.3/bin/hadoop jar /usr/local/hadoop-2.7.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar randomtextwriter \
   -D mapreduce.randomtextwriter.totalbytes=10240 \
   -D mapreduce.randomtextwriter.bytespermap=1024 \
   -D mapreduce.job.maps=4  \
   -D mapreduce.job.reduces=2  \
   /flink-test/input \
   ```

2. 查看在文件引擎上生成的测试数据。

   ```
   /usr/local/hadoop-2.7.3/bin/hadoop fs -cat /flink-test/input/*
   ```

3. 提交wordcount程序。

```
/usr/local/flink-1.9.0/bin/flink run -m yarn-cluster -yn 1 -yjm 1024 -ytm 1024 \
/usr/local/flink-1.9.0/examples/batch/WordCount.jar \
--input hdfs://ld-uf630q8031846lsxm/flink-test/input \
--output hdfs://ld-uf630q8031846lsxm/flink-test/output \
```

4. 查看在Lindorm文件引擎上的结果文件。

```
/usr/local/hadoop-2.7.3/bin/hadoop fs -cat /flink-test/output
```

```
[root@ ~]# /usr/local/hadoop-2.7.3/bin/hadoop fs -cat /flink-test/output
1 1
1xi 1
2 3
3s 1
43endomycetaceae 1
4sedentariness 1
4y 1
5chilblain 1
5prepavement 1
7 1
76babylonism 1
9y 1
_ 2
_trillium 1
a 2
abscission 1
absvolt 3
abthainry 3
abusiveness 4
acocotl 2
adatom 3
adscendent 1
```

10.4.4. 使用Presto访问

本章节介绍如何通过自建的Presto使用文件引擎。

背景信息

Presto是一个开源的分布式SQL查询引擎，适用于交互式分析查询，数据量支持从GB到PB字节。Presto支持在线数据查询，包括Hive、Cassandra、关系数据库以及专有数据存储。

> **说明**
> 本文中Presto是通过连接Hive的元数据服务来读取文件存储HDFS上的数据，在文件引擎上使用Presto时需要额外配置一些依赖包，详细操作步骤请参见配置Presto。

准备工作

搭建和使用Presto读写文件引擎，需要先完成以下准备工作。

1. 开通文件引擎，详情请参见 开通指南 。
2. 在计算节点上安装JDK，版本不能低于1.8。
3. 在Hadoop集群中安装Apache Hive，详情请参见 使用Hive访问 。
4. 下载Presto压缩包和presto-cli-xxx-executable.jar。
   Presto下载地址：官方链接，在本文档使用Presto的版本为0.241。

配置Presto

您可以参见以下步骤配置Presto，Presto官方配置文档请参见Deploying Presto。

1. 解压Presto压缩包到指定目录。

```
tar -zxvf  presto-server-0.241.tar.gz -C /usr/local/
```

2. 在Presto解压目录下创建etc目录。

```
mkdir /usr/local/presto-server-0.241/etc
```

3. 配置Node Properties。
   i. 创建 `etc/node.properties` 文件。

   ```
   vim /usr/local/presto-server-0.241/etc/node.properties
   ```

   ii. 在 `etc/node.properties` 文件中添加如下内容。

   ```
   node.environment=test
   node.id=ffffffff-ffff-ffff-ffff-ffffffffffff
   node.data-dir=/tmp/presto/data
   ```

4. 配置JVM Config。
   i. 创建 `etc/jvm.config` 文件。

   ```
   vim /usr/local/presto-server-0.241/etc/jvm.config
   ```

   ii. 在 `etc/jvm.config` 文件中添加如下内容。

   ```
   -server
   -Xmx8G
   -XX:+UseG1GC
   -XX:G1HeapRegionSize=32M
   -XX:+UseGCOverheadLimit
   -XX:+ExplicitGCInvokesConcurrent
   -XX:+HeapDumpOnOutOfMemoryError
   -XX:+ExitOnOutOfMemoryError
   ```

5. 配置Config Properties。
   本文将coordinator和worker配置在同一台机器上，您可以参见Presto官方文档将coordinator和worker配置到不同的机器。
   i. 创建 `etc/config.properties` 文件。

   ```
   vim /usr/local/presto-server-0.241/etc/config.properties
   ```

   ii. 在 `etc/config.properties` 中添加如下内容。

   ```
   coordinator=true
   node-scheduler.include-coordinator=true
   http-server.http.port=8080
   query.max-memory=5GB
   query.max-memory-per-node=1GB
   query.max-total-memory-per-node=2GB
   discovery-server.enabled=true
   discovery.uri=http://xx.xx.xx.xx:8080 #xx.xx.xx.xx为当前机器的IP地址
   ```

6. 配置日志级别。
   i. 创建 `etc/log.properties` 文件。

   ```
   vim /usr/local/presto-server-0.241/etc/log.properties
   ```

   ii. 在 `etc/log.properties` 文件中添加如下内容。

   ```
   com.facebook.presto=INFO
   ```

7. 配置Catalog Properties。
   i. 创建 `etc/catalog` 文件夹。

   ```
   mkdir /usr/local/presto-server-0.241/etc/catalog
   ```

ii. 创建 `etc/catalog/hive.properties` 文件。

```
vim /usr/local/presto-server-0.241/etc/catalog/hive.properties
```

iii. 在 `etc/catalog/hive.properties` 文件中添加如下内容。

```
connector.name=hive
hive.metastore.uri=thrift://xxxx:9083 #xxxx为启动hive元数据服务的IP地址
hive.config.resources=/usr/local/hadoop-2.7.3/etc/hadoop/core-site.xml,/usr/local/hadoop-2.7.3/etc/hadoop/hdfs-site.xml #配置为您的Hadoop集群中core-site.xml文件的地址
```

8. 将 `presto-cli-xxx-executable.jar` 复制到Presto安装的bin目录下，重命名并赋予可执行权限。

```
cp ~/presto-cli-0.241-executable.jar  /usr/local/presto-server-0.241/bin/
```

```
mv /usr/local/presto-server-0.241/bin/presto-cli-0.241-executable.jar  /usr/local/presto-server-0.241/bin/presto
```

```
chmod +x /usr/local/presto-server-0.241/bin/presto
```

验证Presto

1. 创建测试数据并加载到Hive中。
   i. 创建测试数据。

```
echo -e "tt1\ntt2\ntt1\ntt2\ntt3\ntt4\ntt4\ntt5\ntt6" > ~/test.txt
```

   ii. 将测试数据上传到文件引擎上。

```
$HADOOP_HOME/bin/hadoop fs -mkdir /presto
$HADOOP_HOME/bin/hadoop fs -put test.txt /presto/
```

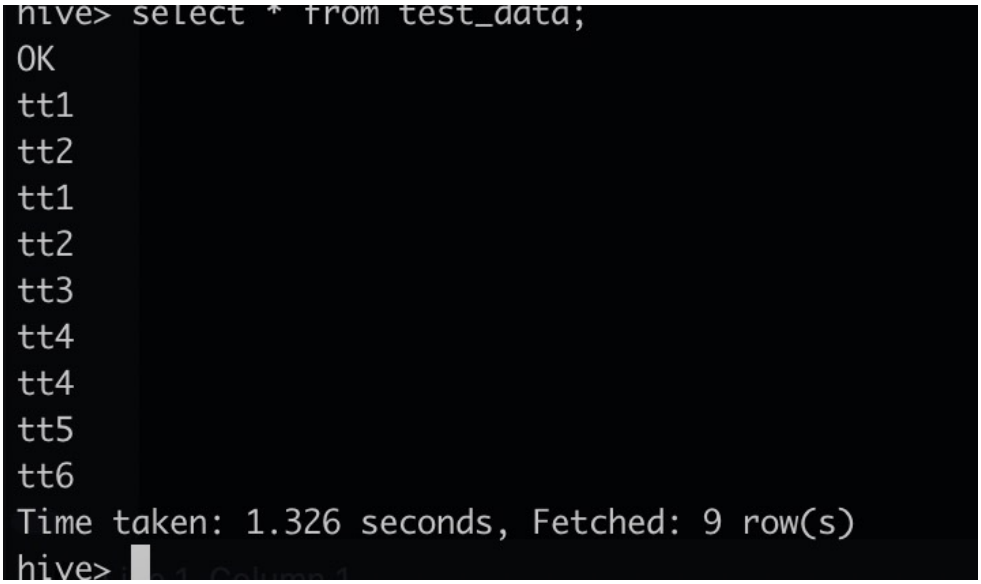
   iii. 使用默认的数据创建test_data并加载数据。

```
hive> create external table test_data(word string) row format delimited fields terminated by '\n' stored as textfile location '/presto';
```

   iv. 查看数据是否加载成功。

```
hive> select * from test_data;
```

   如果显示如下类似信息，则表示数据加载成功。



2. 使用Presto通过Hive读取文件引擎上的数据并进行计算。
   i. 启动presto server。

```
/usr/local/presto-server-0.241/bin/launcher start
```

   ii. 使用presto连接Hive。
      a. 使用presto连接Hive。

```
/usr/local/presto-server-0.241/bin/presto --server localhost:8080 --catalog hive --schema defaul
t
```

      b. 读取文件引擎上的数据。

```
presto:default> select * from test_data;
```

```
presto:default> select * from test_data;
 word
------
 tt1
 tt2
 tt1
 tt2
 tt3
 tt4
 tt4
 tt5
 tt6
(9 rows)

Query 20200928_102616_00001_5auvm, FINISHED, 1 node
Splits: 17 total, 17 done (100.00%)
0:03 [9 rows, 36B] [3 rows/s, 13B/s]
```

      c. 进行word count计算。

```
presto:default> select word, count(*) from test_data group by word;
```

```
presto:default> select word, count(*) from test_data group by word;
 word | _col1
------+-------
 tt6  |     1
 tt5  |     1
 tt2  |     2
 tt4  |     2
 tt1  |     2
 tt3  |     1
(6 rows)

Query 20200928_102728_00002_5auvm, FINISHED, 1 node
Splits: 49 total, 49 done (100.00%)
0:01 [9 rows, 36B] [8 rows/s, 34B/s]
```

10.4.5. 使用DLA访问

本文介绍通过数据湖分析（DLA）连接云原生多模数据库Lindorm文件引擎的方法，并使用SQL查询文件引擎中的数据。

前提条件

- 云原生多模数据库Lindorm实例已开通文件引擎，具体请参见开通指南。
- 将专有网络的IP添加至Lindorm实例的白名单中，具体请参见设置白名单。
- 已开通DLA Serverless SQL CU版本，具体请参见DLA Presto CU版本快速入门。

> **说明**
> 配置DLA虚拟集群的数据源网络时，数据源网络和Lindorm实例使用相同的专有网络ID。

访问方式

数据湖分析（Data Lake Analytics，简称DLA）支持通过DLA Serverless SQL和DLA Serverless Spark方式访问云原生多模数据库Lindorm文件引擎。

- 通过DLA Serverless Spark方式访问Lindorm文件引擎，具体请参见Hadoop。
- 通过DLA Serverless SQL方式访问Lindorm文件引擎的具体步骤请参见操作步骤。

> **说明**
> DLA Serverless SQL仅支持通过CU版本访问Lindorm文件引擎。

准备工作

1. 在本地创建新的文本文件 `example.txt` ，文本内容示例如下：

   ```
   7,8,9
   ```

2. 执行如下命令将文本文件上传至Lindorm文件引擎。

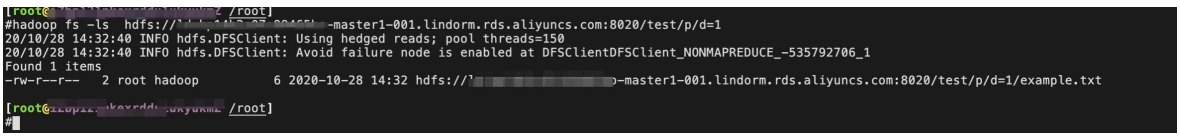
   ```
   hadoop fs -mkdir -p hdfs://${实例ID}-master1-001.lindorm.rds.aliyuncs.com:8020/test/p/d=1
   hadoop fs -copyFromLocal example.txt hdfs://${实例ID}-master1-001.lindorm.rds.aliyuncs.com:8020/test/p/d=1/example.txt
   ```

   > **说明**
   > 由于DLA Serverless SQL访问不支持高可用方式，所以通过该方式访问文件引擎时，需要将上面的地址替换成文件引擎主名称节点的host，主备节点可以在hdfs-site配置中查看，查看方法请参见访问配置生成。

3. 执行如下命令查看数据是否上传成功。

   ```
   hadoop fs -ls hdfs://${实例ID}-master1-001.lindorm.rds.aliyuncs.com:8020/test/p/d=1
   ```

   执行结果如下图表示上传成功。

操作步骤

1. 登录DLA控制台。
2. 在左侧导航栏，选择**Serverless Presto>SQL执行**。
3. 在编辑框中输入以下SQL命令，并单击编辑框上方的**同步执行**。

```
CREATE DATABASE `my_lindorm_store_db`
WITH DBPROPERTIES (
    catalog = 'hive',
    location = 'hdfs://${实例ID}-master1-001.lindorm.rds.aliyuncs.com:8020/test/'
)
```

> **说明**
> 您也可以通过MySQL客户端或者程序代码等方式连接DLA，然后执行SQL命令创建库。创建库语句的参数说明如下：
> - CATALOG：取值为hive，表示创建的是hive Schema。
> - LOCATION：库所在的目录。

4. 执行以下创建表的命令。

```
USE `my_lindorm_store_db`;
CREATE EXTERNAL TABLE p (
    `a` int,
    `b` int,
    `c` int
) partitioned by (d int)
ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
STORED AS `TEXTFILE`
LOCATION 'hdfs://${实例ID}-master1-001.lindorm.rds.aliyuncs.com:8020/test/p/';
```

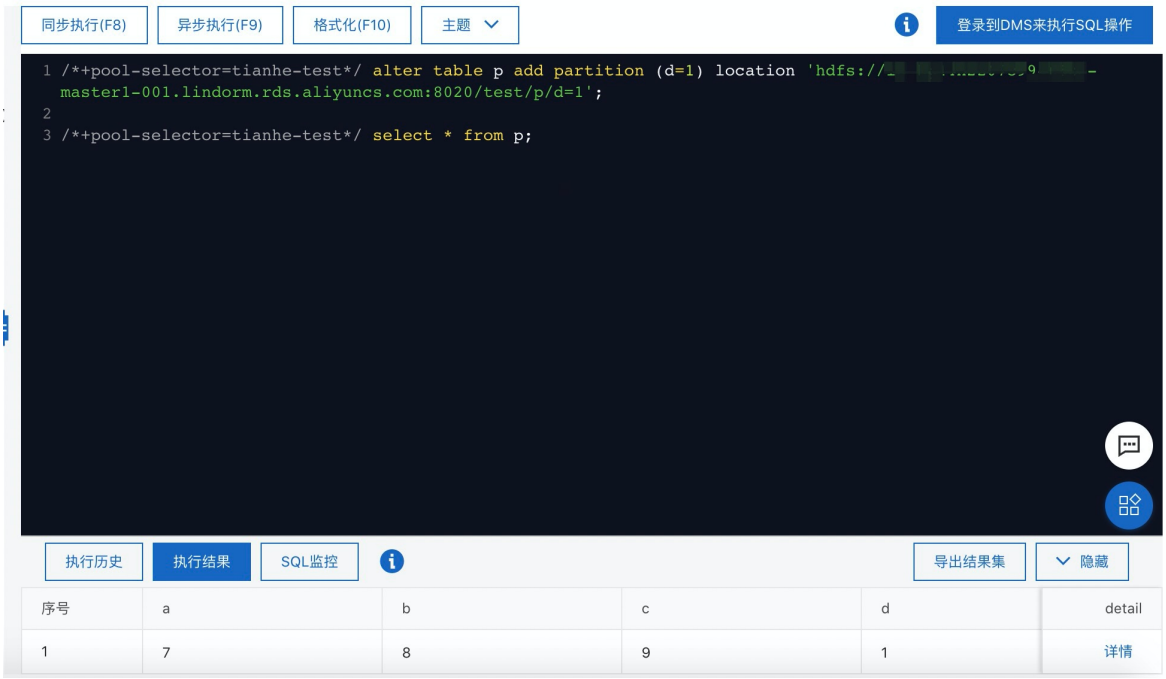
> **说明**
> 文件引擎表的参数、表属性和OSS表大部分都是一样的。主要的区别如下：
> - 由于网络连通性的问题，在创建库、表时不会对目录内容进行检查，用户需要自己保证目录的正确性。
> - 基于同样的原因，文件引擎表不支持auto.create.location属性。

5. 查询数据。由于只有CU的计算资源与Lindorm文件引擎网络可以连通，因此所有访问文件引擎表的SQL语句都需要指定 `hint： /*+pool-selector=your-vc-name*/` ，指定后SQL语句会在CU中执行。

```
 /*+ pool-selector=test-vc */
alter table p
add partition (d=1)
location 'hdfs://${实例ID}-master1-001.lindorm.rds.aliyuncs.com:8020/test/p/d=1';
Query OK, 0 rows affected (8.63 sec)

/*+ pool-selector=test-vc */ select * from  p;
+------+------+------+------+
| a    | b    | c    | d    |
+------+------+------+------+
|    7 |    8 |    9 |    1 |
+------+------+------+------+
1 row in set (4.74 sec)
```

执行结果如下图：



> **说明**
> DLA中常用的SQL类型和语法，更多请参见常用SQL。

10.4.6. 使用E-Mapreduce访问

本文介绍如何通过E-Mapreduce访问文件引擎。

背景信息

阿里云E-MapReduce是构建在阿里云云服务器ECS上的开源Hadoop、Spark、Hive、Flink生态大数据PaaS产品。提供用户在云上使用开源技术建设数据仓库、离线批处理、在线流式处理、即时查询、机器学习等场景下的大数据解决方案。

E-MapReduce数据迁移

1. 准备工作。
   i. 开通并创建E-MapRedece集群，详情请参见创建集群。

   > **说明**
   > 当使用存储引擎替换E-MapReduce HDFS服务时，您可以选择使用高效云盘、SSD云盘或者本地盘作为Shuffle数据的临时本地存储。具体存储规划可以参见存储说明。

ii. 数据迁移。

a. 登录阿里云E-MapReduce控制台。

b. 在主机列表界面，选择任一的一台ECS，单击进入**云服务器管理控制台**界面，在**基础信息**中单击**远程连接**，进入控制台。

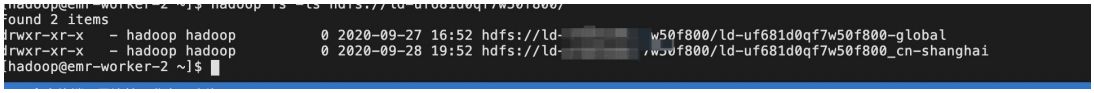
c. 在服务配置中，单击**hdfs-site**，查找dfs.nameservices，将文件引擎的namespace添加到后面。

d. 单击**自定义配置**添加如下的配置，配置信息在文件引擎**一键生成配置信息**中可以找到。

e. 更改配置。选择集群服务 > HDFS，单击部**署客户端配置**。

f. 验证文件引擎是否可以正常访问。
在emr的一个节点上执行如下命令，实例ID为文件引擎的实例ID，根据实际情况修改。

```
hadoop fs -ls hdfs://${实例ID}/
```



g. 数据迁移。
建议将/user、/hbase、/spark-history、/apps等服务目录和相关的数据目录全量迁移至文件引擎。

a. E-MapReduce HDFS文件系统上的数据，需要使用数据迁移工具hadoop distcp进行数据迁移，详情请参见从自建HDFS迁移数据 。

b. 执行下面的指令将E-MapReduce HDFS全部数据迁移到文件引擎，实例ID为文件引擎的实例ID，根据实际情况修改。

```
hadoop distcp /apps hdfs://${实例ID}/
hadoop distcp /emr-flow hdfs://${实例ID}/
hadoop distcp /emr-sparksql-udf hdfs://${实例ID}/
hadoop distcp /hbase hdfs://${实例ID}/
hadoop distcp /spark-history hdfs://${实例ID}/
hadoop distcp /tmp hdfs://${实例ID}/
hadoop distcp /user hdfs://${实例ID}/
```

修改目录权限。

```
hadoop fs -chmod 777  hdfs://${实例ID}/tmp
```

c. 验证数据迁移情况。
执行如下执行，查看数据迁移情况，实例ID为文件引擎的实例ID，根据实际情况修改。

```
hadoop fs -ls hdfs://${实例ID}/
```

如果查询结果如下图，表示迁移成功。



2. 开通文件引擎服务，详情请参见开通指南。

> **说明**
> 文件引擎的专有网络和交换机必须与E-MapReduce集群侧的配置保持一致。您可以通过以下方法获取：
>
> i. 登录阿里云E-MapReduce控制台。
>
> ii. 单击**集群基础信息**，在网络信息区域中获取专有网络和交换机信息。

配置E-MapReduce服务使用文件引擎

1. 配置HDFS服务。

i. 登录阿里云E-MapReduce控制台。
ii. 在**集群管理**页面，找到需要挂载文件存储HDFS的目标E-MapReduce集群，单击**管理**。
iii. 更改配置。
   a. 选择**集群服务** > **HDFS**，单击**配置**。
   b. 在**服务配置**中，单击**core-site**。
   c. 找到配置项**fs.defaultFS**，将其替换为文件引擎的地址。
   d. 单击**保存**，在**确认保存**对话框中，输入执行原因，单击**确定**。
   e. 单击**部署客户端配置**，在**确认保存**对话框中，输入执行原因，单击**确定**。
iv. 重启YARN服务。
   a. 选择**集群服务** > **YARN**。
   b. 在页面右侧的操作栏中，单击**重启All Components**。

2. 配置HIVE服务。

> **说明**
> - 配置HDFS服务完成后，才能配置Hive服务。
> - 在配置Hive服务之前，请确认 `/user/hive/` 目录中的数据已完成全量迁移，迁移方法请参见从自建HDFS迁移数据。

i. 更改配置。
   a. 选择**集群管理** > **HIVE**，单击**配置**。
   b. 在**服务配置**中，单击**hive-site**。
   c. 找到配置项 `hive.metastore.warehouse.dir` ，删除其对应值中的 `E-MapReduce HDFS` 文件系统域名，只保留 `/user/hive/warehouse` 。



   d. 单击**保存**，在**确认保存**对话框中，输入执行原因，单击**确定**。
   e. 单击**部署客户端配置**，在**确认保存**对话框中，输入执行原因，单击**确定**。

ii. 修改元数据。

a. 进入ERM hiveMeta所在的ECS终端，用 `cd $HIVE_CONF_DIR` 进入hive的配置目录，在 `hivemetastore-site.xml` 中，获取数据库相关信息。

a. 在配置项 `javax.jdo.option.ConnectionURL` 中，获取MySQL服务的主机名和元数据存储的数据库。

b. 在配置项 `javax.jdo.option.ConnectionUserName` 中，获取MySQL服务的用户名。

c. 在配置项 `javax.jdo.option.ConnectionPassword` 中，获取MySQL服务的用户密码。

b. Hive的元数据存储在MySQL中，进入存储Hive元数据的MySQL数据库hivemeta中，修改DBS表和SDS表，如下所示。

```
MariaDB [(none)]> use hivemeta;
##修改表“DBS”

MariaDB [hivemeta]> select * from DBS;
+-------+-----------------------+-------------------------------------------------------------------
------------------+------------------+------------+------------+
| DB_ID | DESC                  | DB_LOCATION_URI
| NAME              | OWNER_NAME | OWNER_TYPE |
+-------+-----------------------+-------------------------------------------------------------------
------------------+------------------+------------+------------+
|     1 | Default Hive database | hdfs://emr-header-1.cluster-190507:9000/user/hive/warehouse
| default           | public     | ROLE       |
|     2 | NULL                  | hdfs://emr-header-1.cluster-190507:9000/user/hive/warehouse/em
r_presto_init__.db | emr_presto_init__ | has        | USER       |
+-------+-----------------------+-------------------------------------------------------------------
------------------+------------------+------------+------------+
2 rows in set (0.00 sec)

MariaDB [hivemeta]> UPDATE DBS
    -> SET    DB_LOCATION_URI = 'hdfs://ld-uf681d0qf7w50f800/user/hive/warehouse'
    -> WHERE  DB_ID = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
MariaDB [hivemeta]> UPDATE DBS
    -> SET    DB_LOCATION_URI = 'hdfs://ld-uf681d0qf7w50f800/user/hive/warehouse/emr_presto_init
__.db'
    -> WHERE  DB_ID = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
##修改表“SDS” 由于是新的集群，所以该表没有数据
MariaDB [hivemeta]> select * from  SDS ;
Empty set (0.00 sec)
```

c. 在页面右侧的操作栏，单击**重启All Components**，重启服务。

3. 配置Spark服务。

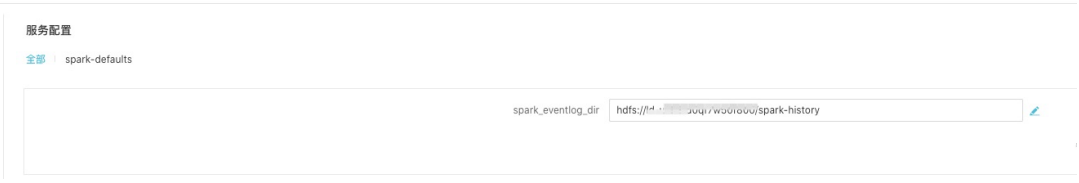
> **说明**
>
> - 配置HDFS服务完成后，才能配置Spark服务。
> - 配置Spark服务前，请确认 `/spark-history` 目录中的数据已经完成了全量迁移。迁移方法请参见从自建HDFS迁移数据。

i. 更改配置。

a. 选择**集群服务** > **Spark**，单击**配置**。

b. 在**服务配置**中，单击**spark-defaults**。

c. 找到配置项 `spark_eventlog_dir` ，将其对应的值替换为文件引擎。



d. 单击**保存**，在**确认保存**对话框中，输入执行原因，单击**确定**。

e. 单击**部署客户端配置**，在**确认保存**对话框中，输入执行原因，单击**确定**。

ii. 在页面右侧的操作栏，单击**重启 All Components**，重启服务。

4. 配置HBase服务。

> **说明**
> - 配置HDFS服务完成后，才能配置HBase服务。
> - 配置HBase服务前，请确认 `/hbase` 目录中的数据已经完成了全量迁移。迁移方法请参见从自建HDFS迁移数据。

i. 更改配置。

a. 选择**集群服务** > **Hbase**，单击配置。

b. 在**服务配置**中，单击**Hbase-site**。

c. 找到配置项 `hbase.rootdir` ，删除其对应值中的 `E-MapReduce HDFS` 文件系统域名，只保留 `/hbase` 。



d. 单击**保存**，在**确认保存**对话框中，输入执行原因，单击**确定**。

e. 单击**部署客户端配置**，在**确认保存**对话框中，输入执行原因，单击**确定**。

ii. 在页面右侧的操作栏，单击**重启All Components**，重启服务。

5. 关闭HDFS服务。

> **说明**
> 关闭HDFS服务前，请确认原来E-MapReduce HDFS上存储的数据都已经迁移到文件引擎。迁移方法请参见从自建HDFS迁移数据。

i. 选择**集群服务** > **HDFS**。

ii. 在页面右侧的操作栏，单击**停止All Components**，关闭服务。

iii. 在**执行集群操作**对话框中，输入执行原因，单击**确定**。

6. 验证服务正确性。

i. hadoop的验证。 使用E-MapReduce hadoop中自带的测试包 `hadoop-mapreduce-examples-2.x.x.jar` 进行测试。该测试包默认放置在 `/opt/apps/ecm/service/hadoop/2.x.x-1.x.x/package/hadoop-2.x.x-1.x.x/share/hadoop/mapreduce/` 目录下。

a. 执行以下命令，在 `/tmp/randomtextwriter` 目录下生成128 M大小的文件。

```
hadoop jar /opt/apps/ecm/service/hadoop/2.8.5-1.5.3/package/hadoop-2.8.5-1.5.3/share/hadoop/mapr
educe/hadoop-mapreduce-examples-2.8.5.jar randomtextwriter  -D mapred
uce.randomtextwriter.totalbytes=134217728  -D mapreduce.job.maps=2 -D mapreduce.job.reduces=2
/tmp/randomtextwriter
```

其中 `hadoop-mapreduce-examples-2.8.5.jar` 为E-MapReduce的测试包，请根据实际情况修改。

b. 执行以下命令验证文件是否生成成功，从而验证文件系统实例的连通性。

```
hadoop fs -ls hdfs://${实例ID}/tmp/randomtextwriter
```

其中${实例ID}为您的文件引擎的实例ID，请根据实际情况修改。

7. Spark的验证。

i. 使用E-MapReduce Spark中自带的测试包 `spark-examples_2.x-2.x.x.jar` 进行测试。该测试包默认放置在 `/opt/apps/ecm/service/spark/2.x.x-1.0.0/package/spark-2.x.x-1.0.0-bin-hadoop2.8/examples/jars` 下。

a. 执行以下命令，在 `/tmp/randomtextwriter` 目录下生成128 M大小的文件。

```
hadoop jar /opt/apps/ecm/service/hadoop/2.8.5-1.5.3/package/hadoop-2.8.5-1.5.3/share/hadoop/mapr
educe/hadoop-mapreduce-examples-2.8.5.jar randomtextwriter  -D mapred
uce.randomtextwriter.totalbytes=134217728  -D mapreduce.job.maps=2 -D mapreduce.job.reduces=2
/tmp/randomtextwriter
```

其中 `hadoop-mapreduce-examples-2.8.5.jar` 为E-MapReduce的测试包，请根据实际情况修改。

b. 使用Spark测试包从文件存储HDFS上读取测试文件并按照word count的格式展示。

```
spark-submit   --master yarn --executor-memory 2G --executor-cores 2  --class org.apache.spark.e
xamples.JavaWordCount /opt/apps/ecm/service/spark/2.4.5-hadoop2.8-1.1.0/package/spark-2.4.5-hado
op2.8-1.1.0/examples/jars/spark-examples_2.11-2.4.5.jar  /tmp/randomtextwriter
```

如果回显信息类似如下图所示，表示配置成功。



8. Hive的验证。

i. 执行以下命令进入Hive命令界面。

```
[hadoop@emr-worker-2 ~]$ hive

Logging initialized using configuration in file:/etc/ecm/hive-conf-2.3.5-1.2.0/hive-log4j2.properties Async: true
Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
hive>
```

ii. 执行以下命令创建测试表。

```
hive> create table default.tt(id int , name string )  row format delimited   fields terminated by '\t'  lines terminated by '\n';
OK
Time taken: 2.058 seconds
```

iii. 执行以下命令查看测试表。
如果回显信息中的Location属性对应的值为文件引擎的路径，则表示配置Hive成功。

```
ive>  desc formatted  default.tt ;
OK
# col_name              data_type               comment

id                      int
name                    string

# Detailed Table Information
Database:               default
Owner:                  hadoop
CreateTime:             Mon Sep 28 21:26:14 CST 2020
LastAccessTime:         UNKNOWN
Retention:              0
Location:               hdfs://ld-uf681d0qf7w50f800/user/hive/warehouse/tt
Table Type:             MANAGED_TABLE
Table Parameters:
        COLUMN_STATS_ACCURATE   {\"BASIC_STATS\":\"true\"}
        numFiles                0
        numRows                 0
        rawDataSize             0
        totalSize               0
        transient_lastDdlTime   1601299574

# Storage Information
SerDe Library:          org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
InputFormat:            org.apache.hadoop.mapred.TextInputFormat
OutputFormat:           org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
Compressed:             No
Num Buckets:            -1
Bucket Columns:         []
Sort Columns:           []
Storage Desc Params:
        field.delim             \t
        line.delim              \n
        serialization.format    \t
Time taken: 0.507 seconds, Fetched: 33 row(s)
```

9. HBase的验证。

i. 执行以下命令进入hbase Shell命令界面。

```
[hadoop@emr-worker-2 ~]$ hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/apps/ecm/service/hbase/1.4.9-1.0.0/package/hbase-1.4.9-1.0.0/
lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/apps/ecm/service/hadoop/2.8.5-1.5.3/package/hadoop-2.8.5-1.5.
3/share/hadoop/common/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.clas
s]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
Version 1.4.9, r8214a16c5d80f077abf1aa01bb312851511a2b15, Thu Jan 31 20:35:22 CST 2019

hbase(main):001:0>
```

ii. 在HBase中创建测试表。

```
hbase(main):001:0> create 'hbase_test','info'
0 row(s) in 1.6700 seconds

=> Hbase::Table - hbase_test
hbase(main):002:0> put 'hbase_test','1', 'info:name' ,'Sariel'
0 row(s) in 0.0960 seconds

hbase(main):003:0>  put 'hbase_test','1', 'info:aa' ,'33'
0 row(s) in 0.0110 seconds

hbase(main):004:0>
```

iii. 执行以下命令查看文件引擎的 `/hbase/data/default/` 路径，如果 `/hbase/data/default/` 路径下有 `hbase_test` 目录，则证明配置链接成功。

```
hadoop fs -ls /hbase/data/default
```

```
[root@emr-header-1 hadoop-hbase]# hadoop fs -ls /hbase/data/default
Found 1 items
drwxr-x--x   - hbase hadoop          0 2020-09-28 21:57 /hbase/data/default/hbase_test
[root@emr-header-1 hadoop-hbase]#
```

卸载并释放E-MapReduce HDFS使用的云盘

当E-MapReduce已经成功运行在文件引擎上时，ECS挂载的云盘只用来存储运算中的临时Shuffle文件，可以选择卸载原来用于构建E-MapReduce HDFS服务的云盘，降低集群的拥有成本。具体操作如下：

1. 卸载数据盘，详情请参见卸载数据盘。
2. 释放云盘，详情请参见释放云盘。

10.4.7. 使用MaxCompute访问

本章节介绍如何使用DataWorks实现MaxCompute与文件引擎之间的双向数据同步。您可以将MaxCompute数据同步至文件引擎，也可以将文件引擎的数据同步至MaxCompute。

> **说明** 请确保Maxcompute、文件引擎、Dataworks在同一个地域（region）下。

准备工作

1. 开通文件引擎，详情请参见开通指南。
2. 开通MaxCompute，详情请参见开通MaxCompute。
3. 开通DataWorks，详情请参见DataWorks 入门概述。

配置步骤

1. 配置DataWorks独享数据集成资源组。开通配置及使用步骤请参见购买独享资源，并且注意以下几方面配置。
   - 配置的独享数据集成资源组，需要与文件引擎在同一region的同一可用区（AZ）下。
   - 如果文件引擎相对应的AZ下的DataWorks无资源，或者文件引擎与独享数据集成资源组在相同region不同AZ下，则需要提交工单联系DataWorks工作人员获取支持。
   - 在配置DataWorks专有网络及交换机地址时，要与文件引擎所绑定的专有网络和交换机地址一致，详情请参见独享操作资源。



2. 配置数据同步任务。
   配置数据任务的步骤请参见通过向导模式配置任务，详细参数设置如下：

i. 配置数据走向。
如果配置从MaxCompute数据同步到文件引擎时，配置的数据去向的数据源为HDFS 。



如果配置从文件引擎同步数据到MaxCompute时，配置的数据来源的数据源为HDFS。



ii. 根据提示使用脚本模式配置同步任务。



iii. 在配置脚本中，添加文件引擎配置参数。

- 同步数据到文件引擎时，需要配置HDFS Writer，详情请参见配置HDFS Writer。
- 从文件引擎同步数据时，需要配置HDFS Reader，详情请参见配置HDFS Reader。在配置HDFS Writer和HDFS Reader时，需要指定数据源，数据源配置详见：配置HDFS数据源，目前该数据源仅支持独享数据集成资源组，所以在配置之前需要先配置独享数据集成资源组，如下图所示。

```
        },
        {
            "stepType": "hdfs",
            "parameter": {
                "path": "/maxcompute2lindormstore",
                "fileName": "maxcompute2lindormstore",
                "datasource": "xxxx",
                "column": [
                    {
                        "name": "name",
                        "type": "string"
                    },
                    {
                        "name": "gender",
```

xxxx 是前面已经配置完成的数据源。

> **说明**
>
> - 由于MaxCompute默认资源组不支持Hadoop高级参数HA的配置，如果想通过HA的模式访问文件引擎，那么请新增自定义资源，详情请参见文档新增自定义资源组，并且在数据源中配置对应的 defaultFS 和 hadoopConfig 配置需要的信息都可以在文件引擎控制界面上，通过**一键生成配置**按钮查看。
> - 如果在创建数据源的时候出现网络超时的报错可以根据数据源测试连通性进行解决，如果还是有问题，可以提交工单联系DataWorks工作人员获取支持。

验证MaxCompute数据同步至文件引擎

以下示例用来验证MaxCompute数据是否同步到文件引擎上。

1. 在MaxCompute中创建测试表。详情请参见在MaxCompute创建表。

```
CREATE TABLE IF NOT EXISTS maxcompute2lindormstore
(
 name              STRING COMMENT '姓名',
 gender            STRING COMMENT '性别',
 age               INT COMMENT '年龄',
);
```

2. 在测试表中插入测试数据。

```
insert into maxcompute2lindormstore values('测试用户1','男',20);
insert into maxcompute2lindormstore values('测试用户2','男',20);
insert into maxcompute2lindormstore values('测试用户3','女',20);
insert into maxcompute2lindormstore values('测试用户4','女',20);
```

3. 在文件存储HDFS上创建目录。

```
hadoop fs -mkdir hdfs://${实例ID}/maxcompute2lindormstore
```

其中${实例ID} 表示为LindormStore的实例ID。

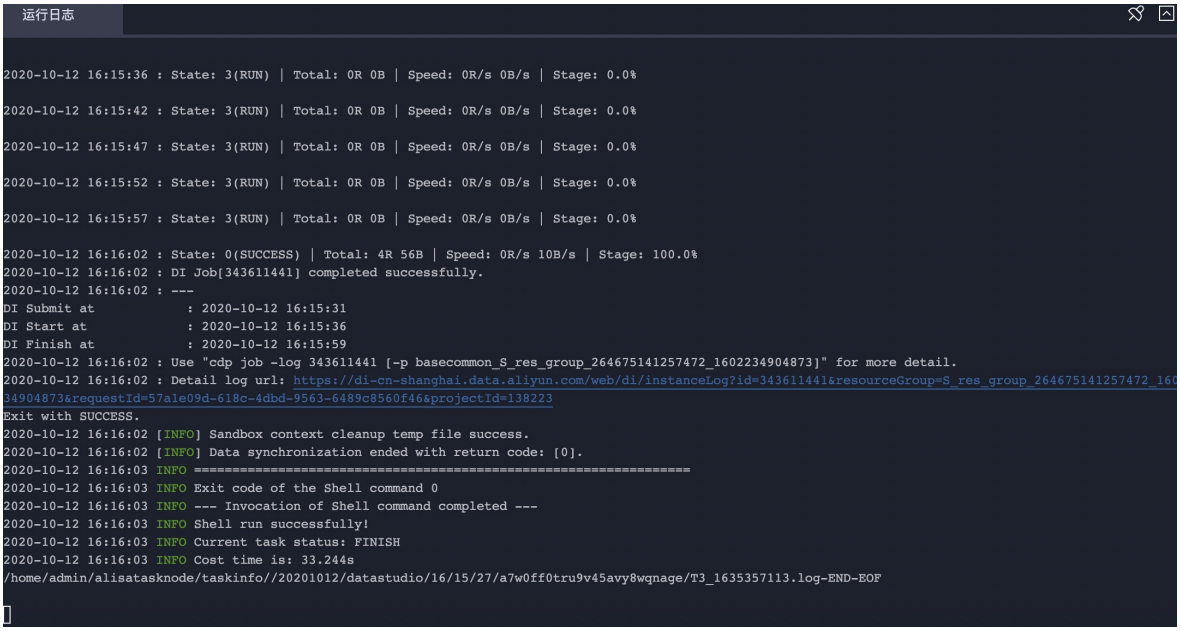
4. 在DataWorks中编写数据同步脚本。
配置MaxCompute Reader和HDFS Writer脚本，详情请参见配置MaxCompute Reader和配置HDFS Writer。

```
{
    "type": "job",
    "steps": [
        {
            "stepType": "odps",
            "parameter": {
                "partition": [],
```

```
                "datasource": "odps_first",
                "column": [
                    "*"
                ],
                "emptyAsNull": false,
                "table": "maxcompute2lindormstore"
            },
            "name": "Reader",
            "category": "reader"
        },
        {
            "stepType": "hdfs",
            "parameter": {
                "path": "/maxcompute2lindormstore",
                "fileName": "maxcompute2lindormstore",
                "datasource": "xxxx",
                "column": [
                    {
                        "name": "name",
                        "type": "string"
                    },
                    {
                        "name": "gender",
                        "type": "string"
                    },
                    {
                        "name": "age",
                        "type": "int"
                    }
                ],
                "writeMode": "append",
                "encoding": "UTF-8",
                "fieldDelimiter": ",",
                "fileType": "text"
            },
            "name": "Writer",
            "category": "writer"
        }
    ],
    "version": "2.0",
    "order": {
        "hops": [
            {
                "from": "Reader",
                "to": "Writer"
            }
        ]
    },
    "setting": {
        "errorLimit": {
            "record": ""
        },
        "speed": {
            "throttle": false,
            "concurrent": 2
        }
    }
}
{
    "type": "job",
    "steps": [
        {
            "stepType": "odps",
            "parameter": {
```

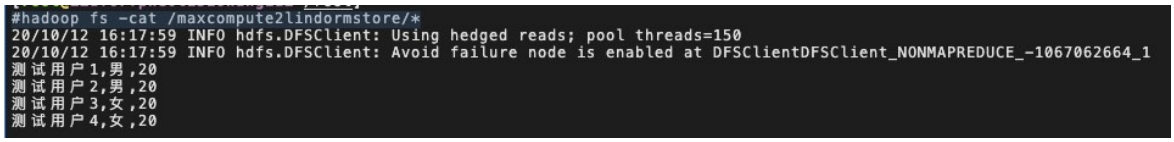
```
                "parameter": {
                    "partition": [],
                    "datasource": "odps_first",
                    "column": [
                        "*"
                    ],
                    "emptyAsNull": false,
                    "table": "maxcompute2lindormstore"
                },
                "name": "Reader",
                "category": "reader"
            },
            {
                "stepType": "hdfs",
                "parameter": {
                    "path": "/maxcompute2lindormstore",
                    "fileName": "maxcompute2lindormstore",
                    "datasource": "xxxx",
                    "column": [
                        {
                            "name": "name",
                            "type": "string"
                        },
                        {
                            "name": "gender",
                            "type": "string"
                        },
                        {
                            "name": "age",
                            "type": "int"
                        }
                    ],
                    "writeMode": "append",
                    "encoding": "UTF-8",
                    "fieldDelimiter": ",",
                    "fileType": "text"
                },
                "name": "Writer",
                "category": "writer"
            }
        ],
        "version": "2.0",
        "order": {
            "hops": [
                {
                    "from": "Reader",
                    "to": "Writer"
                }
            ]
        },
        "setting": {
            "errorLimit": {
                "record": ""
            },
            "speed": {
                "throttle": false,
                "concurrent": 2
            }
        }
    }
```

5. 加载创建的独享数据集成资源组并执行数据脚本。



6. 查看MaxCompute是否成功同步数据到文件引擎。

```
hadoop fs -cat /maxcompute2lindormstore/*
```



验证文件引擎数据同步至MaxCompute

以下示例用来验证文件引擎数据是否同步到了MaxCompute上。

> **说明** 在验证文件引擎数据同步到MaxCompute中时，文件引擎上的测试数据是使用验证MaxCompute数据同步至文件引擎中由MaxCompute同步过去的数据，将该数据再同步到MaxCompute的另外一张表中。

1. 在MaxCompute中创建新的测试表。详情请参见在MaxCompute创建表。

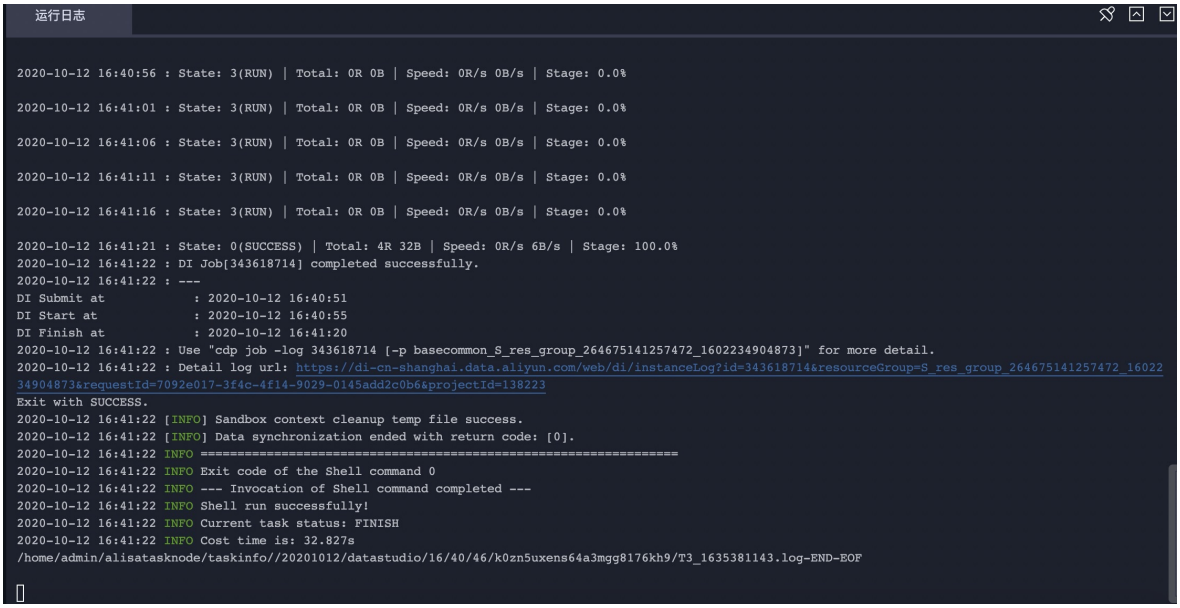
```
CREATE TABLE IF NOT EXISTS lindormstore2maxcompute
(
 name              STRING COMMENT '姓名',
 gender            STRING COMMENT '性别',
 age               INT COMMENT '年龄'
);
```

2. 在DataWorks中编写数据同步脚本。配置MaxCompute Reader和HDFS Writer脚本，详情请参见配置MaxCompute Reader和配置HDFS Writer。

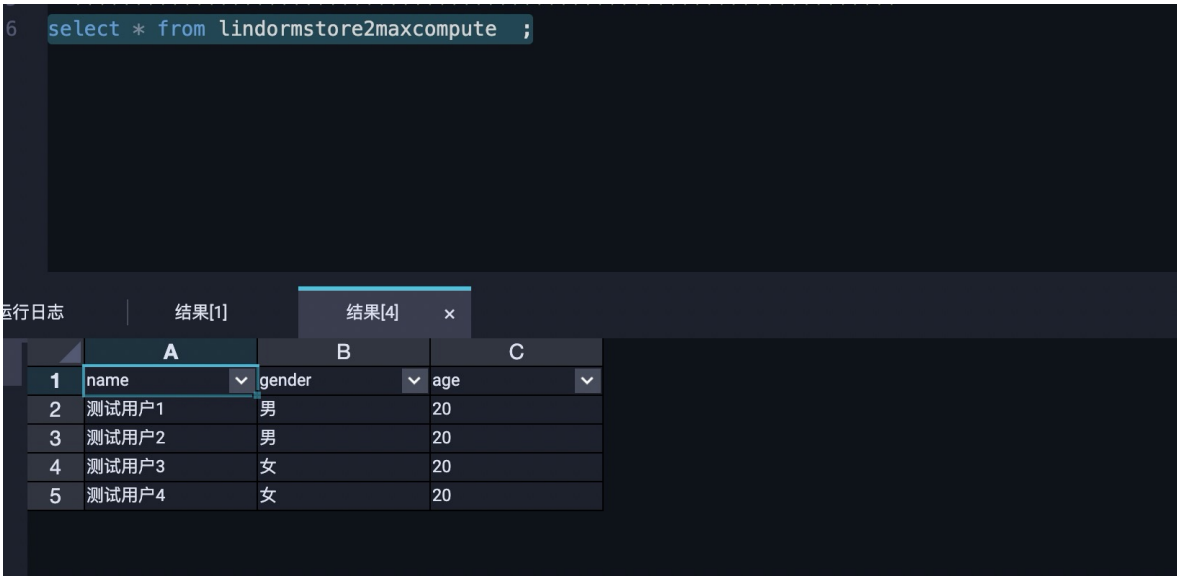
```
{
    "type": "job",
    "steps": [
        {
            "stepType": "hdfs",
            "parameter": {
                "path": "/maxcompute2lindormstore",
                "fileName": "maxcompute2lindormstore*",
                "datasource": "xxxx",
                "column": [
                    {
                        "index": 0,
                        "type": "string"
                    },
```

```
                    {
                        "index": 1,
                        "type": "string"
                    },
                    {
                        "index": 2,
                        "type": "long"
                    }
                ],
                "encoding": "UTF-8",
                "fieldDelimiter": ",",
                "fileType": "text"
            },
            "name": "Reader",
            "category": "reader"
        },
        {
            "stepType": "odps",
            "parameter":{
                "partition":"",
                "truncate":true,
                "compress":false,
                "datasource":"odps_first",
                "column": [
                        "name",
                        "gender",
                        "age"
                ],
                "guid": null,
                "emptyAsNull": false,
                "table": "lindormstore2maxcompute"
            },
            "name": "Writer",
            "category": "writer"
        }
    ],
    "version": "2.0",
    "order": {
        "hops": [
            {
                "from": "Reader",
                "to": "Writer"
            }
        ]
    },
    "setting": {
        "errorLimit": {
            "record": ""
        },
        "speed": {
            "concurrent": 2,
            "throttle": false
        }
    }
}
```

3. 加载创建的独享数据集成资源组并执行数据脚本。



4. 查看文件引擎的数据是否成功同步数据到MaxCompute。



10.5. 最佳实践

10.5.1. Ambari与文件引擎集成

Ambari提供Hadoop组件的安装、运维、监控等功能，您可以使用Ambari管理您的Hadoop集群。 本文介绍如何将Ambari与Lindorm文件引擎集成，来替换底层HDFS存储。您可以基于Ambari+Lindorm文件引擎构建云原生存储计算分离的开源大数据系统。

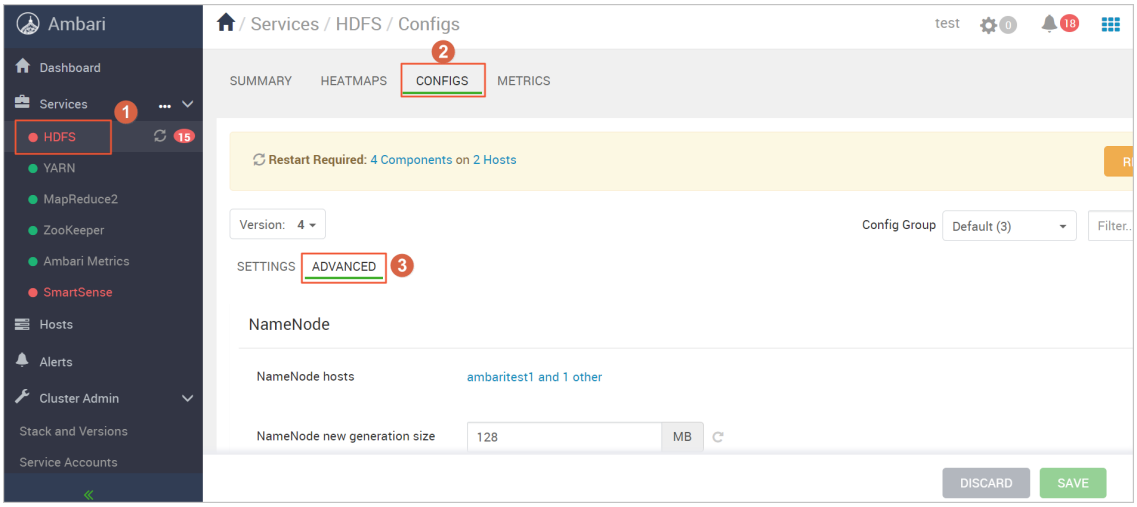
前提条件

- 请保证您的Lindorm实例和Ambari集群在同一VPC网络内。
- 请保证您的Ambari节点在Lindorm白名单中，如何添加白名单请参见设置白名单。

配置文件引擎为默认的存储引擎

1. 开通Lindorm文件引擎，具体步骤请参见开通指南。
2. 配置Lindorm文件引擎链接。

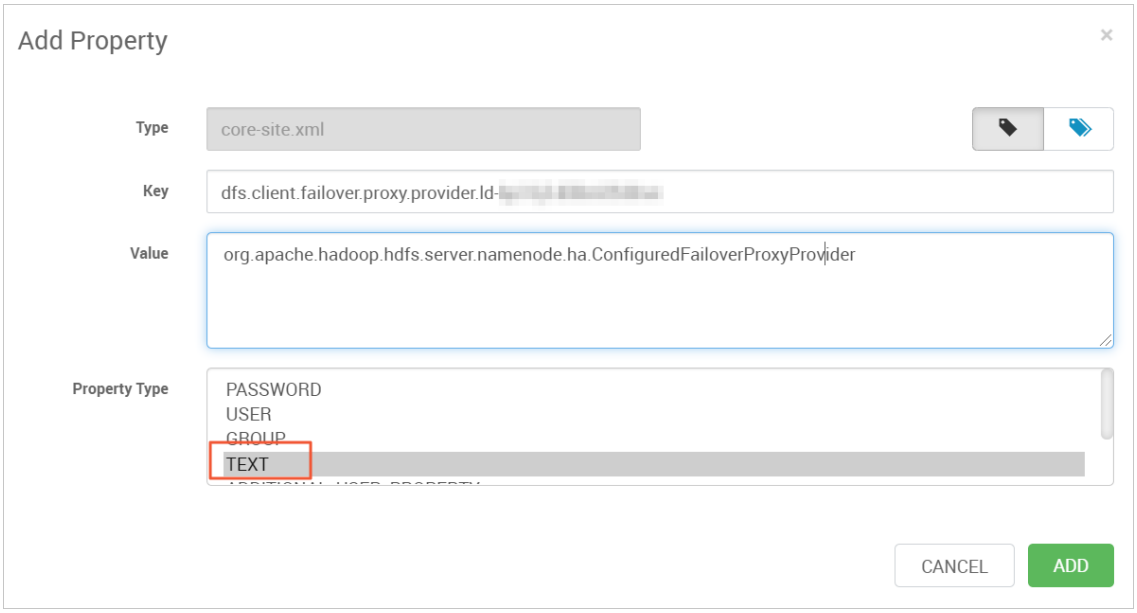
i. 登录**Ambari管理系统**，单击左侧导航栏HDFS>CONFIGS >ADVANCED，进入**HDFS配置**页面。



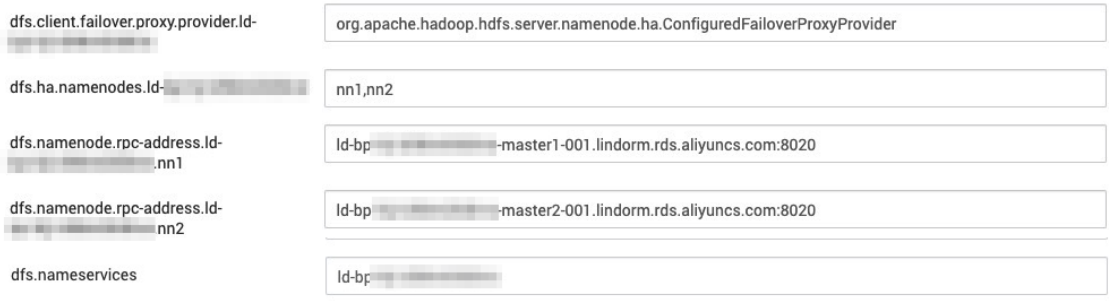
ii. 原HDFS采用HA模式（HDFS NameNode主备模式）部署，如果原HDFS非HA模式部署，请将其初始化为HA模式。

iii. 找到**Custom hdfs-site**区块，单击**Custom hdfs-site>Add Property**。配置Lindorm文件引擎配置项**Key**、**Value**、**Property Type**参数值如下：

Key	Value	Property Type	描述	
dfs.client.failover.proxy.provider.{实例Id}	org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider	TEXT	无	
dfs.ha.namenodes.{实例Id}	nn1、nn2	TEXT	HA模式下主备NameNode的服务ID名称	
dfs.namenode.rpc-address.{实例Id}	{实例Id}-master1-001.lindorm.rds.aliyuncs.com:8020	TEXT	无	
dfs.namenode.rpc-address.{实例Id}	{实例Id}-master2-001.lindorm.rds.aliyuncs.com:8020	TEXT	无	
dfs.nameservices	{实例Id}	TEXT	无	

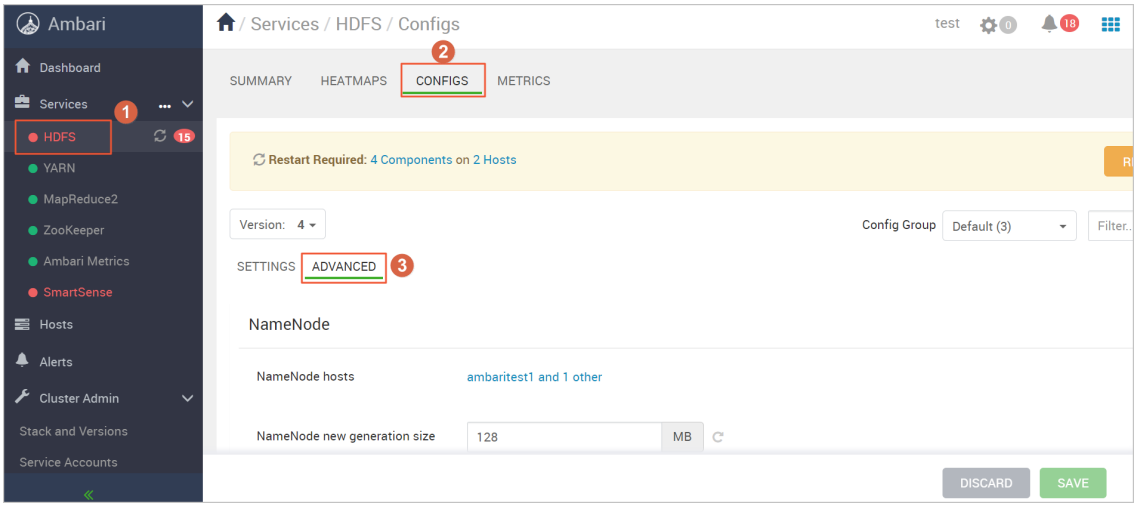


iv. 单击**ADD**完成配置。



3. 配置Ambari默认访问Lindorm文件引擎。

i. 登录**Ambari管理系统**，单击左侧导航栏**HDFS>CONFIGS >ADVANCED**，进入**HDFS配置**页面。



ii. 找到**Advanced core-site**区块，修改*fs.defaultFS*配置项为您文件引擎的HDFS连接串，获取该连接串请参见开通指南。



iii. 修改*hdfs-site*配置文件。

a. 在**Ambari管理系统**主页面，单击**HDFS > CONFIGS > ADVANCED**，找到**Custom hdfs-site**区块。

b. 修改配置项*dfs.internal.nameservices*值为实例Id。

c. 添加配置项*dfs.namenode.http-address.{实例Id}.nn1*，设置参数值：`{实例Id}-master1-001.lindorm.rds.aliyuncs.com:50070`。

d. 添加配置项*dfs.namenode.http-address.{实例Id}.nn2*，设置参数值：`{实例Id}-master2-001.lindorm.rds.aliyuncs.com:50070`。

iv. 单击**SAVE**保存修改。

v. 单击页面右上角**ACTIONS>Restart All**，重启HDFS。

> **注意** 该步骤是为了将修改后的配置信息分发到Ambari的各个节点上，并且由于配置已经被修改，HDFS启动会失败，直接将服务关闭即可。验证Ambari群能否成功访问文件引擎。在Ambari集群中的任意节点上执行如下指令。若返回结果如下，则表示链接配置成功。

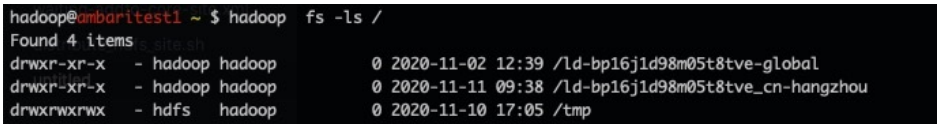
vi. 单击页面右上角**ACTIONS>Stop**，关闭HDFS。

4. 验证Ambari集群是否成功访问文件引擎。

i. 在 Ambari集群中任意节点上执行如下指令。

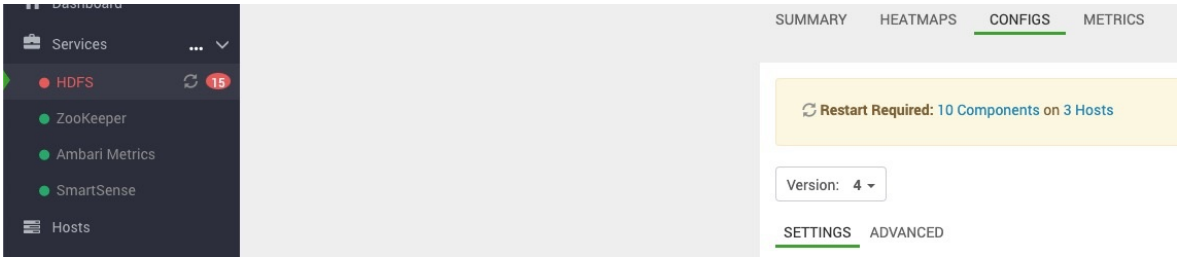
```
$ hadoop fs -ls /
```

ii. 执行结果如下，表示链接配置成功。



> **注意** 如果验证时发现配置没有生效，可以在**Ambari管理系统**页面右上角单击**ACTIONS>Restart All**来重新启动HDFS，然后再单击Stop关闭HDFS，配置即可生效。

5. 若当前已经启动的各个组件的状态如下图，则底层的文件引擎已经替换成功。

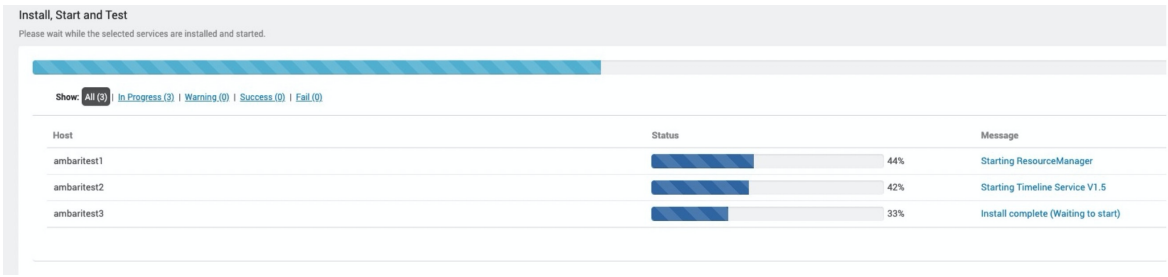


安装YARN服务

1. 登录**Ambari管理系统**，单击Services旁边的 ••• 图标，单击**Add Service**，并将**YARN+MapReduce2**选项打钩。



2. 在**Add Service Wizard**页面按照安装向导安装YARN服务，配置完成后单击**DEPLOY**进行安装，等待安装完成。



3. 验证YARN服务是否启动成功。
   使用Ambari Hadoop中自带的测试包 `hadoop-mapreduce-examples-3.1.1.3.1.4.0-315.jar` 进行测试。在Ambari中，该测试包在*usr/hdp/3.1.4.0-315/hadoop-mapreduce*目录下。
   i. 登入Ambari任意机器，执行以下命令，在*/tmp/randomtextwriter*目录下生成128M大小的测试文件。

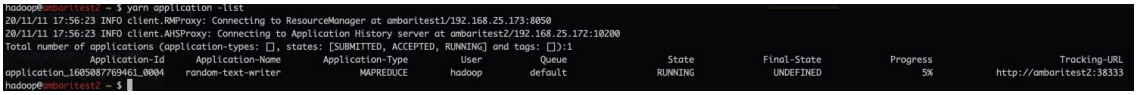
```
$ yarn jar /usr/hdp/3.1.4.0-315/hadoop-mapreduce/hadoop-mapreduce-examples-3.1.1.3.1.4.0-315.jar ran
domtextwriter  -D mapreduce.randomtextwriter.totalbytes=134217728  -D mapreduce.job.maps=4 -D mapred
uce.job.reduces=4   /tmp/randomtextwriter
```

> **注意** 其中 `hadoop-mapreduce-examples-3.1.1.3.1.4.0-315.jar` 为Ambari中的测试包，请根据实际情况修改。

   ii. 检查任务是否已经提交到YARN服务上执行。
   在Ambari任意机器执行以下命令。

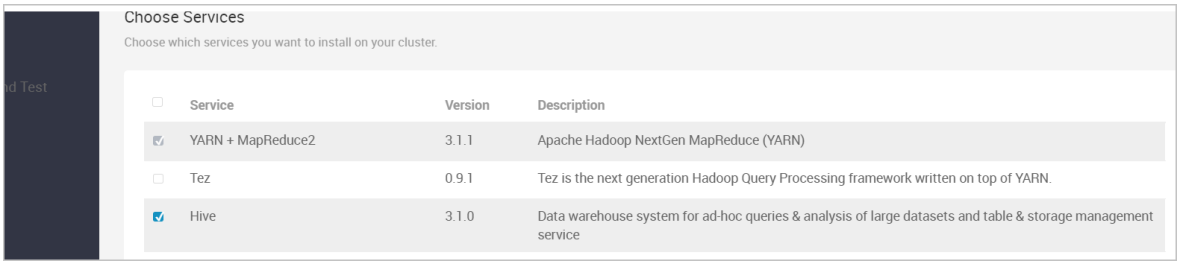
```
$ yarn application -list
```

   如果返回结果如下，则表示YARN服务正常运行。

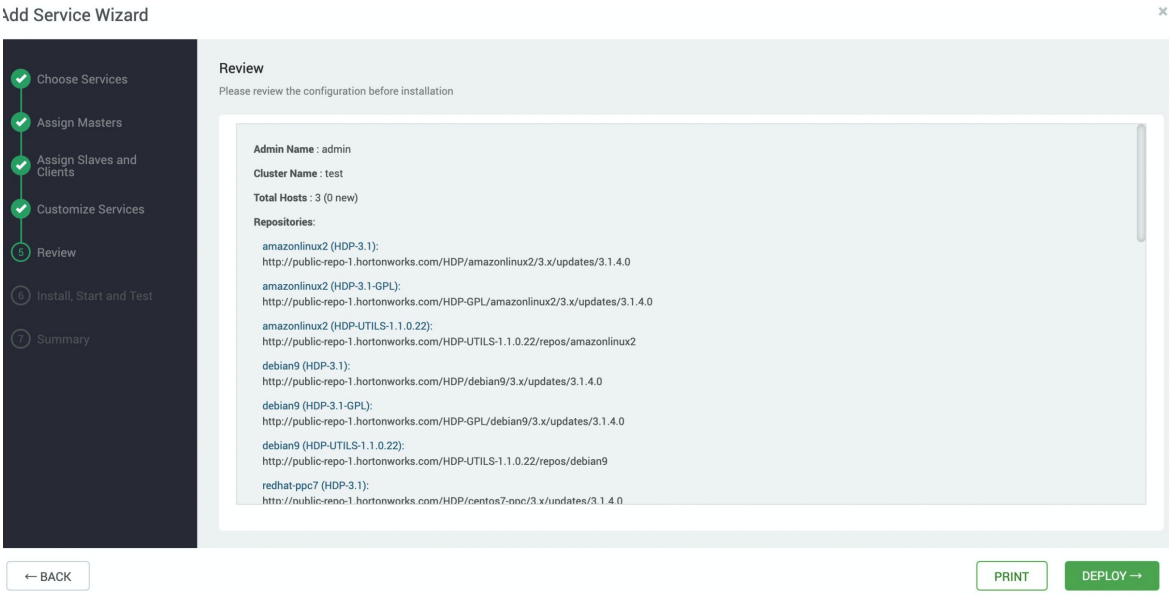


安装Hive服务

1. 登录**Ambari管理系统**，单击Services旁边的 ••• 图标，单击**Add Service**，并将**Hive**选项打钩。



2. 在**Add Service Wizard**页面按照安装向导安装Hive服务，配置完成后单击**DEPLOY**进行安装，等待安装完成。



3. 如下页面表示安装成功。



4. 设置proxyuser代理。由于Hive是通过proxyuser代理方式访问文件引擎，所以我们提前先配置了一些常用用户的proxy访问权限，如Hive、spark等。当您有其他用户需要添加的时候，可以咨询技术支持帮您手动添加。
5. 重启YARN服务。由于Hive为了加速任务的执行而使用Tez组件，而Tez组件依赖于YARN服务，因此在安装Hive时Ambari会添加部分YARN的配置，为了是配置可以生效您需要重启YARN服务。
   单击左侧导航栏的YARN服务，单击页面右上角**ACTIONS>Restart All**，重启 YARN服务 。
6. 验证Hive服务是否成功启动。

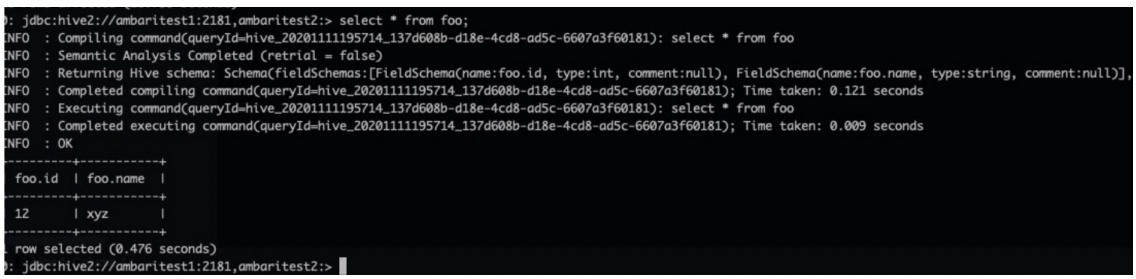
i. 登入Ambari任意机器，执行以下命令。

```
# su - hive
# 登入 hive 客户端
hive@ambaritest2 ~]$ hive
Beeline version 3.1.0.3.1.4.0-315 by Apache Hive
0: jdbc:hive2://ambaritest1:2181,ambaritest2:> create table foo (id int, name string);
INFO  : Compiling command(queryId=hive_20201111193943_5471ede8-e51f-44b8-a91a-b6fde9f58b49): create
table foo (id int, name string)
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20201111193943_5471ede8-e51f-44b8-a91a-b6fde9f58b49
); Time taken: 1.337 seconds
INFO  : Executing command(queryId=hive_20201111193943_5471ede8-e51f-44b8-a91a-b6fde9f58b49): create
table foo (id int, name string)
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20201111193943_5471ede8-e51f-44b8-a91a-b6fde9f58b49
); Time taken: 0.814 seconds
INFO  : OK
No rows affected (2.596 seconds)
0: jdbc:hive2://ambaritest1:2181,ambaritest2:> insert into table foo select * from (select 12,"xyz")
a;
# su - hive
# 登入 hive 客户端
hive@ambaritest2 ~]$ hive
Beeline version 3.1.0.3.1.4.0-315 by Apache Hive
0: jdbc:hive2://ambaritest1:2181,ambaritest2:> create table foo (id int, name string);
INFO  : Compiling command(queryId=hive_20201111193943_5471ede8-e51f-44b8-a91a-b6fde9f58b49): create
table foo (id int, name string)
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=hive_20201111193943_5471ede8-e51f-44b8-a91a-b6fde9f58b49
); Time taken: 1.337 seconds
INFO  : Executing command(queryId=hive_20201111193943_5471ede8-e51f-44b8-a91a-b6fde9f58b49): create
table foo (id int, name string)
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20201111193943_5471ede8-e51f-44b8-a91a-b6fde9f58b49
); Time taken: 0.814 seconds
INFO  : OK
No rows affected (2.596 seconds)
0: jdbc:hive2://ambaritest1:2181,ambaritest2:> insert into table foo select * from (select 12,"xyz")
a;
```

ii. 执行以下命令，查询Hive上的数据。

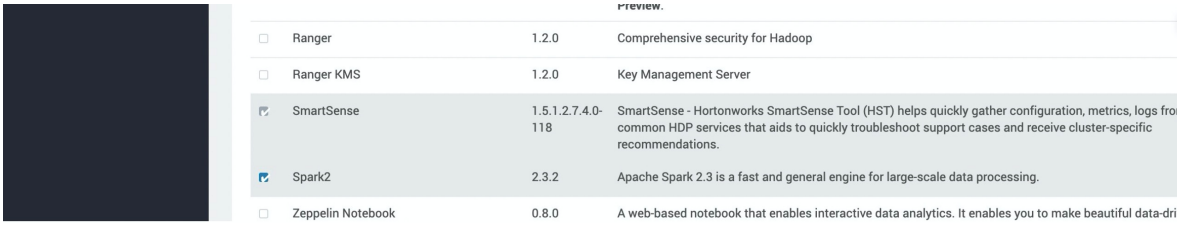
```
0: jdbc:hive2://ambaritest1:2181,ambaritest2:> select * from foo;
```

若返回结果如下，则表示Hive服务安装并启动成功。

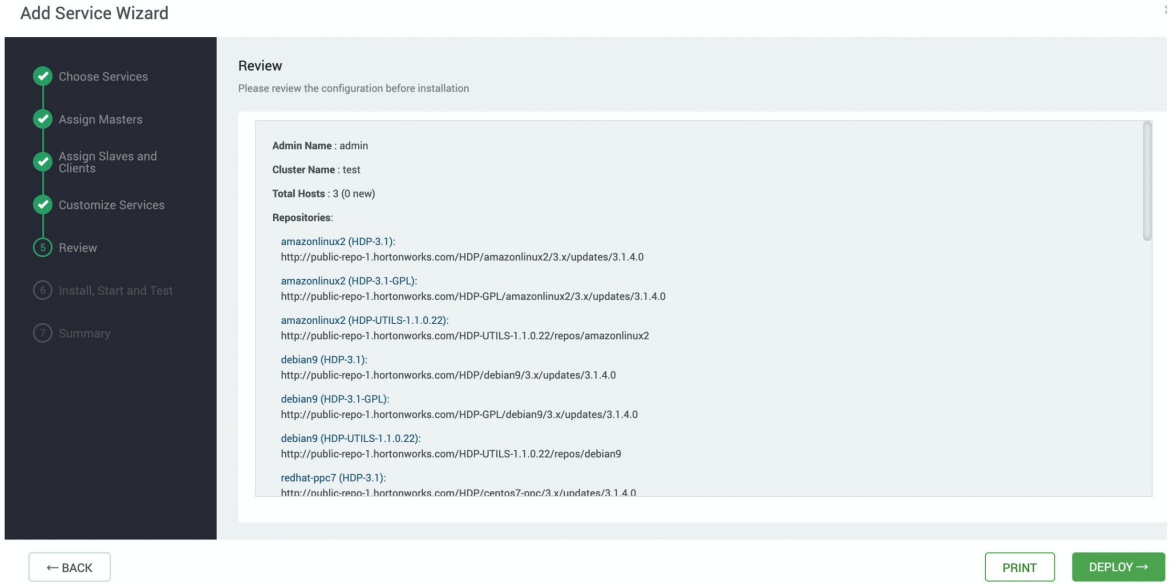


安装Spark服务

1. 登录**Ambari管理系统**，单击Services旁边的 ••• 图标，单击**Add Service**，并将**Spark2**选项打钩。



2. 按照安装向导，安装Spark2服务，配置完后单击**DEPLOY**进行安装，等待安装完成。



3. 验证Spark是否启动成功。
   使用Ambari spark中自带的测试包 `spark-examples_2.11-x.x.x.x.x.x.0-315.jar` 进行测试。在Ambari中，该测试包在*/usr/hdp/3.1.4.0-315/spark2/examples/jars/*下。

   i. 登入Ambari任意机器，执行以下命令，在/tmp/randomtextwriter目录下生成128M大小的测试文件，如果之前已经生成请跳过该步骤。

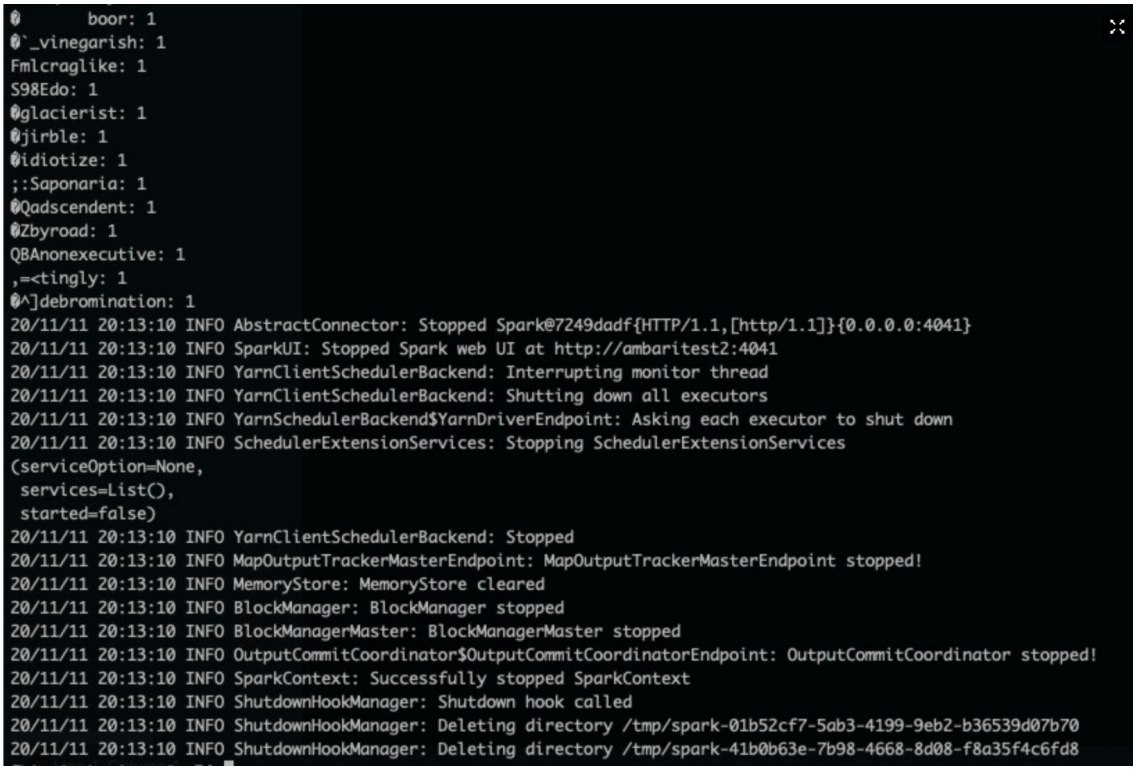
```
$ yarn jar /usr/hdp/3.1.4.0-315/hadoop-mapreduce/hadoop-mapreduce-examples-3.1.1.3.1.4.0-315.jar randomtextwriter  -D mapreduce.randomtextwriter.totalbytes=134217728  -D mapreduce.job.maps=4 -D mapreduce.job.reduces=4   /tmp/randomtextwriter
```

   说明 其中 `hadoop-mapreduce-examples-3.1.1.3.1.4.0-315.jar` 为Ambari中的测试包，请根据实际情况进行修改。

ii. 登入Ambari任意机器，执行以下命令，使用spark测试包从文件引擎上读取测试文件并输出word count程序运行结果。

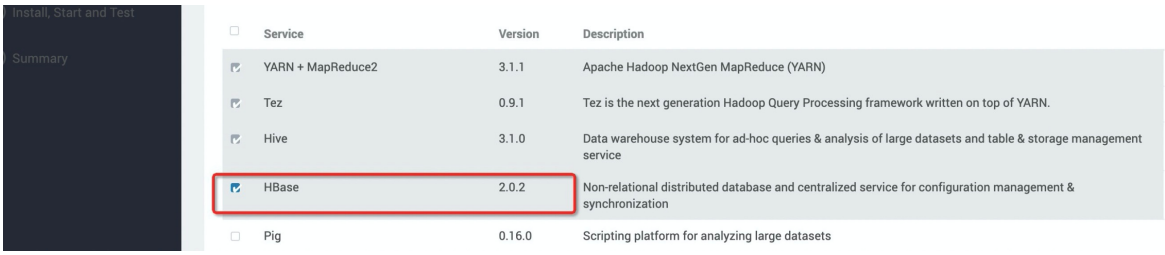
```
$ spark-submit   --master yarn --executor-memory 2G --executor-cores 2  --class org.apache.spark.examples.JavaWordCount  /usr/hdp/3.1.4.0-315/spark2/examples/jars/spark-examples_2.11-2.3.2.3.1.4.0-315.jar  /tmp/randomtextwriter
```

若任务运行结果如下图所示，则表示Spark启动成功。

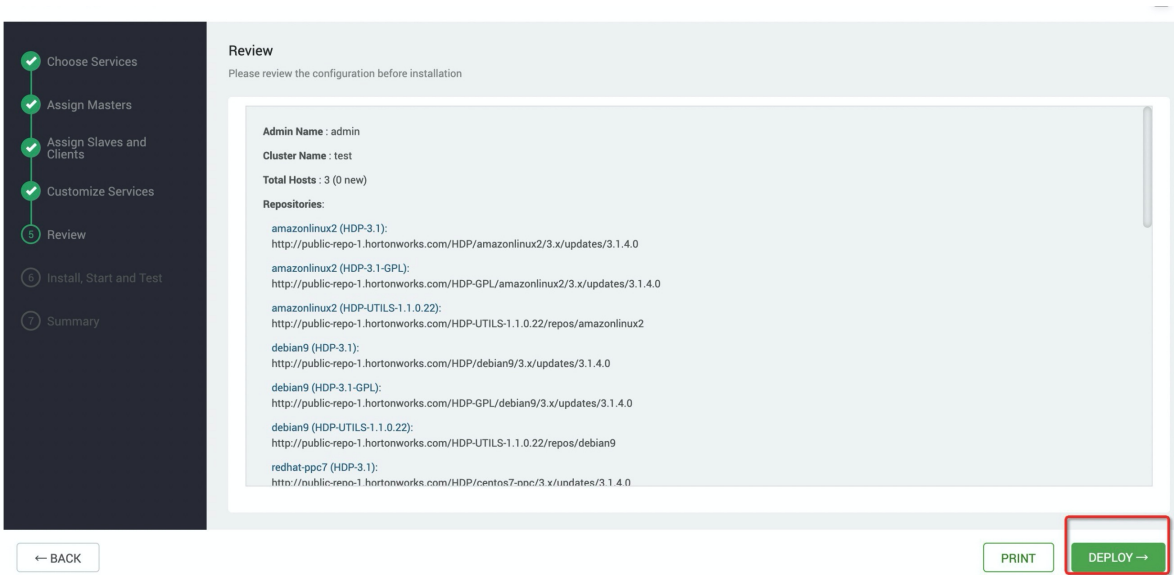


安装Hbase服务

1. 登录**Ambari管理系统**，单击Services旁边的 ••• 图标，单击**Add Service**，并将**Hbase**选项打钩。



2. 按照安装向导，安装Hbase服务，配置完后单击**DEPLOY**进行安装，等待安装完成。



3. 验证Hbase服务是否启动成功。
   i. 登入Ambari任意机器，执行以下命令进入**hbase Shell**命令界面。

```
[spark@ambaritest1 ~]$  hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.1.4.0-315/phoenix/phoenix-5.0.0.3.1.4.0-315-server.jar!
/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.1.4.0-315/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4
j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.0.2.3.1.4.0-315, r, Fri Aug 23 05:15:48 UTC 2019
Took 0.0023 seconds
hbase(main):001:0>
```

ii. 执行以下命令，在HBase中创建测试表。

```
[hive@ambaritest2 ~]$ hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.1.4.0-315/phoenix/phoenix-5.0.0.3.1.4.0-315-server.jar!
/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.1.4.0-315/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4
j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.0.2.3.1.4.0-315, r, Fri Aug 23 05:15:48 UTC 2019
Took 0.0023 seconds
hbase(main):001:0> create 'hbase_test','info'
Created table hbase_test
Took 1.9513 seconds
=> Hbase::Table - hbase_test
hbase(main):002:0> put 'hbase_test','1', 'info:name' ,'Sariel'
Took 0.2576 seconds
hbase(main):003:0> put 'hbase_test','1', 'info:age' ,'22'
Took 0.0078 seconds
hbase(main):004:0>  put 'hbase_test','1', 'info:industry' ,'IT'
Took 0.0077 seconds
hbase(main):005:0> scan 'hbase_test'
ROW                                                                                 COLUMN+C
ELL
 1                                                                                  column=i
nfo:age, timestamp=1605097177701, value=22
 1                                                                                  column=i
nfo:industry, timestamp=1605097181758, value=IT
 1                                                                                  column=i
nfo:name, timestamp=1605097174400, value=Sariel
1 row(s)
Took 0.0230 seconds
hbase(main):006:0>
```

iii. 执行以下命令查看文件引擎的*/apps/hbase/data/data/default*路径，如果*/apps/hbase/data/data/default*路径下有*hbase_test*目录，则证明配置链接成功。

```
[hive@ambaritest2 ~]$ hadoop  fs -ls /apps/hbase/data/data/default
Found 1 items
drwxr-xr-x   - hbase hadoop          0 2020-11-11 20:19 /apps/hbase/data/data/default/hbase_test
[hive@ambaritest2 ~]$
```

安装其他服务

其他服务的安装和上面服务的安装基本类似，安装过程可以查考上述步骤进行安装。

10.5.2. CDH6与文件引擎集成

CDH（Cloudera's Distribution, including Apache Hadoop）提供Hadoop组件的安装、运维、监控等功能，您可以使用CDH6（表示CDH 6.X 版本）管理您的Hadoop集群。本文介绍如何将CDH6与Lindorm文件引擎集成，来替换底层HDFS存储。您可以基于CDH6和Lindorm文件引擎构建云原生存储计算分离的开源大数据系统。

前提条件

- 请保证您的Lindorm实例和CDH6集群所在同一VPC内。

- 请保证您的CDH6节点在Lindorm白名单中，如何添加白名单请参见设置白名单。

> **说明**
> 本文中所涉及范例使用的CDH版本为V6.3.2。Lindorm文件引擎支持CDH 5.13.0以上版本，其他版本请咨询专家服务。

一、配置文件引擎为默认的存储引擎

1. 开通Lindorm文件引擎，具体步骤请参见开通指南。
2. 配置Lindorm文件引擎链接。
   i. 登入CDH6系统主页，选择**配置>高级配置代码段**，进入**高级配置代码段**页面。



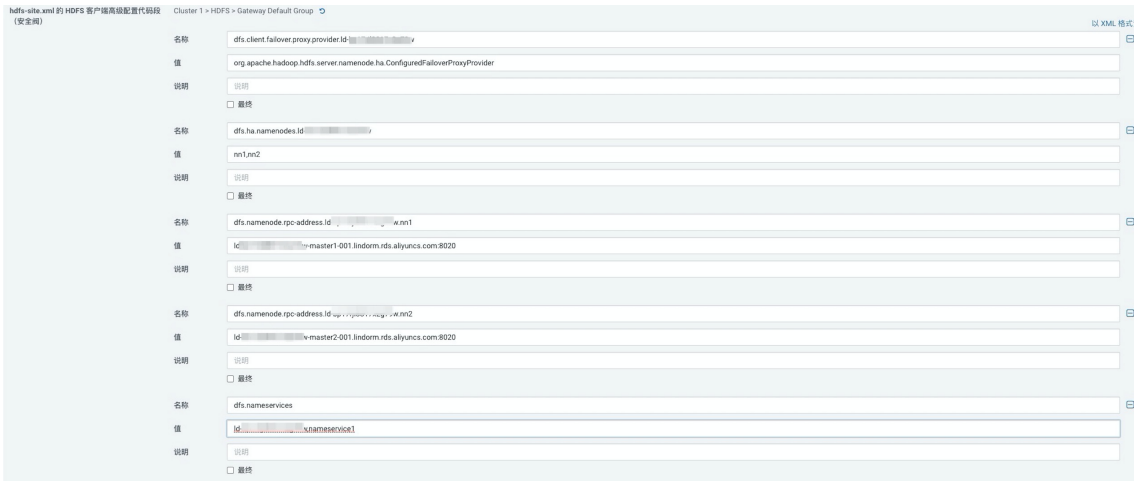
   ii. 原HDFS采用HA模式（HDFS NameNode主备模式）部署，如果原HDFS非HA模式部署，请将其初始化为HA模式。
   iii. 在最上面的搜索框中搜索**hdfs-site.xml**。

iv. 在**hdfs-site.xml的HDFS客户端高级配置代码段（安全阀）**区域中，添加如下Lindorm文件引擎配置项，如下：

名称	值	说明	是否必配
dfs.client.failover.proxy.provider.{实例ID}	org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider	主备切换策略配置	是
dfs.ha.namenodes.${实例ID}	nn1,nn2	HA模式下主备NameNode的服务ID名称	是
dfs.namenode.rpc-address.${实例ID}.nn1	${实例ID}-master1-001.lindorm.rds.aliyuncs.com：8020	NameNode第一个节点RPC通信地址	是
dfs.namenode.rpc-address.${实例ID}.nn2	${实例ID}-master2-001.lindorm.rds.aliyuncs.com：8020	NameNode第二个节点RPC通信地址	是
dfs.nameservices	${实例ID},nameservice1	nameservice1为原来的HDFS的服务名称	是

单击下图的按钮可以添加配置。





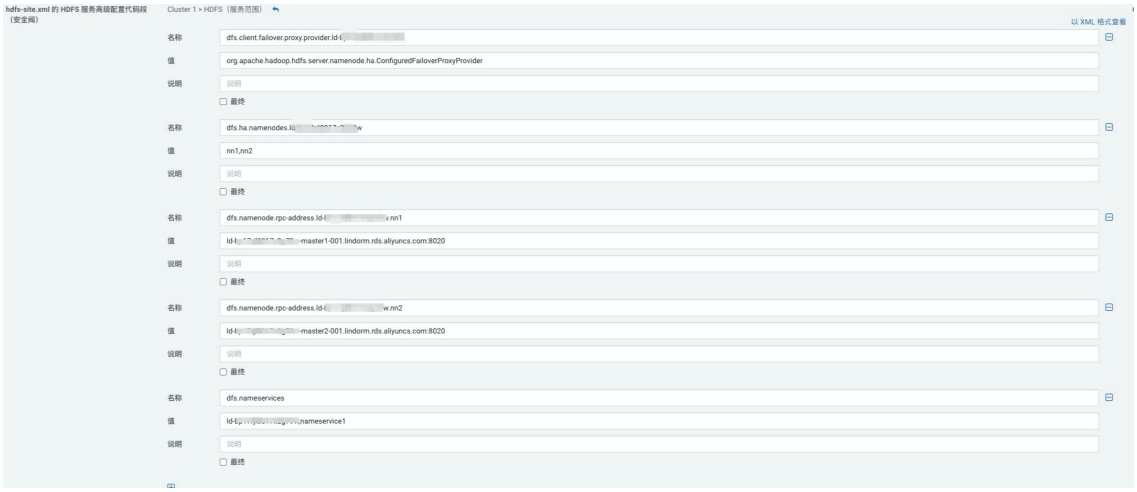
> **说明**
> 其中${实例ID}，需要根据实际情况进行修改。

v. 在**hdfs-site.xml的HDFS服务高级配置代码段（安全阀）**区域中，添加如下Lindorm文件引擎配置项，如下：

名称	值	说明	是否必配
dfs.client.failover.proxy.provider.{实例ID}	org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider		是
dfs.ha.namenodes.${实例ID}	nn1,nn2	HA模式下主备NameNode的服务ID名称	是
dfs.namenode.rpc-address.${实例ID}.nn1	${实例ID}-master1-001.lindorm.rds.aliyuncs.com：8020		是
dfs.namenode.rpc-address.${实例ID}.nn2	${实例ID}-master2-001.lindorm.rds.aliyuncs.com：8020		是
dfs.nameservices	${实例ID},nameservice1	nameservice1为原来的hdfs的服务名称	是

单击下图的按钮可以添加配置。





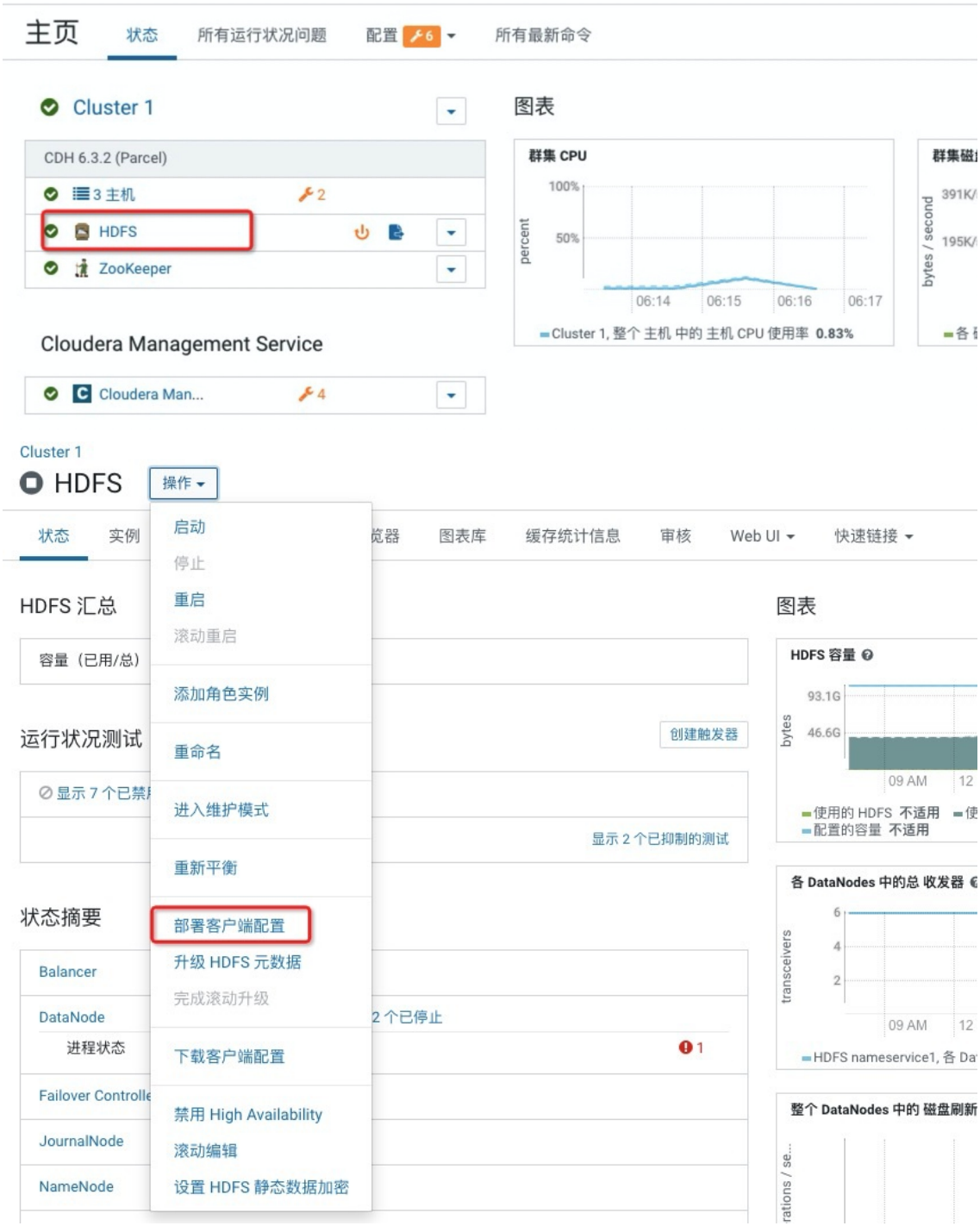
> **说明**
> 其中${实例ID}，需要根据实际情况进行修改。

vi. 在搜索框中搜索 **core-site.xml** 。

vii. 在 **core-site.xml 的群集范围高级配置代码段（安全阀）**区域中，添加 fs.defaultFS， 其值：hdfs：//${实例ID} ，含义是将文件引擎设置为默认的存储引擎。

viii. 单击**保存更改**。

ix. 返回CDH6系统主页，找到**HDFS**，单击图标进入HDFS控制台，然后再单击**操作>部署客户端配置**。



3. 验证CDH6集群是否默认访问文件引擎。
   i. 在CDH6集群中的任一一个节点上执行如下指令。

```
$ hadoop fs -ls /
```

   ii. 执行结果如下，表示链接配置成功。

```
hadoop@cdhlindorm002 ~ $ hadoop fs -ls /
Found 2 items
drwxr-xr-x   - hadoop hadoop          0 2020-11-16 14:49 /ld-bp17rjl8817x2g79w-global
drwxr-xr-x   - hadoop hadoop          0 2020-11-16 17:49 /ld-bp17rjl8817x2g79w_cn-hangzhou
hadoop@cdhlindorm002 ~ $
```

二 、安装Yarn服务

1. 修改Yarn的安装脚本。
   i. 登入您准备作为ResourceManager（Yarn服务中资源的管控节点）。
   ii. 执行如下的指令修改安装脚本。

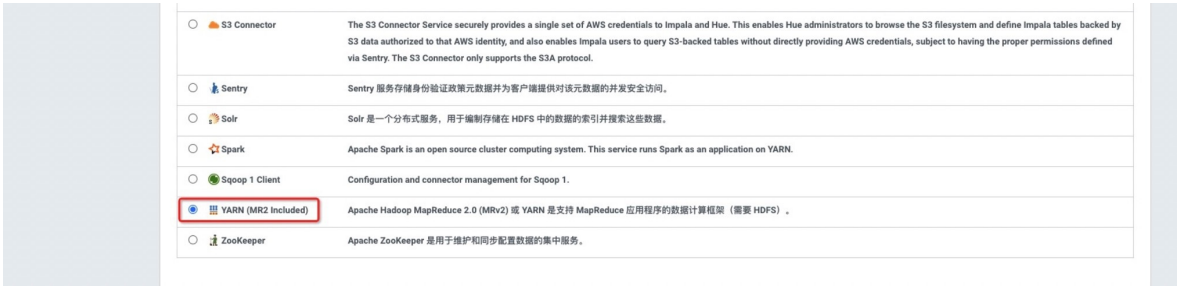
```
# sudo su -
root@cdhlindorm001 /opt/cloudera/cm-agent/service $ vim /opt/cloudera/cm-agent/service/yarn/yarn.sh
  # 找到 DEFAULT_FS ，并在其下面添加如下配置信息。
  DEFAULT_FS="$3"
  DEFAULT_FS="hdfs：//{实例ID}
  # 其中${实例ID}，需要根据实际情况进行修改。
```

> **说明**
> 该脚本是Yarn 在安装的时候，用于配置环境的脚本。

2. 登录CDH6管理系统，单击集群名称右边的 ，然后单击**添加服务**，进入安装向导界面。

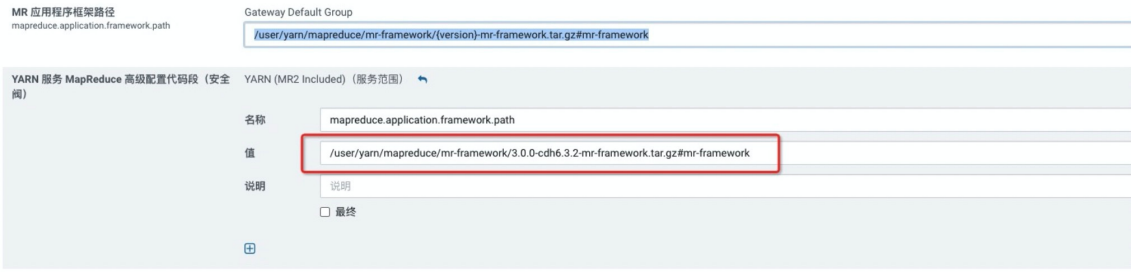


3. 选择YRAN服务，单击**继续**，根据**安装指南**配置YARN服务，配置都可以按照默认配置。



4. 等服务安装完成之后，需要修改mapred-site.xml配置。
   i. 登入 CDH6 系统主页，选择**配置>高级配置代码段**，进入**高级配置代码段**页面。
   ii. 在最上面的搜索框中搜索mapred-site.xml。

iii. 在**YARN 服务 MapReduce 高级配置代码段（安全阀）**中，添加如下配置项。
配置项mapreduce.application.framework.path，其值：/user/yarn/mapreduce/mr-framework/{version}-mr-framework.tar.gz#mr-framework

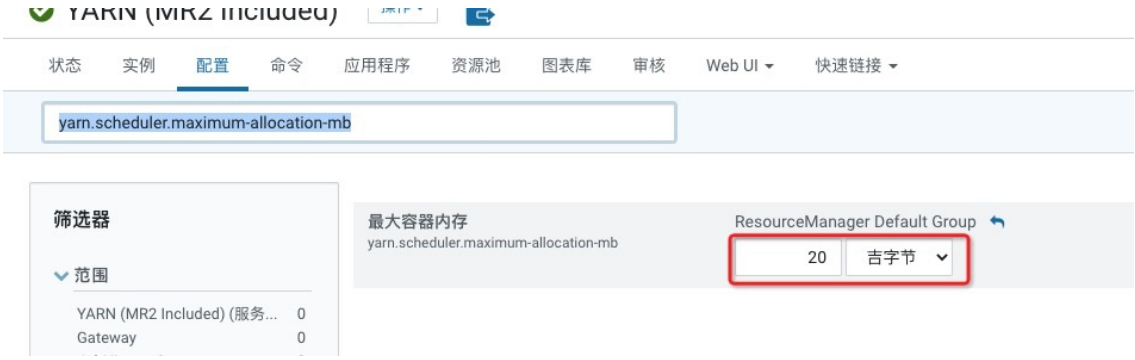


iv. 单击**保存更改**。

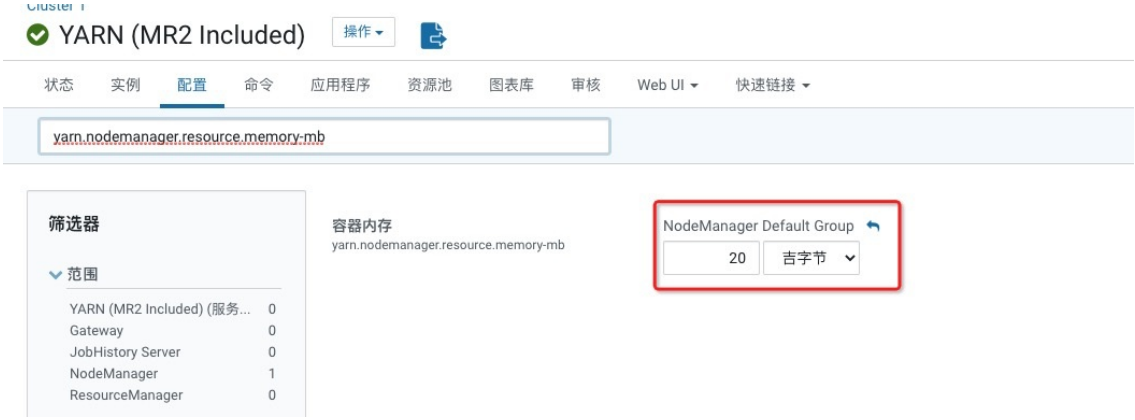
5. 配置 Yarn 可以分配的内存容量（因为在我们完成Spark安装之后需要通过Yarn启动程序）。
   i. 返回 CDH6 系统主页，找到**YARN**，单击进入Yarn控制台。
   ii. 找到配置，单击进入。



iii. 在最上面的搜索框中输入yarn.scheduler.maximum-allocation-mb，将其值设置为20 G（该值根据您实际的情况配置）。



iv. 在最上面的搜索框中输入yarn.nodemanager.resource.memory-mb，将其值设置为20 G（该值根据您实际的情况配置）。



v. 单击**保存更改**。

6. 部署新配置并重启服务。

i. 返回CDH6管理系统界面，找到**YARN**，单击重新部署图标，进行重新部署。

ii. 在**过期配置**页面，单击**重启过时服务**。



iii. 在**重启过时服务**页面，选择需要重启的服务，单击**立即重启**。

iv. 等待服务全部重启完成，并重新部署客户端配置后，单击**完成**。

7. 验证hive服务是否启动成功。
使用CDH6 Hadoop中自带的测试包hadoop-mapreduce-examples-3.0.0-cdh6.3.2.jar（版本号以客户自带的版本为准）进行测试。在CDH6中，该测试包在/opt/cloudera/parcels/CDH-6.3.2-1.cdh6.3.2.p0.1605554/jars 下。

i. 登入CDH6任一一台机器，执行以下命令，在/tmp/randomtextwriter目录下生成128M大小的文件。

```
yarn jar /opt/cloudera/parcels/CDH-6.3.2-1.cdh6.3.2.p0.1605554/jars/hadoop-mapreduce-examples-3.0.0-cdh6.3.2.jar randomtextwriter  -D mapreduce.randomtextwriter.totalbytes=134217728  -D mapreduce.job.maps=4 -D mapreduce.job.reduces=4   /tmp/randomtextwriter
```

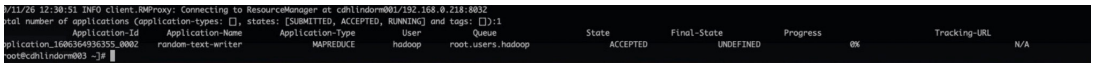
其中hadoop-mapreduce-examples-3.1.1.3.1.4.0-315.jar为CDH6中的测试包，请根据实际情况修改。

ii. 检查任务是否已经提交到Yarn上执行。

a. 在 CDH6 任一一台机器，执行以下命令。

```
$ yarn application -list
```

如果执行结果如下，表示YARN正常运行。



b. 检查文件是否正常生成。

a. 在CDH6任一一台机器，执行以下命令。

```
$  hadoop fs -ls  /tmp/randomtextwriter
```

如果执行结果如下，正常运行。



三、安装HIVE服务

1. 安装MySQL数据库，并创建用来存储hive元数据信息的数据库。
   i. 登入CDH6任意的一台机器，执行如下指令，进行安装。

```
# 切换到 root
sudo su -
# 下载 MySQL的rpm 源
root@cdhlindorm001 ~/tool $ wget http：//repo.mysql.com/mysql-community-release-el7-5.noarch.rpm、
#安装 MySQL server和MySQL客户端
root@cdhlindorm001 ~/tool $ yum install mysql-server mysql -y
```

   ii. 启动MySQL服务，并且创建用户，设置用户的权限。
      a. 执行如下指令进行操作。

```
# 启动 MySQL服务
root@cdhlindorm001 ~/tool $ systemctl start mysqld.service
# 登入MySQL终端，创建用户，并且设置权限
root@cdhlindorm001 ~/tool $ mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version：  5.6.50 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE USER 'hive'@'%' IDENTIFIED BY '123456';
Query OK, 0 rows affected (0.00 sec)
mysql> create DATABASE hive;
Query OK, 1 row affected (0.00 sec)
mysql> use mysql;
Query OK, 0 rows affected (0.00 sec)
mysql> delete from user;
Query OK, 9 rows affected (0.00 sec)
mysql> grant all privileges on *.* to 'root'@'%' identified by '123456' with grant option;
# 刷新权限
mysql> flush privileges;
Query OK, 0 rows affected (0.01 sec)
```

      b. 验证MySQL是否配置成功。

```
# 登入成功，配置完成
root@cdhlindorm001 ~/tool $ mysql -uroot -p
Warning：  Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10
Server version：  5.6.50 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

2. 登录CDH6管理系统，单击集群名称右边的 ，然后单击**添加服务**，进入安装向导界面。

3. 选择 HIVE 服务，单击**继续**，根据**安装指南**配置HIVE服务，根据向导配置对应的参数。



i. 配置数据库信息。



ii. 开始安装，并且等待安装完成。



4. 验证是否启动成功。

i. 登入CDH6 任一一台机器，执行以下命令。

```
# 登入 hive 客户端
[root@cdhlindorm001 ~]# hive
WARNING: Use "yarn jar" to launch YARN applications.
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar: file: /opt/cloudera/parcels/CDH-6.3.2-1.cdh6.3.2.p0.1605554/jars/log4j
-slf4j-impl-2.8.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar: file: /opt/cloudera/parcels/CDH-6.3.2-1.cdh6.3.2.p0.1605554/jars/slf4j
-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http: //www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]

Logging initialized using configuration in jar: file: /opt/cloudera/parcels/CDH-6.3.2-1.cdh6.3.2.p0.1
605554/jars/hive-common-2.1.1-cdh6.3.2.jar!/hive-log4j2.properties Async: false

WARNING: Hive CLI is deprecated and migration to Beeline is recommended.
hive> create table foo (id int, name string);
OK
Time taken: 1.409 seconds
hive> insert into table foo select * from (select 12,"xyz")a;
Query ID = root_20201126162112_59e6a5fc-99c2-45a4-bf84-73c16c39de8a
Total jobs = 3
Launching Job 1 out of 3
Number of reduce tasks is set to 0 since there's no reduce operator
20/11/26 16: 21: 12 INFO client.RMProxy: Connecting to ResourceManager at cdhlindorm001/192.168.0.21
8: 8032
20/11/26 16: 21: 13 INFO client.RMProxy: Connecting to ResourceManager at cdhlindorm001/192.168.0.21
8: 8032
Starting Job = job_1606364936355_0003, Tracking URL = http: //cdhlindorm001: 8088/proxy/application_1
606364936355_0003/
Kill Command = /opt/cloudera/parcels/CDH-6.3.2-1.cdh6.3.2.p0.1605554/lib/hadoop/bin/hadoop job -kil
l job_1606364936355_0003
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0
2020-11-26 16: 21: 20,758 Stage-1 map = 0%, reduce = 0%
2020-11-26 16: 21: 25,864 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 1.17 sec
MapReduce Total cumulative CPU time: 1 seconds 170 msec
Ended Job = job_1606364936355_0003
Stage-4 is selected by condition resolver.
Stage-3 is filtered out by condition resolver.
Stage-5 is filtered out by condition resolver.
Moving data to directory hdfs: //ld-xxxxxxxxxxx/user/hive/warehouse/foo/.hive-staging_hive_2020-11-26
_16-21-12_133_7009525880995260840-1/-ext-10000
Loading data to table default.foo
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Cumulative CPU: 1.17 sec HDFS Read: 5011 HDFS Write: 74 HDFS EC Read:
0 SUCCESS
Total MapReduce CPU Time Spent: 1 seconds 170 msec
OK
Time taken: 15.429 seconds
```

ii. 通过Hive的终端，查询写入的数据，如果正确查询出数据，表示Hive安装完成。

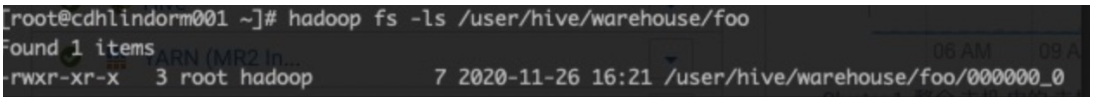
```
hive> select * from foo;
OK
12 xyz
Time taken: 0.091 seconds, Fetched: 1 row(s)
hive>
```

iii. 查看文件是否正常生成 。

a. 通过下面的执行，查询文件信息。

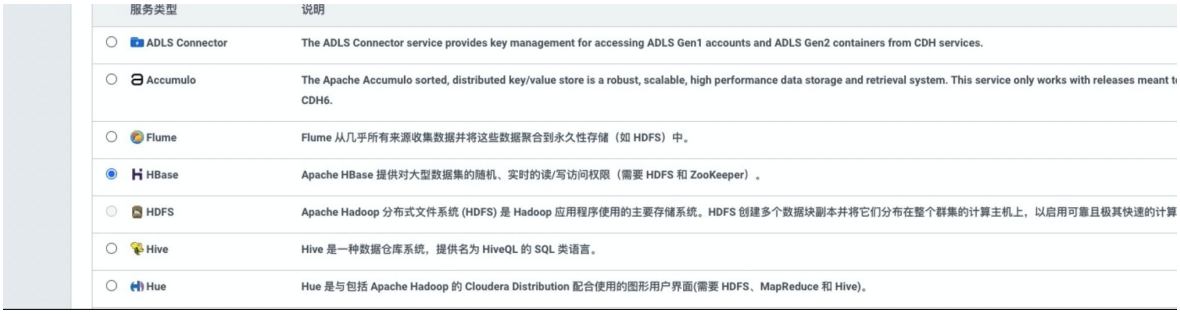
```
[root@cdhlindorm001 ~]# hadoop fs -ls /user/hive/warehouse/foo
```

b. 如果结果如下图，表示 hive 对接文件引擎正常。



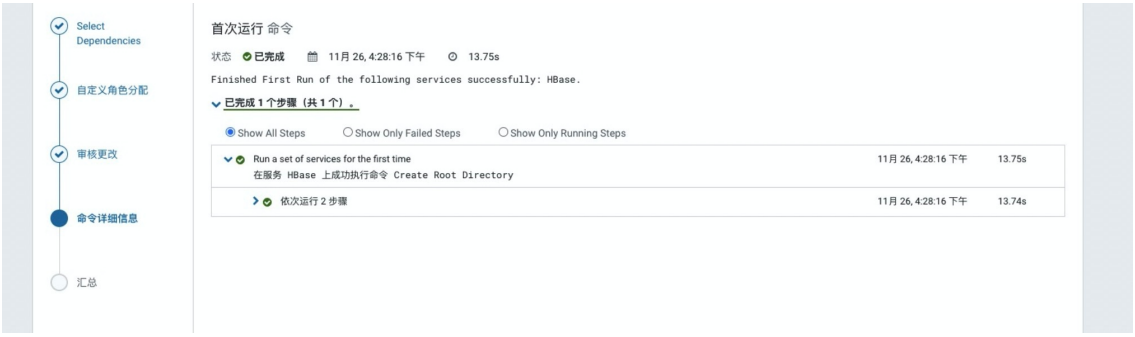
四 、 安装HBASE服务

1. 登录CDH6管理系统，单击集群名称右边的 ，然后单击**添加服务**，进入安装向导界面。

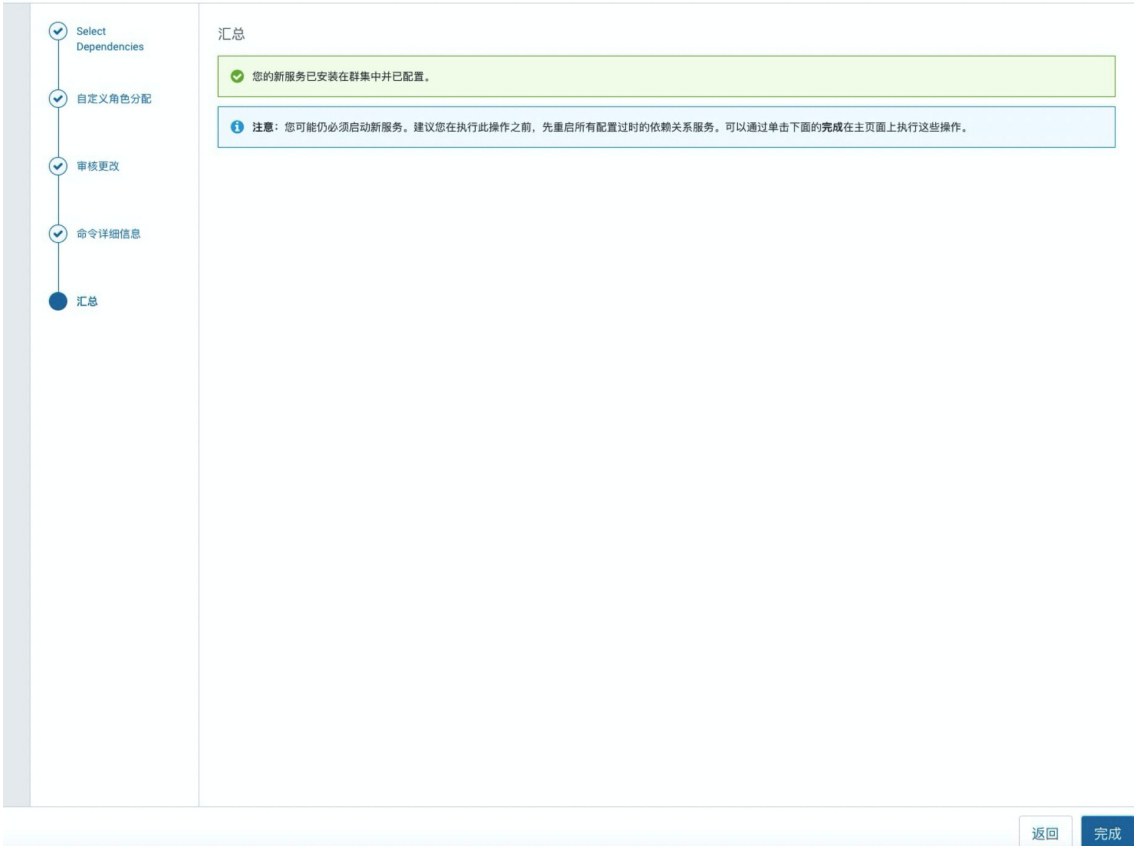


2. 选择HBase服务，单击**继续**，根据**安装指南**配置HBase服务，配置都可以按照默认配置。

i. 开始安装。



ii. 出现下面的截图，表示安装成功，单击完成。



3. 修改 hbase-site.xml 配置。
   i. 在CDH6 系统主页，选择**配置>高级配置代码段**，进入**高级配置代码段**页面。
   ii. 在最上面的搜索框中搜索hbase-site.xml。
   iii. 在**hbase-site.xml 的HBase服务高级配置代码段（安全阀）**和 **hbase-site.xml 的 HBase 客户端高级配置代码段（安全阀）**中，都要添加如下配置项。
      a. 配置项hbase.rootdir，其值：/hbase。
      b. 配置项hbase.unsafe.stream.capability.enforce，其值：false。



      c. 单击**保存更改**。
4. 部署新配置并启动服务。
   i. 返回CDH6管理系统界面，找到**YARN**，单击重新部署图标，进行重新部署。



   ii. 在**过期配置**页面，单击**重启过时服务**。
   iii. 在**重启过时服务**页面，选择需要重启的服务，单击**立即重启**。
   iv. 等待服务全部重启完成，并重新部署客户端配置后，单击**完成**。
5. 验证 HBase 是否运行正常，登入CDH6 任一一台机器执行下面的命令。

i. 执行以下命令进入hbase Shell命令界面。

```
[root@cdhlindorm001 ~]# hbase shell
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.1.0-cdh6.3.2, rUnknown, Fri Nov  8 05:44:07 PST 2019
Took 0.0009 seconds
hbase(main):001:0>
```

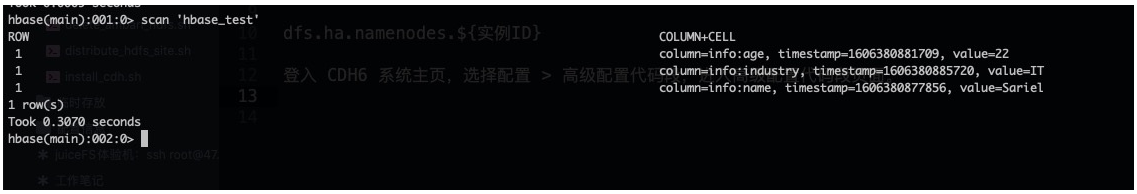
ii. 在HBase中创建测试表，并且写入一些数据。

```
[root@cdhlindorm001 ~]# hbase shell
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.1.0-cdh6.3.2, rUnknown, Fri Nov  8 05:44:07 PST 2019
Took 0.0009 seconds
hbase(main):001:0> create 'hbase_test','info'
Created table hbase_test
Took 1.0815 seconds
=> Hbase::Table - hbase_test
hbase(main):002:0> put 'hbase_test','1', 'info:name' ,'Sariel'
Took 0.1233 seconds
hbase(main):003:0>  put 'hbase_test','1', 'info:age' ,'22'
Took 0.0089 seconds
hbase(main):004:0> put 'hbase_test','1', 'info:industry' ,'IT'
Took 0.0115 seconds
```

iii. 执行如下指令查询刚刚写入的数据，验证Hbase是否正常。

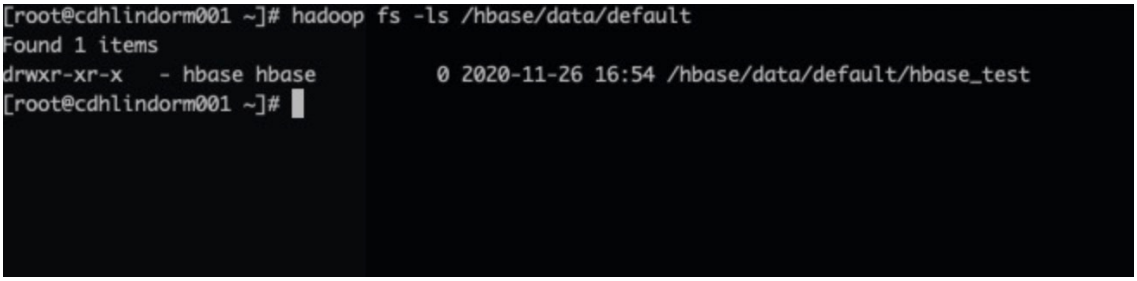
```
hbase(main):005:0> scan 'hbase_test'
```

如果执行结果如下，表示Hbase 服务正常。



iv. 执行以下命令查看文件引擎的/hbase/data/default 路径，如果/hbase/data/default 路径下有hbase_test目录，则证明配置成功。

```
# hadoop fs -ls /hbase/data/default
```



> **注意**
> 如果用户在部署中发现 beeline 或者 hiveserver2 无法正常服务，报不能被代理错误，这是需要开启文件引擎的代理模式， 具体的方式请提交工单或者联系值班 专家服务 协助完成。

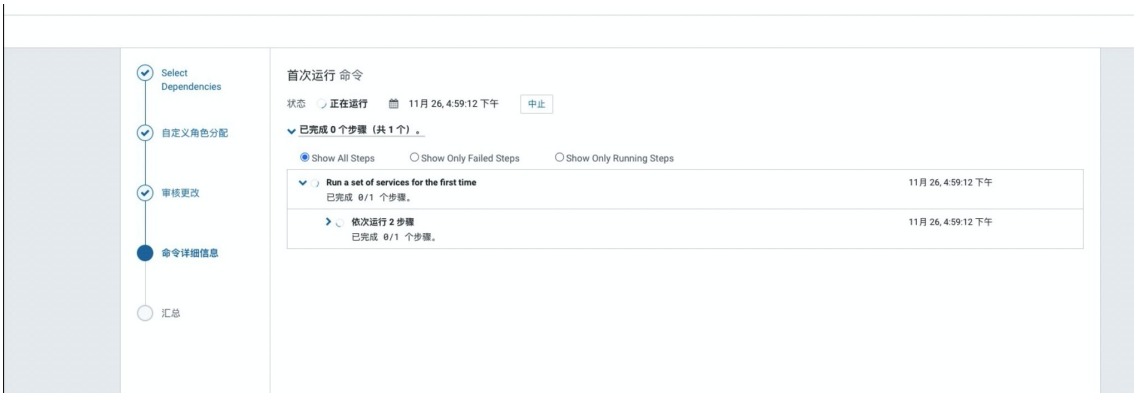
五 、安装Spark服务

1. 登录CDH6管理系统，单击集群名称右边的 ，然后单击**添加服务**，进入安装向导界面。

2. 选择 Spark 服务，单击**继续**， 根据**安装指南**配置 Spark 服务，配置都可以按照默认配置。
   i. 选择依赖， 根据您实际的情况进行选择。



   ii. 开始安装。

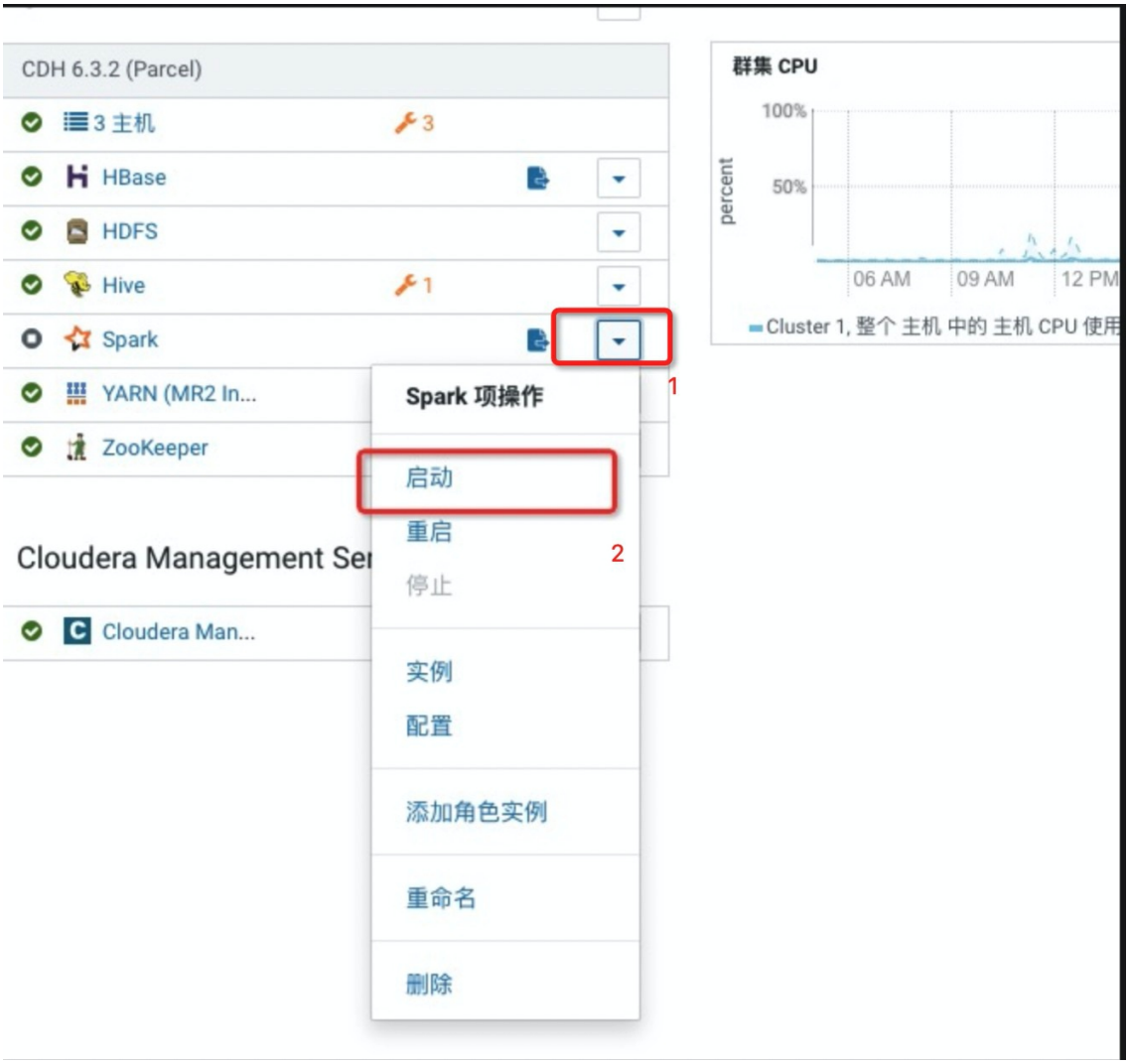


   iii. 安装完成。



3. 启动服务（这是因为安装完成之后，spark并没有启动）。
   i. 返回 CDH6 管理系统界面，找到 **Spark**。

ii. 展开下拉列表，单击**启动**。



4. 验证Spark启动是否成功。

i. 登入CDH 6任意一台机器，使用spark测试包从文件引擎上读取测试文件。

```
[root@cdhlindorm001 ~]# spark-submit   --master yarn --executor-memory 900M --executor-cores 2  --cl
ass org.apache.spark.examples.DFSReadWriteTest  /opt/cloudera/parcels/CDH/jars/spark-examples_2.11-2
.4.0-cdh6.3.2.jar   /etc/profile /output
```

ii. 如果回显信息类似如下图所示，执行成功。

```
20/11/26 17:48:19 INFO cluster.YarnClientSchedulerBackend: Shutting down all executors
20/11/26 17:48:19 INFO cluster.YarnSchedulerBackend$YarnDriverEndpoint: Asking each executor to shut down
20/11/26 17:48:19 INFO cluster.SchedulerExtensionServices: Stopping SchedulerExtensionServices
(serviceOption=None,
 services=List(),
 started=false)
20/11/26 17:48:19 INFO cluster.YarnClientSchedulerBackend: Stopped
20/11/26 17:48:19 INFO spark.MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
20/11/26 17:48:19 INFO memory.MemoryStore: MemoryStore cleared
20/11/26 17:48:19 INFO storage.BlockManager: BlockManager stopped
20/11/26 17:48:19 INFO storage.BlockManagerMaster: BlockManagerMaster stopped
20/11/26 17:48:19 INFO scheduler.OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
20/11/26 17:48:19 INFO spark.SparkContext: Successfully stopped SparkContext
Success! Local Word Count 253 and DFS Word Count 253 agree.
20/11/26 17:48:19 INFO util.ShutdownHookManager: Shutdown hook called
20/11/26 17:48:19 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-3c3bcadf-8098-45ed-a3a5-90d05adc2fc4
20/11/26 17:48:19 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-7adbe838-5c44-44fe-aa34-b13845f847b6
```

iii. 执行下面的指令，查看生成的结果数据。

```
[root@cdhlindorm001 ~]# hadoop fs -ls /output/dfs_read_write_test
```

如果执行结果如下，表示spark 启动没有问题。

- 说明
根据业务场景如果配置复杂可能有些步骤不成功，请议提交工单或者联系值班。

11.计算引擎

11.1. 引擎简介

Lindorm计算引擎是一款基于云原生多模数据库Lindorm核心能力实现的数据库内**高性能、低成本、稳定可靠**的分布式计算服务，满足用户在云原生多模数据库Lindorm支撑场景下的数据生产、交互式分析、机器学习和图计算等场景中的计算需求，支持对海量数据的高并发处理。Lindorm计算引擎完整支持开源Spark计算模型以及编程接口，并深度融合Lindorm存储引擎的特性，充分利用底层数据存储特征以及索引能力，高效地完成分布式计算任务。

功能特性

计算引擎服务提供了如下功能：

- **多种接入方式**：您可以通过JDBC方式对数据进行交互式分析，也可以通过提交JAR包的方式自定义分布式计算任务。
- **开箱即用**：计算引擎与存储引擎（包括宽表引擎、时序引擎、搜索引擎和文件引擎）的访问权限打通，同时避免了复杂的底层组件设置，开发者只需要具备SQL知识和Spark开发经验就可以使用。
- **免运维**：您无需关注集群运维操作（配置，升降级，扩缩容等），只需通过控制台以及SparkUI关注作业管理即可。
- **弹性能力**：计算引擎具有弹性伸缩能力，对于波峰波谷特征明显的业务负载，可以显著降低计算成本，减少闲置资源浪费，并快速应对流量高峰。
- **按量付费**：计算引擎是按照计算资源实际使用量收费，不使用不计费。

11.2. 开通与变配

本文介绍开通和变配计算引擎服务的具体方法。

前提条件

已创建目标Lindorm实例，具体操作请参见创建实例。

计费方式

资源用量根据计算负载可以弹性伸缩，最终账单按照资源实际使用量付费，计费单位为CU，1CU=1核4GB，1CU的单价为0.364元/小时。

开通计算引擎服务

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**数据库连接**。
5. 切换至**计算引擎**页签并单击**立即开通**。



6. 选择**弹性资源上限**。



> **说明** 弹性资源上限用于设置计算引擎资源池的大小，最终账单按照资源实际使用量付费。

7. 阅读并勾选服务协议。
8. 配置完成后，单击**立即购买**。目标Lindorm实例状态会显示**节点扩容中**，请等待5~10分钟。
9. （可选）开通计算引擎服务完成后，您可以根据需求在**计算引擎**页签中单击**开通外网地址**开通公网访问地址。

变配计算引擎资源

计算引擎服务具有弹性伸缩能力，您可以调整弹性资源上限，配置资源池大小。

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在**配置信息**区域，单击计算引擎的**管理 > 更新资源上限**。
5. 根据需求选择**弹性资源上限**。
6. 阅读并勾选服务协议。
7. 配置完成后，单击**立即购买**。目标Lindorm实例状态会显示**节点扩容中**，请等待5~10分钟。

11.3. 快速入门

本文介绍如何快速通过Lindorm计算引擎使用SQL语言对宽表中的数据进行查询分析。

前提条件

- 已创建并开通Lindorm实例的宽表引擎，具体操作请参见创建实例。
- 已开通Lindorm实例的计算引擎服务，具体操作请参见开通与变配。
- 已安装Java环境，要求使用JDK1.8及以上版本。

准备数据

在已开通宽表引擎的Lindorm实例中通过Lindorm-cli连接并执行以下SQL语句，连接方法请参见通过Lindorm-cli连接并使用宽表引擎。

```
CREATE DATABASE test;
USE test;
CREATE TABLE authors (id int, name varchar, country varchar, primary key(id, name));
CREATE TABLE books (id int, title varchar, author_id int, intro varchar, primary key(id, title));
UPSERT INTO authors (id, name, country) values ('222', 'Tom', 'British'), ('333', 'Jerry', 'America');
UPSERT INTO books (id, title, author_id, intro) values (1, 'booka', '333', 'comic'), (2, 'bookb', '333', 'st
ory'), (3, 'bookc', '222', 'story');
```

使用计算引擎查询宽表引擎数据

1. 下载Beeline客户端。
2. 获取连接地址并执行下述命令连接Lindorm计算引擎。

   ```
   ./bin/beeline -u "hostname" -n username -p password
   ```

   参数说明

参数	说明	获取方法
hostname	Lindorm计算引擎的JDBC连接地址。	具体操作，请参见查看连接地址。
username	Lindorm宽表引擎实例的用户名。	
password	用户名的密码。	

3. 连接成功后，执行以下语句查询宽表引擎数据。

```
SELECT a.name, b.title, b.intro FROM lindorm_table.test.authors as a, lindorm_table.test.books as b wher
e a.id = b.author_id and a.name='Jerry';
```

说明 示例中 `SELECT` 语句中的参数值解释如下，具体请参见SELECT语法介绍。

- `lindorm_table`：宽表引擎在计算引擎中的数据源。`lindorm_table` 为默认的Catalog名称。
- `test`：访问宽表引擎的数据库名。
- `books` 和 `authors`：访问的数据所在的表名。

查询结果

```
+-------+-------+-------+
| name  | title | intro |
+-------+-------+-------+
| Jerry | booka | comic |
| Jerry | bookb | story |
+-------+-------+-------+
2 rows selected (1.052 seconds)
```

11.4. 开发指南

11.4.1. 使用SQL访问Lindorm计算引擎

Lindorm计算引擎兼容Hive JDBC驱动，您可以通过JDBC Driver连接计算引擎，然后使用Spark SQL访问Lindorm数据完成数据查询、数据分析和数据可视化等操作。

前提条件

- 已创建并开通Lindorm实例的宽表引擎，具体操作请参见创建实例。
- 已开通Lindorm实例的计算引擎服务，具体操作请参见开通与变配。
- 已安装Java环境，要求使用JDK 1.8及以上版本。

访问方法

使用JDBC Driver连接Lindorm计算引擎有两种方法：

- 通过Beeline客户端连接Lindorm计算引擎
- 通过编写Java程序连接Lindorm计算引擎

Lindorm计算引擎连接地址参数说明

查看Lindorm计算引擎连接地址的具体步骤请参见查看连接地址。



连接地址格式

```
jdbc:hive2://<host>:<port>/;<sessionConfs>?<sparkConfs>#<[spark|hive]Vars>
```

连接地址参数说明

参数	说明
host	Lindorm计算引擎的域名。
port	Lindorm计算引擎的端口号。
sessionConfs	可选参数，由半角分号（;）分隔key=value集合。用于创建相应的JDBC Session连接，仅支持运行时SQL配置。相关内容请参见配置分类。
sparkConfs	可选参数，由半角分号（;）分隔key=value集合。用于创建相应的Spark SQL引擎。如果引擎已存在则忽略。
[spark|hive]Vars	可选参数，由半角分号（;）分隔key=value集合。用于替换Spark或者Hive中的变量。

连接地址示例：通过sparkConfs参数设置Spark SQL引擎的Executor内存大小为8 GB，默认Shuffle分区数为2，连接地址如下：

```
jdbc:hive2://${host}:${port}/;?token=${token};spark.executor.memory=8g;spark.sql.shuffle.partitions=2
```

通过Beeline客户端连接Lindorm计算引擎

常用命令说明如下：

- `!help`：查看帮助命令。
- `!connect`：连接Lindorm计算引擎命令。
- `!exit` 和 `!quit`：退出连接命令。

1. 使用Spark发布包中自带的Beeline客户端，下载Spark发布包。
2. 连接Lindorm计算引擎。
   i. 解压Spark发布包并进入解压目录。

   ```
   cd Spark发布包解压目录
   ```

   ii. 执行以下语句连接Lindorm计算引擎。

   ```
   ./bin/beeline -u "Lindorm计算引擎连接地址" -n 用户名 -p 密码
   ```

   > **说明** 获取Lindorm计算引擎连接地址和用户名密码的方法请参见查看连接地址。

3. 使用SQL对数据进行操作，SQL语法详情请参见计算引擎访问Lindorm宽表引擎的SQL语法介绍。

   示例：使用以下语句查询Lindorm宽表引擎（名为 `lindorm_table` ）默认命名空间（名为 `default` ）的一张数据表（名为 `testtable` ）。

   ```
   SELECT * FROM lindorm_table.default.testtable;
   ```

   查询结果

   ```
   +-----+------+-------+--------+
   | id  | age  | name  | score  |
   +-----+------+-------+--------+
   | 2   | 13   | tom   | 88     |
   | 1   | 12   | jack  | 95     |
   +-----+------+-------+--------+
   ```

通过编写Java程序连接Lindorm计算引擎

1. 您可以通过以下方式获取*hive-jdbc*软件包。关于*hive-jdbc*软件包的版本说明请参见Hive JDBC版本说明。
   - 在pom.xml文件中添加Maven依赖：

```
<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-jdbc</artifactId>
    <version>2.3.8</version>
</dependency>
```

- 在sbt中：

```
libraryDependencies += "org.apache.hive" % "hive-jdbc" % "2.3.8"
```

- 在gradle中：

```
implementation group: 'org.apache.hive', name: 'hive-jdbc', version: '2.3.8'
```

2. 编写Java程序代码，连接Lindorm计算引擎并查询Lindorm宽表引擎数据，代码示例如下：

```
import java.sql.*;
public class App
{
    private static String driverName = "org.apache.hive.jdbc.HiveDriver";
    public static void main(String[] args) throws SQLException {
        try {
            Class.forName(driverName);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Connection con = DriverManager.getConnection(
                "Lindorm计算引擎连接地址", "用户名", "密码");
        Statement stmt = con.createStatement();
        String sql = "select * from lindorm_table.default.testtable";
        ResultSet res = stmt.executeQuery(sql);
        while (res.next()) {
            System.out.println(res.getString(1) + "\t" + res.getString(2));
        }
    }
}
```

> **说明** 如果您需要配置Spark分布式作业，可以设置计算引擎的连接地址参数，具体操作请参见配置方法。

11.4.2. JAR作业开发实践

本文介绍Lindorm计算引擎JAR作业开发的详细步骤，以及如何通过HTTP RESTful API的方式向Lindorm计算引擎提交JAR作业，您可以按照这种方式运行流批任务、机器学习和图计算任务。

Lindorm计算引擎JAR作业开发流程

Lindorm计算引擎JAR作业完全兼容社区版Spark，Spark作业开发示例请参考Spark作业示例。

1. 参数配置。Lindorm计算引擎参数配置完全兼容社区版Spark参数配置，详情请参见作业配置说明。
2. 添加Maven依赖。

   Lindorm计算引擎JAR作业依赖社区版Spark JAR包，版本为3.1.2，将 `scope` 字段配置为 `provided` 。示例如下：

```
<!-- Example -->
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.12</artifactId>
  <version>3.1.2</version>
  <scope>provided</scope>
</dependency>
```

3. 配置以下参数，您就可以在Java程序中通过Spark SQL的方式访问Lindorm宽表引擎数据，相关Spark SQL语法请参见访问Lindorm宽表引擎数据的介绍。

参数	参数值	说明
spark.sql.catalog.lindorm_table	固定值：com.alibaba.lindorm.ldspark.datasources.v2.LindormCatalog。	访问Lindorm宽表引擎的实现类。
spark.sql.catalog.lindorm_table.url	ld-bp1z3506imz2f****-proxy-lindorm.lindorm.rds.aliyuncs.com:30020。	访问Lindorm宽表引擎的地址。请填写Lindorm实例宽表引擎的HBase Java API专有网络访问地址，仅支持访问相同Lindorm实例的宽表引擎。
spark.sql.catalog.lindorm_table.username	默认用户名root。	访问Lindorm宽表引擎的用户名。
spark.sql.catalog.lindorm_table.password	默认密码root。	访问Lindorm宽表引擎的密码。

提交Spark JAR作业流程

1. 获取Spark JAR作业提交入口。Lindorm计算引擎提供RESTful API接口支持JAR作业提交。Spark JAR作业提交入口查看方法请参见查看连接地址。



2. 发送RESTful API请求。
   RESTful API接口说明：
   - 请求路径：*/api/v1/jars/submitJar*
   - 请求方法：POST

   请求参数

参数	参数类型	是否必填	示例值	描述
token	String	是	****	提交JAR作业时用于进行计算资源鉴权的token。您可以通过云原生多模数据库Lindorm控制台获取，单击目标实例ID，在左侧导航栏选择**数据库连接**，切换至**计算引擎**页面并查看token值。
appName	String	是	SparkPi	JAR作业名称，最大不超过46字节。
mainResource	String	是	- JAR包存储至OSS：*oss://testBucketName/path/spark-examples_2.12-3.1.1.jar* - JAR包存储至HDFS：*hdfs:///path/spark-examples_2.12-3.1.1.jar*	JAR包存储的路径，存储至OSS或者HDFS。

参数	参数类型	是否必填	示例值	描述
mainClass	String	是	com.aliyun.ldspark.SparkPi	JAR作业的程序入口类。
args	Array	否	["arg1", "arg2"]	传入mainClass参数。
configs	Json	否	{"spark.executor.memory": "8g"}	JAR作业的其他相关配置。

返回数据

参数	返回信息	说明
200	Jar application($appName) submit successfully. See Alibaba Cloud console with your appName for detail.	JAR作业提交成功。
400	error message	作业提交失败，如有疑问请提交工单。

3. 上传本地JAR包，包括JAR作业程序入口的JAR包（即配置mainResource参数）或者mainResource依赖的本地JAR包（即配置configs参数中的spark.jars）。提交JAR作业之前，请将本地的JAR包上传至HDFS或者OSS，提交JAR作业时需要指定对应的JAR包路径。
   - 上传至HDFS：将JAR包提交至Lindorm计算引擎的LDFS，并提供HDFS协议的访问路径。

     > **说明** 将本地的JAR包上传至HDFS仅支持配置mainResource参数。

   - 上传至OSS：将JAR包提交至OSS，并在configs参数中配置以下信息。配置configs相关参数

参数	示例值	说明
spark.hadoop.fs.oss.endpoint	例如：oss-cn-beijing-internal.aliyuncs.com。	存储JAR包的OSS地址。
spark.hadoop.fs.oss.accessKeyId	testAccessKey ID	通过阿里云控制台创建的Access Key ID和Access Key Secret，获取方法请参见获取AccessKey。
spark.hadoop.fs.oss.accessKeySecret	testAccessKey Secret	
spark.hadoop.fs.oss.impl	固定值：org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem	访问OSS的类。

     > **说明** 更多参数请参见参数说明。

4. 请求示例如下：
   JAR作业上传至HDFS的请求示例：

```
curl --location --request POST 'http://LINDORM_SPARK_ADDR/api/v1/jars/submitJar' \
--header 'Content-Type: application/json' \
--data '{"token": "TOKEN","appName": "APP_NAME","mainClass": "MAIN_CLASS","mainResource": "hdfs:///path/
spark-examples.jar", "args": ["args1", "args2"], "configs": {"spark.executor.memory": "8g"}}'
```

   JAR作业上传至OSS的请求示例：

```
curl --location --request POST 'http://LINDORM_SPARK_ADDR/api/v1/jars/submitJar' \
--header 'Content-Type: application/json' \
--data '{"token": "TOKEN","appName": "APP_NAME","mainClass": "MAIN_CLASS","mainResource": "oss://OSS_ADD
R/myjars/LindormSparkExamples.jar", "args": ["args1", "args2"], "configs":{"spark.hadoop.fs.oss.endpoint
":"OSS_ENDPOINT","spark.hadoop.fs.oss.impl":"org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem","spark
.hadoop.fs.oss.accessKeyId":"testAccessKey ID","spark.hadoop.fs.oss.accessKeySecret":"testAccessKey Secr
et"}}'
```

相关文档

- 通过提交JAR作业可以运行流计算引擎任务，具体操作请参见流式作业开发实践。
- JAR作业提交成功后，可以在SparkUI界面查看相关信息，具体操作请参见SparkUI界面介绍。

11.4.3. Python作业开发实践

Lindorm计算引擎通过HTTP RESTful API的方式提供Spark Python作业提交入口，您可以按照这种方式运行流批任务、机器学习和图计算任务。本文介绍Lindorm计算引擎Python作业开发的详细步骤。

前提条件

已开通Lindorm计算引擎，具体操作请参见开通与变配。

Spark Python作业开发流程

1. 准备Spark Python作业
2. 打包Spark Python作业
3. 上传Spark Python作业
4. 提交Spark Python作业

步骤一：准备Spark Python作业

1. 下载Spark Python作业示例压缩包，下载链接为Spark Python作业示例。
2. 解压Spark Python作业示例压缩包，解压后的目录名称为*lindorm-spark-examples*。打开*lindorm-spark-examples/python*目录，参考*python*目录结构。
3. 项目开发的根目录以*your_project*为例，介绍项目的目录结构。
   i. 在*your_project*目录下新建*__init__.py*文件，内容为空。
   ii. 改造入口文件。
      a. 在入口文件中编写代码，将*your_project*添加到*sys.path*中，代码详情请参见Spark Python作业示例中的*lindorm-spark-examples/python/your_project/main.py*文件的Notice1部分。

```
# Notice1: You need to do the following step to complete the code modification:
# Step1: Please add a "__init__.py" to your project directory, your project will act as a module
of launcher.py
# Step2: Please add current dir to sys.path, you should add the following code to your main file
current_dir = os.path.abspath(os.path.dirname(__file__))
sys.path.append(current_dir)
print("current dir in your_project: %s" % current_dir)
print("sys.path: %s \n" % str(sys.path))
```

      b. 在入口文件中将入口逻辑封装到`main(argv)`方法中。代码详情请参见Spark Python作业示例中的*lindorm-spark-examples/python/your_project/main.py*文件的Notice2部分。

```
# Notice2: Move the code in `if __name__ == "__main__":` branch to a new defined main(argv) func
tion,
# so that launcher.py in parent directory just call main(sys.argv)
def main(argv):
    print("Receive arguments: %s \n" % str(argv))
    print("current dir in main: %s \n" % os.path.abspath(os.path.dirname(__file__)))
    # Write your code here
if __name__ == "__main__":
    main(sys.argv)
```

   iii. 创建Spark Python作业启动的入口文件，用来调用`main(argv)`方法。在根目录*your_project*创建同级目录*launcher.py*，可以复制Spark Python作业示例中的*lindorm-spark-examples/python/launcher.py*文件。

步骤二：打包Spark Python作业

1. 打包项目依赖的Python环境和第三方类库。推荐使用Conda或者Virtualenv将依赖类库打包为tar包，具体操作请参见Python Package Management。

> **注意** 请在Linux环境下完成该操作，以保证Lindorm计算引擎能正常识Python二进制文件。

2. 打包项目文件。将*your_project*文件打包为.zip或者.egg格式文件。
   - 执行以下命令将项目文件打包为.zip格式文件。

     ```
     zip -r project.zip your_project
     ```

   - 将项目文件打包为.egg格式文件，具体操作请参见Building Eggs。

步骤三：上传Spark Python作业

将以下文件都上传至OSS，具体操作请参见上传文件。

- 步骤二中打包的Python环境和第三方类库（也就是tar包）。
- 步骤二中打包的项目文件（也就是.zip或者.egg文件）。
- 步骤一中的*launcher.py*文件。

步骤四：提交Spark Python作业

通过HTTP RESTful API接口提交Spark Python作业，HTTP RESTful API接口说明如下：

- 请求路径：*/api/v1/jars/submitJar*
- 请求方法：POST

请求参数包括以下三个部分：

- 基本参数说明。

参数	参数类型	是否必填	示例值	描述
token	String	是	****	提交Python作业时用于进行计算资源鉴权的token。获取方法请参见查看连接地址。
appName	String	是	pythonKmeans	Python作业名称，最大不超过46字节。
mainResource	String	是	*oss://testBucketName/kmeans.py*	Python文件存储至OSS的路径。
mainClass	String	是	MAIN_CLASS	需要传递任意非空字符串，用来指定Python文件的main函数。
args	Array	否	["arg1", "arg2.2"]	传入mainClass参数。
configs	Json	否	{ "spark.hadoop.fs.oss.impl":"org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem"}	Python作业的其他相关配置。

- Python作业环境参数说明，示例如下：

```
{"spark.archives":"oss://testBucketName/pyspark_conda_env.tar.gz#environment", "spark.kubernetes.driverEnv.PYSPARK_PYTHON":"./environment/bin/python","spark.submit.pyFiles":"oss://testBucketName/your_project.zip"}
```

  - 提交Python作业项目文件（也就是*.zip*、*.egg*或者*.py*格式的文件）时，请配置configs参数中的spark.submit.pyFiles。
  - 提交Python环境和第三方类库（也就是tar包）时，请配置configs参数中的spark.archives和spark.kubernetes.driverEnv.PYSPARK_PYTHON。
    - 配置spark.archives参数时使用井号（#）指定targetDir。
    - 配置spark.kubernetes.driverEnv.PYSPARK_PYTHON参数指定Python文件路径。

- 如果将文件上传至OSS，需要在configs参数中配置以下信息。配置configs相关参数

参数	示例值	说明
spark.hadoop.fs.oss.endpoint	oss-cn-beijing-internal.aliyuncs.com	存储Python文件的OSS地址。
spark.hadoop.fs.oss.accessKeyId	testAccessKey ID	通过阿里云控制台创建的Access Key ID和Access Key Secret，获取方法请参见获取AccessKey。
spark.hadoop.fs.oss.accessKeySecret	testAccessKey Secret	
spark.hadoop.fs.oss.impl	固定值：org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem	访问OSS的类。

> **说明** 更多参数请参见参数说明。

Python作业开发示例

1. 下载并解压Spark Python作业示例。
2. 改造入口文件，打开*your_project/main.py*文件并配置相关代码。
   i. 将*your_project*目录添加到*sys.path*中。

```
current_dir = os.path.abspath(os.path.dirname(__file__))
sys.path.append(current_dir)
print("current dir in your_project: %s" % current_dir)
print("sys.path: %s \n" % str(sys.path))
```

   ii. 在*main.py*文件中添加入口逻辑，例如如下示例初始化SparkSession。

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("PythonImportTest") \
    .getOrCreate()
print(spark.conf)
spark.stop()
```

3. 在Python目录下打包*your_project*文件。

```
zip -r your_project.zip your_project
```

4. 在Linux环境下，使用Conda打包Python运行环境。

```
conda create -y -n pyspark_conda_env -c conda-forge numpy conda-pack
conda activate pyspark_conda_env
conda pack -f -o pyspark_conda_env.tar.gz
```

5. 将打包好的*your_project.zip*和*pyspark_conda_env.tar.gz*上传至OSS，并将*launcher.py*文件上传至OSS。
6. 通过HTTP RESTful API接口提交Python作业示例如下，并传递执行Python作业的参数，访问OSS的参数。

```
curl --location --request POST 'http://LINDORM_SPARK_ADDR/api/v1/jars/submitJar' \
--header 'Content-Type: application/json' \
--data '{"token":"TOKEN","appName":"pythonExample","mainClass":"MAIN_CLASS","mainResource":"oss://testBu
cketName/launcher.py","args":["arg1"],"configs":{"spark.kubernetes.driverEnv.PYSPARK_PYTHON":"./environm
ent/bin/python","spark.submit.pyFiles":"oss://testBucketName/you_project.zip","spark.archives":"oss://te
stBucketName/pyspark_conda_env.tar.gz#environment","spark.hadoop.fs.oss.endpoint":"OSS_ENDPOINT","spark.
hadoop.fs.oss.impl":"org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem","spark.hadoop.fs.oss.accessKey
Id":"testAccessKey ID","spark.hadoop.fs.oss.accessKeySecret":"testAccessKey Secret"}}'
```

作业诊断

Python作业提交成功后，可以在**作业列表**页面查看作业运行状况和SparkUI访问地址，具体操作请参见查看Spark作业。

11.4.4. 流式作业开发实践

本文介绍如何基于Lindorm计算引擎实例提交一个流式作业（Spark Streaming），流式作业在执行中将持续从消息队列Kafka读取消息，同时统计其中的单词数量，并将结果写回消息队列Kafka。

前提条件

- 已部署Kafka。
  - 如果您需要部署Apache Kafka，具体操作请参见快速部署Kafka。
  - 如果您需要部署阿里云消息队列Kafka，具体操作请参见部署操作流程。
- 已开通OSS服务，具体操作请参见开通OSS服务。

操作步骤

1. 创建两个Kafka Topic，名称分别为*input-lines*和*word-count-results*。
   - 如果您使用的是Apache Kafka，创建命令请参考如下：

     ```
     bin/kafka-topics.sh --create --topic input-lines --bootstrap-server 192.168.XX.XX:9092
     bin/kafka-topics.sh --create --topic word-count-results --bootstrap-server 192.168.XX.XX:9092
     ```

   - 如果您使用的是阿里云消息队列Kafka，具体操作请参见创建Topic。
2. 下载JAR包，并上传至OSS服务，具体操作请参见上传文件。
3. 执行以下命令启动Spark Streaming作业。

   ```
   curl --location --request POST 'JAR作业提交入口' \
   --header 'Content-Type: application/json' \
   --data '{"token": "计算引擎实例的Token","appName": "spark-streaming-kafka-test","mainClass": "com.aliyun.lindorm.ldspark.examples.StructuredKafkaWordCountResultBack","mainResource": "oss://JAR包上传至OSS的路径",
   "args": ["Kafka的默认接入点", "subscribe", "input-lines"], "configs": {"spark.hadoop.fs.oss.endpoint": "JAR作业OSS对应的endpoint", "spark.hadoop.fs.oss.impl": "org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem", "spark.hadoop.fs.oss.accessKeyId": "访问OSS的Access Key ID", "spark.hadoop.fs.oss.accessKeySecret": "访问OSS的Access Key Secret", "spark.hadoop.fs.oss.buffer.dir": "/tmp/oss"}}'
   ```

   > **说明** 上述命令中的参数获取方法请参见JAR作业开发实践。

4. Spark Streaming作业启动完成后，执行以下命令向Topic（名称为*input-lines*）中写数据。

   ```
   bin/kafka-console-producer.sh --topic input-lines --bootstrap-server 'Kafka的默认接入点'
   one two three two three three  //需要写入的数据
   ```

   > **说明** 如果您使用的阿里云消息队列Kafka，可以在消息队列Kafka版控制台的**实例详情**页面的**接入点信息**区域获取Kafka的默认接入点。

5. 执行以下命令统计单词数量。

   ```
   bin/kafka-console-consumer.sh --topic word-count-results --from-beginning --bootstrap-server '您的Kafka bootstrap-servers' --property print.key=true --property key.separator=":  "
   ```

   统计结果显示

   ```
   two:  2
   one:  1
   three:  3
   ```

通过Java代码测试流计算引擎作业示例

在*StructuredKafkaWordCountResultBack.scala*文件中执行以下代码，对应的Maven工程示例请参见Maven工程示例。

```
package com.aliyun.lindorm.ldspark.examples
import org.apache.spark.sql.SparkSession
import java.util.UUID
/**
 * Consumes messages from one or more topics in Kafka and does wordcount,
```

```
 * then write back the result to kafka.
 * Usage: StructuredKafkaWordCountResultBack <bootstrap-servers> <subscribe-type> <topics>
 *     [<checkpoint-location>]
 *   <bootstrap-servers> The Kafka "bootstrap.servers" configuration. A
 *   comma-separated list of host:port.
 *   <subscribe-type> There are three kinds of type, i.e. 'assign', 'subscribe',
 *   'subscribePattern'.
 *   |- <assign> Specific TopicPartitions to consume. Json string
 *   |  {"topicA":[0,1],"topicB":[2,4]}.
 *   |- <subscribe> The topic list to subscribe. A comma-separated list of
 *   |  topics.
 *   |- <subscribePattern> The pattern used to subscribe to topic(s).
 *   |  Java regex string.
 *   |- Only one of "assign, "subscribe" or "subscribePattern" options can be
 *   |  specified for Kafka source.
 *   <topics> Different value format depends on the value of 'subscribe-type'.
 *   <checkpoint-location> Directory in which to create checkpoints. If not
 *   provided, defaults to a randomized directory in /tmp.
 *
 * Example:
 *    `$ bin/run-example \
 *      sql.streaming.StructuredKafkaWordCountResultBack host1:port1,host2:port2 \
 *      subscribe topic1,topic2`
 */
object StructuredKafkaWordCountResultBack {
  def main(args: Array[String]): Unit = {
    if (args.length < 3) {
      System.err.println("Usage: StructuredKafkaWordCountResultBack <bootstrap-servers> " +
        "<subscribe-type> <topics> [<checkpoint-location>]")
      System.exit(1)
    }
    val Array(bootstrapServers, subscribeType, topics, _*) = args
    val checkpointLocation =
      if (args.length > 3) args(3) else "/tmp/temporary-" + UUID.randomUUID.toString
    val spark = SparkSession
      .builder
      .appName("StructuredKafkaWordCountResultBack")
      .getOrCreate()
    import spark.implicits
    // Create DataSet representing the stream of input lines from kafka
    val lines = spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option(subscribeType, topics)
      .load()
      .selectExpr("CAST(value AS STRING)")
      .as[String]
    // Generate running word count
    val wordCounts = lines.flatMap(_.split(" ")).groupBy("value").count()
    // Start running the query that writes back the result to kafka
    val query = wordCounts
      .selectExpr(
        "cast(value as string) as key",
        "cast(count as string) as value")
      .writeStream
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option("topic", "word-count-results")
      .outputMode("complete")
      .option("checkpointLocation", checkpointLocation)
      .start()
    query.awaitTermination()
  }
```

```
}
```

11.4.5. 作业配置说明

本文介绍Lindorm计算引擎作业提供的配置项和配置方法。

Spark配置项说明

Lindorm计算引擎Spark配置项支持开源Spark配置项，更多信息请参见Spark Configuration。

Lindorm计算引擎包括部分受限配置项和新增自定义配置项，说明如下：

- 以下配置项为受限配置项，不允许用户自定义。

配置项	说明
spark.master	需要连接的Spark集群管理器，Lindorm计算引擎会根据Lindorm实例自动设置。
spark.submit.deployMode	Spark驱动程序的部署方式。

- 以下配置项为新增自定义配置项，允许用户自定义。

配置项	说明
spark.executor.resourceTag	Lindorm计算引擎支持通过该配置项指定具体的内存资源配置，从而简化Executor资源规格配置，配置值和说明如下： • **"large-mem"**：适用于内存密集型作业，Executor规格配置如下： spark.executor.cores=2 spark.executor.memory=11264m spark.executor.memoryOverhead=5120m • **"xlarge-mem"**：适用于内存超密集型作业，Executor规格配置如下： spark.executor.cores=1 spark.executor.memory=11264m spark.executor.memoryOverhead=5120m • **"large-memory-overhead"**：适用于堆外内存开销（例如pyspark）较大的作业，Executor规格配置如下： spark.executor.cores=2 spark.executor.memory=8192m spark.executor.memoryOverhead=8192m • **"xlarge-memory-overhead"**：适用于堆外内存开销（例如pyspark）超大的作业，Executor规格配置如下： spark.executor.cores=1 spark.executor.memory=8192m spark.executor.memoryOverhead=8192m
spark.dfsLog.executor.enabled	指定是否将Executor日志保存至DFS，默认值为true。Lindorm实例的计算引擎服务规模较大的情况下，通过指定该参数为false可以避免日志流增大导致DFS压力过大。

配置分类

按照Spark SQL配置生效机制可以将配置分为两种：

- 运行时SQL配置：配置完成后，运行时SQL配置都会生效。更多运行时SQL配置请参见Runtime SQL Configuration。
- 静态SQL配置和其他Spark Core配置：例如配置spark.executor.memory，如果创建新的Spark SQL引擎，配置会生效。否则，配置会无效。更多静态SQL配置请参见Static SQL Configuration。

配置方法

- 配置以JDBC方式提交的作业
  - 通过SET语法设置配置项，配置方法请参见SET。
  - 通过JDBC连接地址设置配置项，JDBC连接地址说明请参见使用SQL访问Lindorm计算引擎。
    例如：通过JDBC连接串设置Spark的Shuffle默认分区数为2，Executor使用的内存量为8 GB。

```
jdbc:hive2://${host}:${port}/;?token=${token};spark.executor.memory=8g;spark.sql.shuffle.partitions=2
```

- 通过连接属性设置配置项，通过Java代码设置连接属性，为指定Spark作业设置配置项。
  例如：通过Java代码设置Spark的Shuffle默认的分区数为2，Executor使用的内存量为8 GB。

```
Class.forName("org.apache.hive.jdbc.HiveDriver");
Properties info = new Properties();
info.setProperty("user", ${username})
info.setProperty("password", ${password})
...
// 这里传递一组hiveconfs
info.setProperty("hiveconf:spark.sql.shuffle.partitions", "2");
info.setProperty("hiveconf:spark.executor.memory", "8g");
...
String JDBC_URL="jdbc:hive2://${host}:${port}?token=${token}";
Connection conn = DriverManager.getConnection(JDBC_URL, info);
HiveStatement stat = (HiveStatement) conn.createStatement();
...
```

- 配置以JAR方式提交的作业
  在提交Spark作业时，通过指定的参数，为指定Spark作业设置配置项。
  例如：通过JAR方式设置Spark的Shuffle默认的分区数为2，Executor使用的内存量为8 GB。

```
curl --location --request POST 'http://LINDORM_SPARK_ADDR/api/v1/jars/submitJar' \
--header 'Content-Type: application/json' \
--data '{"token": "TOKEN","appName": "APP_NAME","mainClass": "MAIN_CLASS","mainResource": "hdfs:///path/spark-examples.jar", "configs": {"spark.ui.port": 4088, "spark.sql.shuffle.partitions": "2", "spark.executor.memory": "8g"}}'
```

11.4.6. 本地运行并调试Spark作业

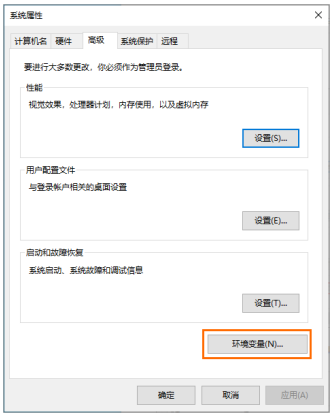
本文介绍Spark作业开发过程中，在本地上运行Spark作业并进行调试。

前提条件

- 已开通云原生多模数据库Lindorm计算引擎的外网地址。
- 已将本地IP地址添加至Lindorm实例的白名单，具体操作请参见设置白名单。
- 已准备Spark作业项目。

本地运行Spark作业

1. 已下载最新版Lindorm计算引擎环境的压缩包，下载链接为环境压缩包。
2. 解压下载的压缩包，解压的路径可以自定义。
3. 在本地配置计算引擎的环境变量，将解压路径设置为SPARK_HOME环境变量。
   - Windows系统配置计算引擎的环境变量步骤如下：
     a. 打开本地的**系统属性**页面并单击**环境变量**。



     b. 在**环境变量**窗口，单击**系统变量**区域下的**新建**。

c. 在**新建系统变量**窗口，输入以下参数。
   - **变量名**：填写SPARK_HOME。
   - **变量值**：输入压缩包解压的路径。

d. 单击**确定**。

e. 单击**应用**。

- Linux系统配置计算引擎的环境变量，请执行`export SPARK_HOME="<压缩包解压的路径>"`命令并将该命令添加至`~/.bashrc`中。

4. 打包Spark作业项目，并使用$SPARK_HOME/bin/spark-submit提交Spark作业。作业提交示例如下。

   i. 以Spark作业示例为例，下载并解压该项目。

   ii. 配置以下参数：

参数	参数值	说明
spark.sql.catalog.lindorm_table.url	ld-bp1z3506imz2f****-proxy-lindorm-pub.lindorm.rds.aliyuncs.com:30020。	访问Lindorm宽表引擎的公网地址。请填写Lindorm实例宽表引擎的HBase Java API公网访问地址，仅支持相同Lindorm实例的宽表引擎访问。
spark.sql.catalog.lindorm_table.username	默认用户名root。	访问Lindorm宽表引擎的用户名。
spark.sql.catalog.lindorm_table.password	默认密码root。	访问Lindorm宽表引擎的密码。

```
$SPARK_HOME/bin/spark-submit \
# 您可以通过使用 --jars 添加作业依赖jar包，更多参数请参考 spark-submit -h
--class com.aliyun.lindorm.ldspark.examples.LindormSparkSQLExample \
lindorm-spark-examples/target/lindorm-spark-examples-1.0-SNAPSHOT.jar
```

> **说明** 提交Spark作业项目时如果不指定运行方式，默认是在本地上运行，也可以通过spark.master=local[*]参数指定。

   iii. 根据SQL代码中涉及的Schema，创建对应的库表结构。

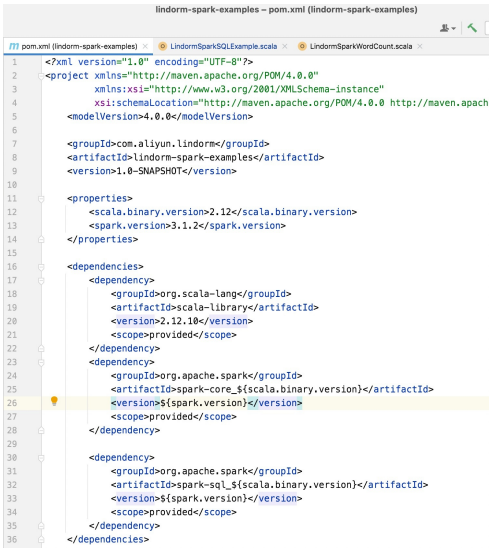
   iv. 使用`mvn clean package`进行作业打包。

5. 完成作业的本地开发后，可以通过提交JAR作业的方式将Spark作业提交至云上运行，具体操作请参见Lindorm计算引擎JAR作业开发流程，请将Spark作业中使用的连接地址修改为Lindorm计算引擎的专有网络连接地址。

调试Spark作业

以Spark作业示例为例，通过IDEA软件调试Spark作业。IDEA软件下载地址为官网IDEA。

1. 执行步骤1至步骤3。
2. 打开IDEA软件，配置*pom.xml*文件中Spark相关依赖为`<scope>provided</scope>`。



3. 将*$SPARK_HOME/jars*添加至项目依赖中。
   i. 在IDEA软件顶部菜单栏，选择**File > Project Structure**。
   ii. 在左侧导航栏，选择**Project Settings > Libraries**，单击+添加Java类库。
   iii. 选择*$SPARK_HOME/jars*。
   iv. 单击**OK**。
4. 运行Spark作业程序。作业运行过程中，您可以根据日志中指定的链接地址查看SparkUI。

```
2022/01/14 15:27:58 INFO SparkUI:Bound SparkUI to 0.0.0.0,and started at http://30.25.XX.XX:4040
```

11.5. 数据访问

11.5.1. 使用须知

本文介绍使用Lindorm计算引擎SQL语句访问数据库的数据时，需要提前了解的相关内容。

了解开源Spark SQL

通过开源Spark SQL访问数据库的数据分为以下三个级别：

- Catalog：用来标识用户不同的数据源。
- Namespace：与数据库中的Database或者Schema相对应。
- Table：与数据库中的表相对应。

了解Lindorm计算引擎SQL

根据您已开通的引擎类型（即宽表引擎、时序引擎、搜索引擎或者文件引擎等），Lindorm计算引擎为您新增对应的数据源（Catalog）。您可以使用`SHOW CATALOGS`语句来查看当前可以使用的数据源，数据源名称和说明如下：

数据源名称	说明	语法规则参考
`spark_catalog`	计算引擎默认提供的数据源。建议使用Hive Metastore来管理数据源的元数据，具体请参见使用Hive Metastore管理计算引擎元信息。	Spark SQL
`lindorm_table`	读写Lindorm宽表引擎使用的数据源。	访问Lindorm宽表引擎数据的介绍
`lindorm_cdc`	读写Lindorm CDC数据使用的数据源。	访问Lindorm CDC数据的介绍

访问语法示例：Lindorm宽表引擎（lindorm_table）中有一个Database名称为DB1，DB1中有一个表名为tableX，可以通过以下两种方式查询表的数据。

- 示例一

```
SELECT fieldA FROM lindorm_table.DB1.tableX;
```

- 示例二

```
USE lindorm_table.DB1;
SELECT fieldA FROM tableX;
```

11.5.2. 使用Hive Metastore管理计算引擎元信息

云原生多模数据库Lindorm计算引擎支持使用Hive Metastore服务来管理库表结构的元信息，从而提供Hive数仓能力，可以存取文件引擎中多种格式的数据，例如Parquet、ORC、CSV等。本文介绍使用Hive Metastore服务管理计算引擎元信息的具体方法和示例。

前提条件

云原生多模数据库Lindorm计算引擎提供三种方式使用Hive Metastore服务来管理库表结构的元信息。不同方式使用的前提条件如下表：

使用方式	前提条件
（推荐）使用内置的Hive Metastore服务	已开通Lindorm计算引擎的内置Hive Metastore服务。
使用外置的Hive Metastore服务	• 已准备Hive Metastore服务，详情请参见Metastore服务介绍。 • 已将Lindorm实例的专有网络交换机网段添加至Hive Metastore服务的白名单中，确保Lindorm计算引擎可以访问外置的Hive Metastore服务。
使用计算引擎内置的Hive Metastore服务连接MySQL数据库	• 已购买云数据库MySQL服务或者自建MySQL数据库，云数据库MySQL服务包括云原生关系型数据库PolarDB MySQL引擎和云数据库RDS MySQL。 • 已创建MySQL数据库账号和密码。 • 已将Lindorm实例的专有网络交换机网段添加至MySQL数据库的白名单中，确保MySQL数据库可以访问计算引擎内置的Hive Metastore服务。

使用内置的Hive Metastore服务

使用内置的Hive Metastore服务来管理库表结构的元数据，仅需要在云原生多模数据库Lindorm控制台开通计算引擎的内置Hive Metastore服务，然后在提交计算引擎作业时直接使用Hive Metastore服务。

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**计算引擎 > 服务管理**。
5. 在**Hive Metastore**页签中单击**立即开通**。



使用外置的Hive Metastore服务

Lindorm计算引擎作业开发中，需要添加以下配置项。

配置项	示例值	说明
spark.hive.metastore.uris	thrift://HMS_IP:HMS_PORT	Hive Metastore连接地址。

示例如下：

```
bin/beeline -u "jdbc:hive2://HOST:PORT/?token=TOKEN;spark.hive.metastore.uris=thrift://HMS_IP:HMS_PORT" -n U
SERNAME -p PASSWORD
```

使用计算引擎内置的Hive Metastore服务连接MySQL数据库

1. 初始化MySQL数据库Schema。
   i. 下载并解压Hadoop 2.10.x的最新版本，下载链接为Hadoop。
   ii. 配置HADOOP_HOME的环境变量，具体操作请参见环境变量配置。
   iii. 下载并解压Hive 2.3.x的最新版本，下载链接为Hive。
   iv. 配置HIVE_HOME的环境变量，具体操作请参见环境变量配置。
   v. 下载*mysql-connector-java-8.0.27.jar*，下载链接为JAR包，并将该JAR包移动至*$HIVE_HOME/lib*目录下。
   vi. 在*$HIVE_HOME/conf*目录下新建*hive-site.xml*文件，并添加如下配置。

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://MYSQL_ADDR:3306/Database</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>com.mysql.cj.jdbc.Driver</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionUserName</name>
    <value>USERNAME</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionPassword</name>
    <value>PASSWORD</value>
  </property>
</configuration>
```

   vii. 执行以下命令初始化MySQL数据库Schema。

```
$HIVE_HOME/bin/schematool -dbType mysql -initSchema
```

2. 提交Lindorm计算引擎作业。

   Lindorm计算引擎作业开发中，需要添加以下配置项。

配置项	示例值	说明
spark.hadoop.javax.jdo.option.ConnectionURL	jdbc:mysql://MYSQL_ADDR:3306/Database	MySQL数据库的JDBC连接地址。例如：云原生关系型数据库PolarDB MySQL引擎，MYSQL_ADDR可以通过云数据库PolarDB控制台获取，在MySQL集群的**基本信息**页查看PolarDB MySQL引擎的链接地址。
spark.hadoop.javax.jdo.option.ConnectionDriverName	固定值：com.mysql.cj.jdbc.Driver	JDBC DriverName。
spark.hadoop.javax.jdo.option.ConnectionUserName	username	MySQL数据库的用户名。
spark.hadoop.javax.jdo.option.ConnectionPassword	password	MySQL数据库用户名的密码。

示例如下：

```
bin/beeline -u "jdbc:hive2://HOST:PORT/?token=TOKEN;spark.hadoop.javax.jdo.option.ConnectionURL=jdbc:mysql:/
/rm-xxx.mysql.rds.aliyuncs.com:3306/hms;spark.hadoop.javax.jdo.option.ConnectionDriverName=
com.mysql.cj.jdbc.Driver;spark.hadoop.javax.jdo.option.ConnectionUserName=MYSQL_USERNAME;spark.hadoop.javax.
jdo.option.ConnectionPassword=MYSQL_PASSWORD" -n USERNAME -p PASSWORD
```

11.5.3. 访问Lindorm宽表引擎数据的介绍

本文介绍Lindorm计算引擎访问宽表引擎数据（包括关系表和HBase表）的相关内容。

表的类型

Lindorm计算引擎访问宽表引擎支持两种类型的表：

- 关系表：表示通过Lindorm宽表SQL进行读写的表。
- HBase表：表示通过HBase客户端进行读写的表。

Lindorm计算引擎访问宽表引擎数据（包括关系表和HBase表）的相关内容如下：

内容	关系表	HBase表
访问方式	使用SQL直接访问。	需要先为HBase表设置LINDORM_HBASE_CATALOG属性，设置方法请参见为HBase表设置表属性。
支持的数据类型	支持的数据类型。	
支持的语法	支持的语法。	

为HBase表设置表属性

Lindorm计算引擎访问宽表引擎中的HBase表时，需要先为HBase表设置LINDORM_HBASE_CATALOG属性，该属性表示SparkSQL Schema与HBase表Schema的对应关系。LINDORM_HBASE_CATALOG属性值为一个JSON，如下所示：

> **说明** LINDORM_HBASE_CATALOG属性中所有的字段和值都区分大小写。

```
{
  "table": {"namespace": "hbase", "name": "table1"},
  "rowkey": "hbase_rowkey1:hbase_rowkey2",
  "columns": {
    "SQL_PK1": {"cf": "rowkey", "col": "hbase_rowkey1", "type": "int"},
    "SQL_PK2": {"cf": "rowkey", "col": "hbase_rowkey2", "type": "string"},
    "SQL_COL1": {"cf": "cf", "col": "hbase_qualifier1", "type": "short"},
    "SQL_COL2": {"cf": "cf", "col": "hbase_qualifier2", "type": "string"}
  }
}
```

参数	示例	描述
table	"namespace": "hbase", "name": "table1"	HBase表的Namespace和表名称。 • namespace：HBase表的Namespace。 • name：HBase表的表名称。
rowkey	hbase_rowkey1:hbase_rowkey2表示rowkey由hbase_rowkey1和hbase_rowkey2组成，那么columns中需要有两个字段的 `"cf:rowkey"` ，并且col字段分别为hbase_rowkey1和hbase_rowkey2。	HBase表的rowkey字段组成，需要与columns参数中 `"cf:rowkey"` 的字段 `col` 定义保持一致。

参数	示例	描述
columns	"SQL_PK1": {"cf": "rowkey", "col": "hbase_rowkey1", "type": "int"}	SQL字段与HBase表字段的对应信息，每个SQL字段的定义由字段名称，对应的 HBase字段及类型组成。 • SQL_PK1：字段名称，可以自定义，使用SparkSQL访问时使用该字段名称。 • cf和col：SQL字段对应的HBase columnFamily和qualifier，其中rowkey字段的cf需要定义为rowkey，rowkey由多个字段构成时，string或者binary只能为最后一个字段。 • type：SQL字段对应的类型，SparkSQL根据type定义来进行字段解析，支持的字段类型请参考支持的数据类型。

为HBase表设置LINDORM_HBASE_CATALOG属性有两种方法：

- 通过HBase Shell方式：

```
alter 'hbase:table1', METADATA => { 'LINDORM_HBASE_CATALOG' => '<CATALOG_JSON>' }
```

- 通过HBase Java API方式：

```
HTableDescriptor tableDescriptor = admin.getTableDescriptor(TableName.valueOf(tableName));
tableDescriptor.setValue("LINDORM_HBASE_CATALOG", catalogJSON);
admin.modifyTable(tableDescriptor.build());
```

示例

HBase表的Namespace为hbase，表名称为table1。该表的rowkey字段由一个string组成，有一个columnFamily名为 cf，该columnFamily中有两个qualifier：col1（int类型）和col2（string类型）。通过Lindorm计算引擎访问该HBase表，使用pk字段对应rowkey字段，col1对应 `"cf:col1"` 字段，col2对应 `"cf:col2"` 字段。

1. HBase表的LINDORM_HBASE_CATALOG属性值如下：

```
{
  "table": {"namespace": "hbase", "name": "table1"},
  "rowkey": "rk1",
  "columns": {
    "pk": {"cf": "rowkey", "col": "rk1", "type": "string"},
    "col1": {"cf": "cf", "col": "col1", "type": "int"},
    "col2": {"cf": "cf", "col": "col2", "type": "string"}
  }
}
```

2. 使用HBase Shell设置表属性。

```
ALTER 'hbase:table1', METADATA => { 'LINDORM_HBASE_CATALOG' => '<CATALOG_JSON>' }
```

3. 通过JDBC方式连接Lindorm计算引擎，具体操作请参见使用SQL访问Lindorm计算引擎。
4. 执行以下语句查询HBase表的数据。

```
SELECT pk, col1 FROM lindorm_table.hbase.table1 LIMIT 5;
```

支持的数据类型

Lindorm计算引擎访问宽表引擎数据时，支持的数据类型如下表：

> **说明**
> - 关于开源Spark语法的数据类型介绍请参见Spark数据类型。
> - 下表中的短划线（-）表示不支持该数据类型。

关系表的数据类型	HBase表的数据类型	Spark数据类型	描述
BOOLEAN	boolean	BooleanType	长度为1字节。定长精确数值类型。值域取决于长度，以及是否为无符号。0表示false，1表示true。

关系表的数据类型	HBase表的数据类型	Spark数据类型	描述
BYTE	byte	ByteType	长度为1字节。定长精确数值类型。值域取决于长度，以及是否为无符号。
SHORT	short	ShortType	长度为2字节。定长精确数值类型。值域取决于长度，以及是否为无符号。
INT	int	IntegerType	长度为4字节。定长精确数值类型。值域取决于长度，以及是否为无符号。
LONG	long	LongType	长度为8字节。定长精确数值类型。值域取决于长度，以及是否为无符号。
FLOAT	float	FloatType	长度为4字节。定长非精确数值类型。值域和精度取决于长度、precision和scale，以及是否为无符号。
DOUBLE	double	DoubleType	长度为8字节。定长非精确数值类型。值域和精度取决于长度、precision和scale，以及是否为无符号。
STRING	string	StringType	变长二进制类型，支持中文。
VARBINARY	-	BinaryType	变长二进制类型，作为主键时只能是最后一列主键。
BINARY(N)	binary	BinaryType	定长二进制类型，N表示字节数。写入数据时，包括两种情况。 • 若给出的值不足N字节，系统会自动在末尾将缺的字节用0补充。 • 若超出N字节，则自动截断。
DATE	-	DateType	日期时间（仅存储日期，不存储时间，在时区转换过程中容易出现日期不对的现象，不推荐使用）。
TIMESTAMP	-	TimestampType	时间戳。Lindorm默认使用GMT时间（UTC+0时区）读取数据，与中国时区相差8小时。

支持的语法

Lindorm计算引擎访问宽表引擎数据支持的语法包括DDL和DQL。

DDL

- USE/SHOW/DESCRIBE

语句	描述	语法
SHOW DATABASES	查看所有的Database。	`SHOW { DATABASES | SCHEMAS } [ LIKE regex_pattern ]`
USE DATABASE	使用某个Database。	`USE database_name`
SHOW TABLES	查看所有的Table。	`SHOW TABLES [ { FROM | IN } database_name ] [ LIKE regex_pattern ]`
DESCRIBE TABLE	查看某个表的详细信息。	`{ DESC | DESCRIBE } [ TABLE ] table_identifier`

- INSERT

语句	描述	语法
INSERT	在表中插入数据。	`INSERT INTO [ TABLE ] table_identifier [ partition_spec ] [ ( column_list ) ] { VALUES ( { value | NULL } [ , ... ] ) [ , ( ... ) ] | query }` Lindorm计算引擎支持SparkSQL `INSERT INTO` 语法，具体使用方法请参见INSERT使用方法。

DQL

Lindorm计算引擎访问宽表引擎数据，仅支持SELECT语句。

语法

```
select_statement [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select_statement, ... ]
    [ ORDER BY { expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [ , ... ] } ]
    [ SORT BY { expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [ , ... ] } ]
    [ LIMIT { ALL | expression } ]
```

select_statement定义如下：

```
SELECT [ ALL | DISTINCT ] named_expression
    FROM { from_item [ , ... ] }
    [ WHERE boolean_expression ]
    [ GROUP BY expression [ , ... ] ]
    [ HAVING boolean_expression ]
```

参数说明

SELECT语法的参数说明请参见参数说明。其中 `from_item` 参数支持的表达式和示例如下：

- Table relation: `table_name`
- Subquery: `SELECT * FROM tableX WHERE count > 20`
- Join relation: `tableA LEFT JOIN tableB ON tableA.deptno=tableB.deptno`

示例

```
SELECT * FROM student WHERE age > 12;
SELECT * FROM score WHERE exam_round = 1 and subject = 'Chinese' ORDER BY score LIMIT 5;
SELECT student.school_id, avg(score.score) AS avg_score FROM student INNER JOIN score ON score.student_id =
student.id GROUP BY student.school_id ORDER BY avg_score desc
```

11.5.4. 访问Lindorm CDC数据的介绍

云原生多模数据库Lindorm计算引擎提供Lindorm CDC（Change Data Capture）数据源，您可以通过该数据源使用Spark SQL查询Lindorm实例其他引擎变更的数据。

前提条件

- Lindorm实例已开通LTS（Lindorm Tunnel Service）服务，具体请参见开通并登录LTS。
- Lindorm宽表已创建数据订阅通道，创建方法请参见通过Pull模式创建数据订阅通道。

> **说明** 创建数据订阅通道时需要注意以下几点：
> - 不要勾选**在投递的数据内容中为列名省略family前缀**。
> - **数据序列化格式**中请选择**json**。
> - 一个**主题名**仅对应一个Lindorm表名。

- 为HBase表设置LINDORM_HBASE_CATALOG属性，设置方法请参见为HBase表设置表属性。

> **说明** LINDORM_HBASE_CATALOG属性表示SparkSQL Schema与HBase表Schema的对应关系，Lindorm CDC数据源根据该属性的内容提取HBase表Schema。

使用限制

- 仅支持HBase表（表示通过HBase客户端写入Lindorm宽表引擎的表）。
- 实时数据订阅功能的数据消费格式仅支持**json**格式文件。

作业提交方式

您可以通过以下两种方法编写并提交Lindorm CDC数据源的Spark作业。

- 使用SQL访问Lindorm计算引擎
- JAR作业开发实践

> **说明** 读写Lindorm CDC数据源的语法说明请参见Lindorm CDC数据源配置介绍与语法。

Lindorm CDC数据源配置介绍与语法

Lindorm CDC数据源的库表结构介绍

- Lindorm计算引擎提供的Lindorm CDC数据源名称为lindorm_cdc。
- Lindorm CDC数据源不支持管理Namespace，仅支持管理表。表名称为创建数据订阅通道的**主题名**。

Lindorm CDC数据源输出的Schema介绍

Lindorm CDC数据源根据LINDORM_HBASE_CATALOG属性提取HBase表的Schema作为Lindorm CDC数据源的Schema。Lindorm CDC数据源从Kafka中读取数据，每一条操作记录都会被保存。Lindorm CDC数据源的Schema支持以下meta字段：

字段名	类型	说明	字段配置值
_cdc_timestamp_kafka	long	该操作记录写入Kafka的时间戳。单位为毫秒。	无需配置，Schema中包含默认配置值。
_cdc_operation_type	string	该操作记录的变更类型。 • C：新增数据操作。 • U：更新数据操作。 • D：删除数据操作。	无需配置，Schema中包含默认配置值。
_cdc_timestamp_lindorm	long	该操作记录被Lindorm引擎处理的时间戳。单位为毫秒。	spark.sql.catalog.lindorm_cdc.lindormTsEnabled
_cdc_timestamp_lts	long	该操作记录被LTS处理的时间戳。单位为毫秒。	spark.sql.catalog.lindorm_cdc.ltsTsEnabled

Lindorm CDC数据源的配置项介绍

Lindorm CDC数据源的配置项如下表所示：

配置项	是否必填	说明	示例值
spark.sql.catalog.lindorm_cdc.username	• 必填：提交JAR作业或者Python作业。 • 非必填（系统自动添加）：提交SQL作业。	连接Lindorm宽表引擎的用户名。	默认用户名root。
spark.sql.catalog.lindorm_cdc.password	• 必填：提交JAR作业或者Python作业。 • 非必填（系统自动添加）：提交SQL作业。	连接Lindorm宽表引擎的密码。	默认密码root。

配置项	是否必填	说明	示例值
spark.sql.catalog.lindorm_cdc.lindormTsEnabled	非必填	Lindorm处理该操作记录的时间戳。默认值为false。设置为true时Lindorm CDC数据源的Schema中会增加_cdc_timestamp_lindorm字段。	true
spark.sql.catalog.lindorm_cdc.ltsTsEnabled	非必填	LTS处理该操作记录的时间戳。默认值为false。设置为true时Lindorm CDC数据源的Schema中会增加_cdc_timestamp_lts字段。	true

Lindorm CDC数据源的语法介绍

Lindorm CDC数据源支持的语法如下表所示：

语法	描述	示例
USE table_name	使用某个表。	USE test
SHOW TABLES	查看所有的表。	SHOW TABLES
DESCRIBE table_name	查看某个表的详细信息。	DESC test或者DESCRIBE test
SELECT	关于SELECT语法请参见Spark SQL。 **说明** SELECT语法使用中需要注意以下两点： • 必须使用 `_cdc_timestamp_kafka > $startTimestamp and _cdc_timestamp_kafka < $endTimestamp` 指定Kafka中的数据读取范围。 • 对于 `_cdc_operation_type=D` 的字段，仅显示 `rowKey` 对应的字段，其他字段为空。	

11.6. 作业调度运维

11.6.1. 通过Lindorm控制台管理作业

本文介绍通过云原生多模数据库Lindorm控制台管理计算引擎作业，包括查看Spark作业详情和终止Spark作业。

前提条件

已创建Lindorm计算引擎作业，具体操作请参见JAR作业开发实践或者Python作业开发实践。

查看Spark作业

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**计算引擎 > 作业列表**，可以查看Spark作业的详情信息。

Spark作业	描述
JobId	Spark作业ID。
AppName	JAR作业名称。
WebUI地址	SparkUI界面的访问地址，可以将目标SparkUI地址复制在浏览器地址栏中，查看目标Spark作业的详细信息，SparkUI界面的介绍请参见SparkUI界面介绍。

Spark作业	描述
状态	可以查看Spark作业的运行状态。
详情	可以查看Spark作业的运行状态详情信息。
操作	对Spark作业进行终止操作。

终止Spark作业

> **说明**
> - Spark作业的状态为**启动中**和**运行**才能执行终止操作。
> - 终止操作对Lindorm实例无影响。

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**计算引擎 > 作业列表**。
5. 单击目标Spark作业**操作**列中的**终止作业**。

11.6.2. 通过DMS调度Lindorm Spark任务

云原生多模数据库Lindorm计算引擎支持通过数据管理DMS的任务编排功能来调度Lindorm Spark任务，并查看Lindorm Spark任务发布记录和日志，满足用户在数据生产、交互式分析、机器学习和图计算等场景中的计算需求。本文介绍通过DMS调度Lindorm Spark任务的方法。

前提条件

- 已开通数据管理DMS服务。
- 云原生多模数据库Lindorm实例已开通计算引擎，具体操作请参见开通与变配。

创建Lindorm Spark任务流

1. 登录数据管理DMS 5.0。
2. 在顶部菜单栏中，选择**集成与开发（DTS）**。
3. 在左侧导航栏，选择**数据开发 > 任务编排**。
4. 在**任务编排**页面，单击**新增任务流**。
5. 在**新增任务流**对话框中，输入**任务流名称**和**描述**，并单击**确认**。
6. 在左侧**任务类型**列表中，拖拽**Lindorm Spark**任务节点到空白区域，并通过连线的方式定义任务节点之间的依赖关系。
7. 配置**Lindorm Spark**任务节点。
   i. 双击**Lindorm Spark**任务节点或者单击**Lindorm Spark**任务节点并选择 。
   ii. 在打开的任务节点页面中，配置基础参数和运行作业的自定义参数。

> **说明** 配置**Lindorm Spark**任务节点前，需要提前将目标作业文件（例如：JAR包、Python文件或者SQL文件）上传至OSS中，具体操作请参见上传文件。

- 在**基础配置**区域配置基础参数，基础参数说明如下表：

参数	说明
地域	选择目标Lindorm实例所属的地域。
Lindorm实例	选择目标Lindorm实例ID。
任务类型	选择Spark作业类型，支持以下三种： ■ JAR ■ Python ■ SQL

- 在**作业配置**区域配置运行作业的自定义参数。以下列出不同的Spark作业配置模板和自定义参数说明。
  - JAR作业配置模板和自定义参数说明如下：

```
{
  "mainResource" : "oss://path/to/your/file.jar",
  "mainClass" : "path.to.main.class",
  "args" : [ "arg1", "arg2" ],
  "configs" : {
    "spark.hadoop.fs.oss.endpoint" : "",
    "spark.hadoop.fs.oss.accessKeyId" : "",
    "spark.hadoop.fs.oss.accessKeySecret" : "",
    "spark.hadoop.fs.oss.impl" : "org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem",
    "spark.sql.shuffle.partitions" : "20"
  }
}
```

参数	参数类型	是否必填	说明	示例值
mainResource	String	是	JAR包存储在OSS或者HDFS的路径。	■ JAR包存储至OSS：oss://testBucketName/path/spark-examples_2.12-3.1.1.jar ■ JAR包存储至HDFS：hdfs:///path/spark-examples_2.12-3.1.1.jar
mainClass	String	是	JAR作业的程序入口类。	com.aliyun.ldspark.SparkPi
args	Array	否	传入mainClass参数。	["arg1", "arg2"]
configs	Json	否	JAR作业的其他相关配置，相关配置请参见configs相关参数。	{"spark.executor.memory": "8g"}

- Python作业配置模板和自定义参数说明如下：

```
{
  "mainResource" : "oss://path/to/your/file.py",
  "args" : [ "arg1", "arg2" ],
  "configs" : {
    "spark.hadoop.fs.oss.endpoint" : "",
    "spark.hadoop.fs.oss.accessKeyId" : "",
    "spark.hadoop.fs.oss.accessKeySecret" : "",
    "spark.hadoop.fs.oss.impl" : "org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem",
    "spark.submit.pyFiles" : "oss://path/to/your/project_file.py,oss://path/to/your/project_module.zip",
    "spark.archives" : "oss://path/to/your/environment.tar.gz#environment",
    "spark.sql.shuffle.partitions" : "20"
  }
}
```

参数	参数类型	是否必选	说明	示例值
mainResource	String	是	Python文件存储在OSS或者HDFS的路径。	■ Python文件存储至OSS：oss://testBucketName/path/spark-examples.py ■ Python文件存储至HDFS：hdfs:///path/spark-examples.py
args	Array	否	传入mainClass参数。	["arg1", "arg2"]
configs	Json	否	Python作业的其他相关配置，相关配置请参见configs相关参数。	{"spark.executor.memory" : "8g"}

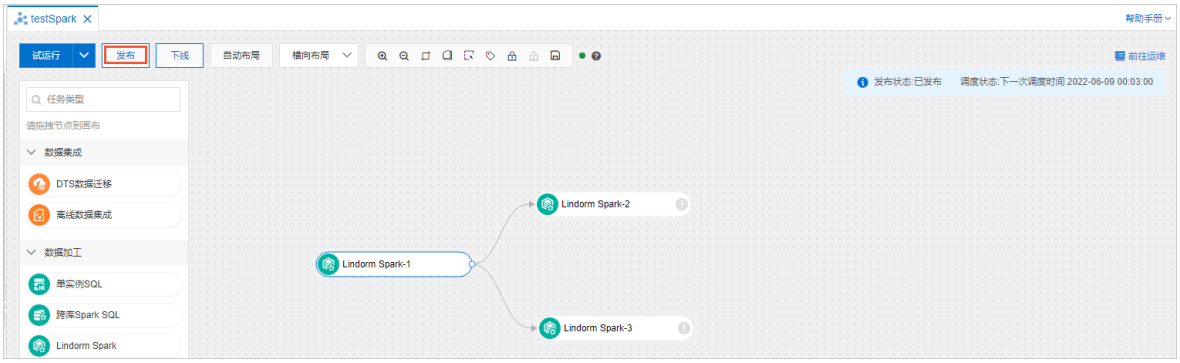
- SQL作业配置模板和自定义参数说明如下：

```
{
  "mainResource" : "oss://path/to/your/file.sql",
  "configs" : {
    "spark.hadoop.fs.oss.endpoint" : "",
    "spark.hadoop.fs.oss.accessKeyId" : "",
    "spark.hadoop.fs.oss.accessKeySecret" : "",
    "spark.hadoop.fs.oss.impl" : "org.apache.hadoop.fs.aliyun.oss.AliyunOSSFileSystem",
    "spark.sql.shuffle.partitions" : "20"
  }
}
```

参数	参数类型	是否必选	说明	示例
mainResource	String	是	SQL文件存储在OSS或者HDFS的路径。	■ SQL文件存储至OSS：oss://testBucketName/path/spark-examples.sql ■ SQL文件存储至HDFS：hdfs:///path/spark-examples.sql
configs	Json	否	SQL作业的其他相关配置。	{"spark.executor.memory" : "8g"}

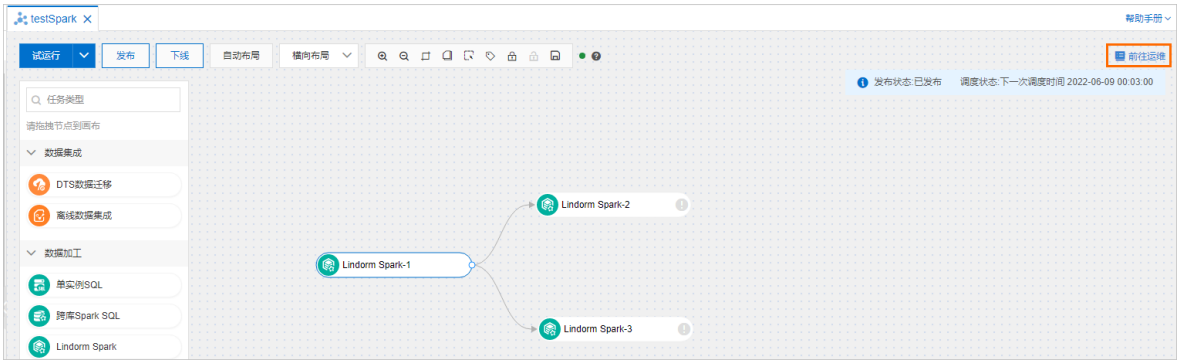
iii. 完成以上配置后，单击页面左上方的**试运行**，检查作业运行效果是否符合预期。

8. 发布任务流。所有的任务节点配置完成后，单击任务流名称页面左上方的**发布**。



查看任务流的发布记录和日志

1. 在**任务编排**页面单击目标任务流名称。
2. 单击任务流名称页面右上角的**前往运维**。

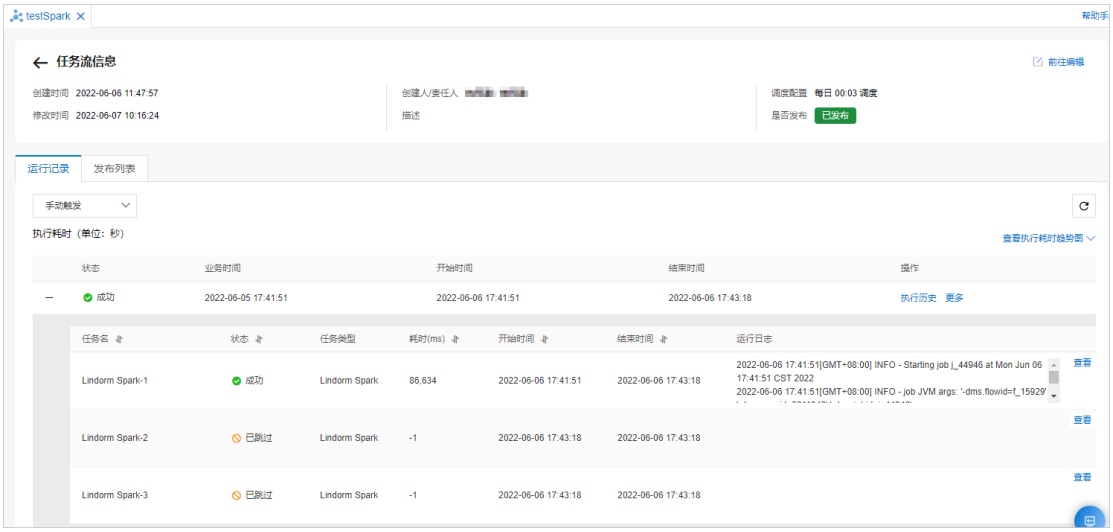


3. 查看任务流的发布记录和日志。
   - 查看任务流的发布记录。在**任务流信息**页面，单击**发布列表**页签可以查看任务流的发布记录。



   - 查看任务流的日志。

a. 在**运行记录**页签左上方的下拉列表中选择**定时触发**或者**手动触发**，可以查看任务流中所有任务节点的详细信息。



b. 单击任务节点名称后面的**查看**，可以查看Lindorm Spark任务作业的提交日志，获取该任务节点的JobID和SparkUI。

> **说明** 如果任务提交失败，请将JobID和SparkUI提供给工单处理人员并提交工单。

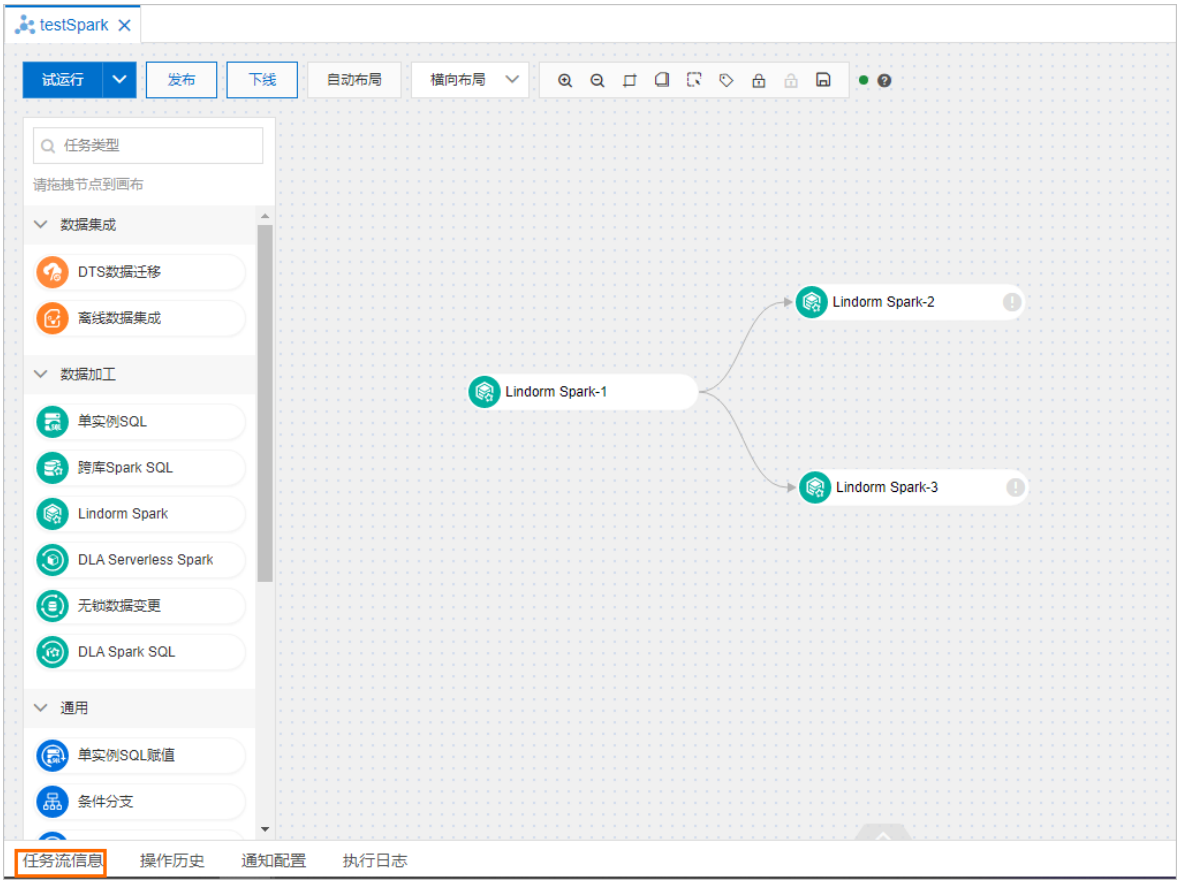
高级配置

> **说明** 通过数据管理DMS控制台对Lindorm Spark任务流进行相关配置，配置完成后需要重新发布任务流。

调度配置

根据业务需求配置相应的调度策略，Lindorm Spark任务流会根据该调度策略自动执行。配置方法如下：

1. 在**任务编排**页面单击目标任务流名称。
2. 单击任务流名称页面左下角的**任务流信息**。



3. 在右侧**调度配置**区域，打开**开启调度**开关，配置调度策略，配置项说明如下表。

   调度周期配置表

配置项	说明
调度类型	选择调度类型： ○ **周期调度**：周期性调度任务，例如一周执行一次任务。 ○ **调度一次**：在指定时间执行一次任务，仅需要配置执行任务的具体时间。
生效时间	选择调度周期生效的区间，默认**1970-01-01~9999-01-01**，表示一直生效。
调度周期	选择调度任务的周期： ○ **小时**：按设定的小时执行任务调度，需要配置定时调度。 ○ **日**：按每日一次的频率执行任务调度，需要配置每日调度的具体时间。 ○ **周**：以周为周期，每个指定天执行一次任务调度，需要配置指定时间和具体时间。 ○ **月**：以月为周期，每个指定天执行一次任务调度，需要配置指定时间和具体时间。

配置项	说明
定时调度	提供了2种定时调度的方式： ○ 固定间隔时间调度： ■ **开始时间**：执行任务的开始时间。 ■ **间隔时间**：执行任务的间隔时间，单位为小时。 ■ **结束时间**：执行任务的结束时间。 例如，配置开始时间为00:00、间隔时间为6小时、结束时间为20:59，系统将在0点、6点、12点、18点执行任务。 ○ **指定时间**调度：选择执行任务的目标时间点。 例如选择和0小时和5小时，系统将在0点和5点执行任务。
指定时间	○ 如果调度周期为周，选择星期几执行任务，支持多选。 ○ 如果调度周期为月，选择每月几号执行任务，支持多选。
具体时间	设置执行任务流的具体时间。 例如配置02:55，系统将在指定天的02时55分执行任务。
cron表达式	不需要手动配置，系统会根据您配置的周期、具体时间自动展现。

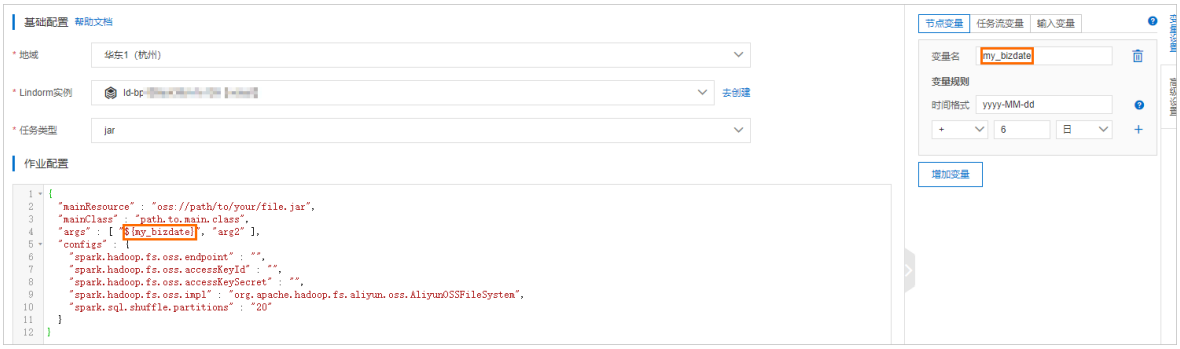
调度配置示例：如果需要配置任务流在每天0点和12点进行调度，调度策略配置如下。

- **调度类型**选择周期调度。
- **调度周期**选择小时。
- **定时调度**选择指定时间，同时在**指定时间**列表中选择0小时和12小时。

变量配置

对于周期调度的任务流，可以将调度时间作为参数传递给要执行的任务。例如：将业务时间 `bizdate` （运行时间的前一天）传递给任务节点，按照如下步骤配置时间变量。

1. 在任务流名称页面，双击Lindorm Spark任务节点或者Lindorm Spark任务节点并选择 。
2. 在右侧菜单栏选择**变量设置**。
3. 在**节点变量**或者**任务流变量**页签中添加变量名。
4. 在**作业配置**区域中使用变量。更多变量请参见变量概述。



通知配置

如果打开通知开关，系统会根据任务流执行的结果发送相应的通知消息。打开通知方法如下：

1. 单击任务流名称页面左下角的**通知配置**。
2. 根据业务需求打开对应的通知开关。
   - 成功通知：任务流执行成功发送通知信息。
   - 失败通知：任务流执行失败发送通知信息。

- 超时通知：任务流执行超时发送通知信息。

3. （可选）配置消息接收人，请参见消息通知管理。

相关文档

有关数据管理DMS的任务编排功能，详情请参见任务编排概述。

11.6.3. SparkUI界面介绍

云原生多模数据库Lindorm计算引擎提供了SparkUI界面，方便您查看Spark作业的运行状态、作业提交时间以及资源使用情况等详细信息。

进入SparkUI界面

> 说明
> - SparkUI地址获取方法请参见查看SparkUI地址。
> - 关于开源SparkUI界面的详细介绍请参见开源SparkUI。

将SparkUI地址复制在浏览器地址栏中，可以进入SparkUI界面。



SparkUI页签介绍

页签	描述
Jobs	所有运行作业的详细信息。
Stages	Spark作业所有Stage的状态信息。
Storage	显示持久化的RDD以及DF的存储信息。
Environment	Spark作业运行的环境。Spark作业启动时，从运行环境、配置文件、用户传入参数共同构建而来。
Executors	Spark作业运行的Executors状态信息。
SQL	计算引擎执行SQL的详细信息。
Kyuubi Query Engine	展示JDBC Session信息。

11.6.4. 查看计算引擎作业运行日志

计算引擎作业提交成功后，您可以通过SparkUI界面或者HDFS Shell工具查看计算引擎作业运行日志。

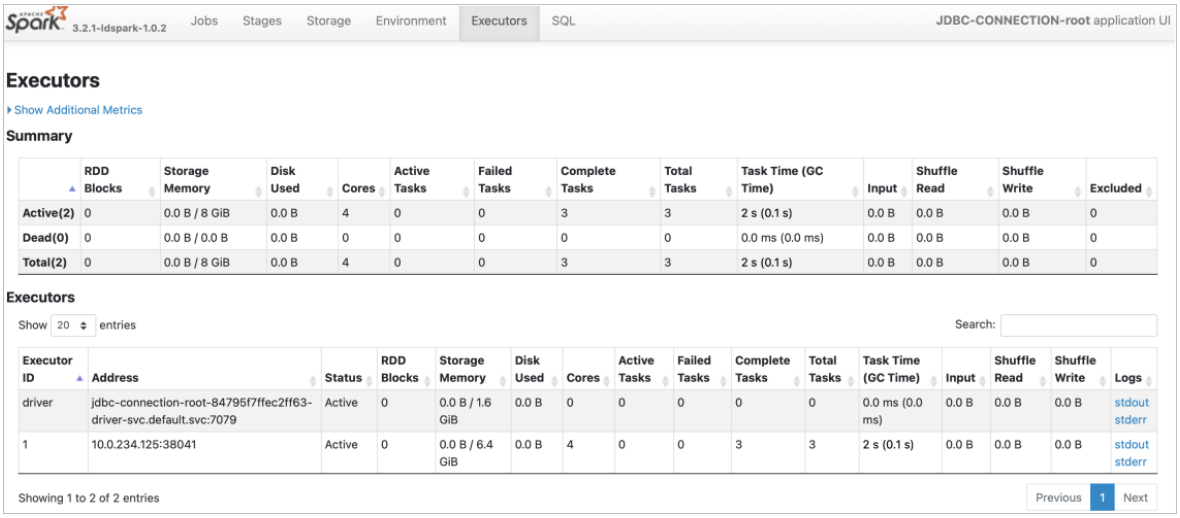
前提条件

- 已开通云原生多模数据库Lindorm实例的计算引擎服务，具体操作请参见开通与变配。
- 已将客户端IP地址添加至云原生多模数据库Lindorm实例的白名单，具体操作请参见设置白名单。

通过SparkUI界面查看计算引擎作业运行日志

通过SparkUI界面查看计算引擎作业运行日志，需要查看的计算引擎作业状态为**运行中**。

1. 登录Lindorm管理控制台。
2. 在**实例列表**页，单击目标实例ID。
3. 在左侧导航栏，选择**计算引擎 > 作业列表**，可以查看正在运行的计算引擎作业。
4. 单击需要查看计算引擎作业的**WebUI地址**，打开SparkUI界面。
5. 在顶部菜单栏单击**Executors**，可以查看计算引擎作业和所有的Executor。



6. 在**Executors**列表中，单击**Logs**列的**stdout**或者**stderr**查看运行日志。

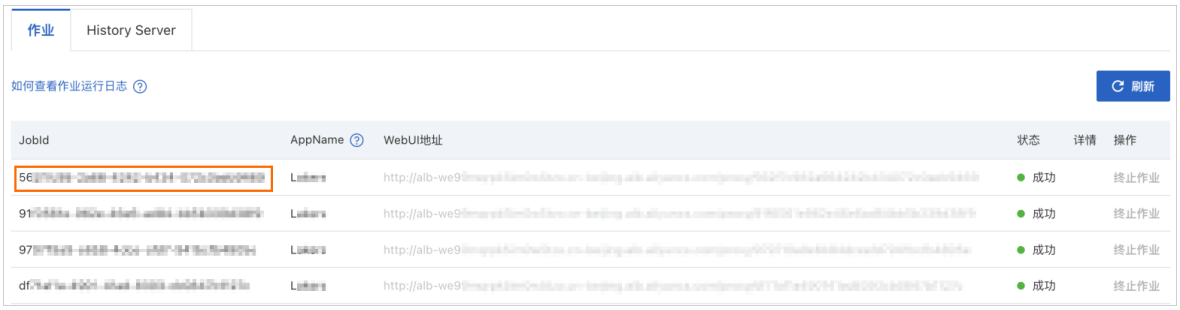
> **说明**
> - 单击**stdout**可以查看标准输出日志。
> - 单击**stderr**可以查看标准错误输出日志。

通过HDFS Shell工具查看计算引擎作业运行日志

计算引擎作业的运行日志会被系统同步并存储至底层的文件引擎中，如果需要查看的计算引擎作业中断，您可以开通云原生多模数据库Lindorm文件引擎，并使用HDFS Shell工具查看作业的运行日志。

> **注意**
> - 建议您在提交JAR作业时，在Main函数开始时初始化SparkSession对象，避免部分异常运行日志无法同步至文件引擎中。
> - 如果Lindorm实例的计算引擎服务规模较大，计算引擎作业运行日志可能给DFS造成较大的压力。您可以在计算引擎作业启动参数中配置spark.dfsLog.executor.enabled=false，该参数指定Executor日志不保存至DFS，而计算引擎作业Driver运行日志的收集不会受到影响。

1. 登录Lindorm管理控制台。
2. 在**实例列表**页，单击目标实例ID。
3. 在左侧导航栏，选择**计算引擎 > 作业列表**，在**作业**页面获取**JobId**，例如：`562f7c98-2a66-****`，如下图所示。



4. 使用HDFS Shell工具查看日志，HDFS Shell工具的配置方法请参见通过HDFS Shell连接并使用文件引擎。
5. 计算引擎作业的日志存储目录为`/ldspark/ldspark-logs/${JobId}`，Driver日志存储目录为`__driver_logs__`，Executor日志存储目录为`__executor_logs__/${EXECUTOR_ID}`。以查看Driver的**stderr**运行日志为例，请执行以下语句：

```
$HADOOP_HOME/bin/hadoop fs -cat  /ldspark/ldspark-logs/${JobId}/__driver_logs__/stderr | less
```

> **说明** 您也可以通过FUSE客户端将文件引擎目录挂载至ECS进行访问，具体操作请参见通过HDFS FUSE连接并使用文件引擎。

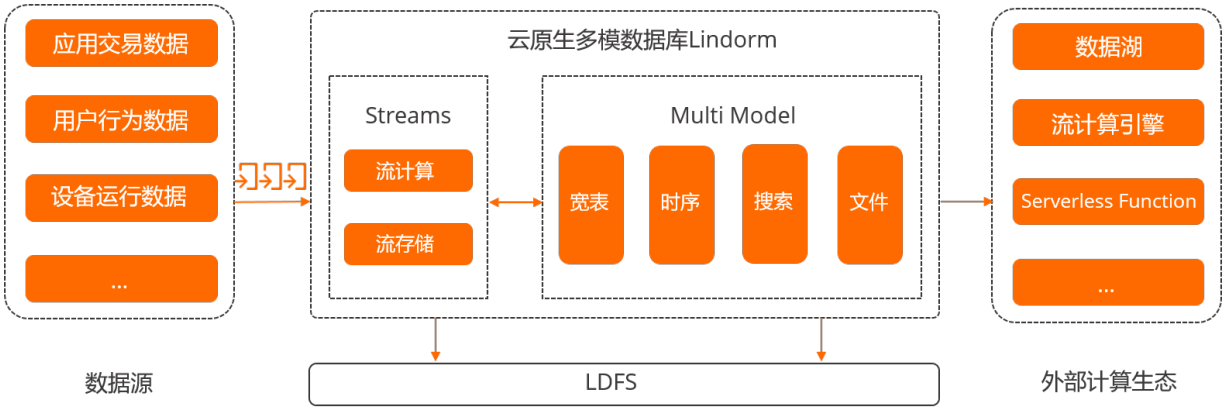
12.流引擎

12.1. 引擎简介

云原生多模数据库Lindorm流引擎是面向流式数据处理的引擎，提供了流式数据的存储和轻计算功能，帮助您轻松实现流式数据存储至云原生多模数据库Lindorm，构建基于流式数据的处理和应用。本文介绍Lindorm流引擎的应用场景和功能特性。

产品架构

云原生多模数据库Lindorm流引擎的架构图如下：

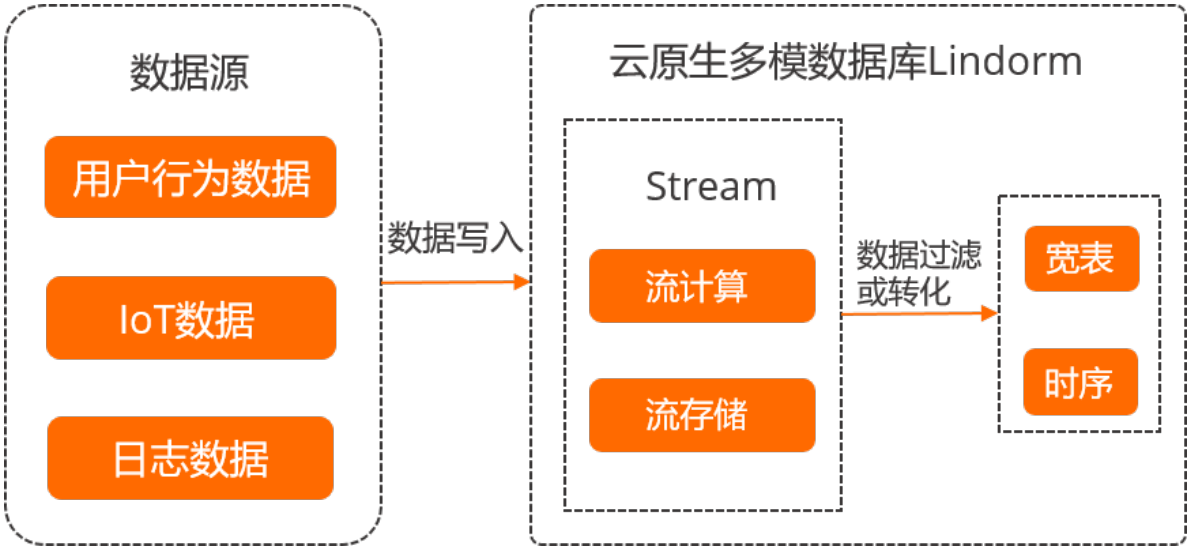


应用场景

Lindorm流引擎主要用于应用日志、物联网实时数据等流式数据存储至云原生多模数据库Lindorm宽表引擎或者时序引擎。Lindorm流引擎的典型应用场景为ELT，流程如下。相比于传统的复杂组合方案（由Kafka、Flink和数据库组成），云原生多模数据库Lindorm提供了一体化的存储、计算和查询能力，简化了运维操作，降低了开发成本。

1. 将数据源写入至Lindorm流引擎，写入的数据格式包括CSV、Avro、JSON等。
2. 通过Lindorm流引擎SQL对数据源进行过滤或者转化等轻量计算。
3. 将计算结果同步至Lindorm宽表引擎或者时序引擎。

流程图如下：



功能特性

Lindorm流引擎的功能特性分为三个方面：

- Lindorm流引擎SQL

功能	说明
SQL客户端	支持JDBC协议，与SQL生态工具融合。
SQL语法	支持基本的DDL和DML操作。
多种类函数	通用的SQL函数、Lindorm内置函数、用户自定义函数。
可视化显示	通过控制台可以查看流处理链路信息。
窗口函数	支持流计算窗口函数。

- Schema管理

功能	说明
Schema映射	可以将数据源映射成为Table Format。数据源的写入格式包括CSV、Avro、JSON等。
脏数据处理	在Schema映射过程中，由于数据类型不符合或者主键列为空会存在一些不符合Schema的数据，这类数据为脏数据。对于脏数据提供一些默认的处理机制，包括阻塞、忽略、等。死信队列

- 数据写入和数据存储

功能	说明
100%兼容Kafka协议	可以使用开源Kafka客户端写入数据到Lindorm流引擎。
存储和计算分离、存储容量大	Lindorm流引擎可以存储PB级别的数据，支持存储独立水平扩展。

12.2. 快速入门

12.2.1. 入门概览

本文介绍使用云原生多模数据库Lindorm流引擎从开通到连接使用的整体流程，指引您快速开通云原生多模数据库Lindorm流引擎、连接流引擎及使用流引擎的方法。

操作流程

从开通Lindorm流引擎到使用流引擎，您需要完成以下操作：

1. 开通Lindorm流引擎。
2. 查看Lindorm流引擎连接地址。
3. 通过Lindorm-cli连接并使用Lindorm流引擎。
4. 通过开源Kafka客户端写入Lindorm流引擎数据。

12.2.2. 开通流引擎

本文介绍创建云原生多模数据库Lindorm实例后如何开通流引擎。

前提条件

已创建云原生多模数据库Lindorm实例，具体操作请参见创建实例。

使用限制

开通Lindorm流引擎仅支持付费类型为包年包月的Lindorm实例。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。

4. 在左侧导航栏，选择**流引擎 > 集群管理**。
5. 单击**这里**开通流引擎。
6. 在**变配**页面，配置流引擎节点规格和节点数量，并勾选服务协议。

> **说明** 如果您在配置流引擎节点规格中有疑问，请提交工单或者在钉钉群中反馈，相关操作请参见技术支持。

7. 配置完成后，单击**立即购买**。目标Lindorm实例状态会显示**节点扩容中**，请等待5~10分钟，当目标Lindorm实例状态显示为**运行中**，实例可以正常使用。

12.2.3. 查看连接地址

本文介绍通过云原生多模数据库Lindorm控制台查看流引擎类的连接地址。

前提条件

已开通云原生多模数据库Lindorm的流引擎，具体操作请参见开通流引擎。

专有网络和公网

网络类型	说明
专有网络	专有网络VPC（Virtual Private Cloud）是您自己独有的云上私有网络，不同的专有网络之间通过二层逻辑隔离，拥有较高的安全性和性能。Lindorm-cli部署在ECS实例上时，通过专有网络连接至Lindorm实例，可获得更高的安全性和更低的网络延迟。
公网	公网即互联网，当本地设备需要连接至Lindorm实例时，您可以为Lindorm实例申请公网连接地址。通过公网连接不会产生流量费用，但存在一定的安全风险，推荐通过专有网络连接以获取更高的安全性。

操作步骤

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏中选择**数据库连接**。
5. 单击**流引擎**页签，可以查看Lindorm实例流引擎的访问地址。



> **说明** 如果您需要使用公网访问Lindorm流引擎，请单击页面右边的**开通外网地址**获取公网访问地址。

访问地址	示例	描述
Lindorm Stream SQL地址	jdbc:streamsql:url=http://ld-bp17pwu1541ia****-proxy-stream.lindorm.rds.aliyuncs.com:30060	通过Lindorm-cli连接Lindorm流引擎时使用该地址。
Lindorm Stream Kafka地址	ld-bp17pwu1541ia****-proxy-stream.lindorm.rds.aliyuncs.com:30080	通过开源Kafka客户端连接Lindorm流引擎时使用该地址。

相关文档

- 通过Lindorm-cli连接并使用Lindorm流引擎
- 通过开源Kafka客户端写入Lindorm流引擎数据

12.2.4. 通过Lindorm-cli连接并使用Lindorm流引擎

本文介绍通过Lindorm-cli连接Lindorm流引擎的具体操作和使用示例。

前提条件

- 已将客户端IP地址添加至Lindorm实例的白名单中，具体操作请参见设置白名单。
- 已获取Lindorm流引擎的Lindorm Stream SQL地址，具体操作请参见查看连接地址。
- 已获取Lindorm宽表引擎中HBase兼容的专有网络连接地址，具体操作请参见查看连接地址。

注意事项

如果应用部署在ECS实例，您需要确保云原生多模数据库Lindorm实例和ECS实例满足以下条件，以保证网络的连通性。ECS实例信息的查看方法请参见查看实例信息。

- 所在地域相同，并建议所在可用区相同（以减少网络延时）。
- 网络类型相同。

连接Lindorm流引擎使用的网络类型

网络类型	说明
专有网络（推荐）	专有网络VPC（Virtual Private Cloud）是您自己独有的云上私有网络，不同的专有网络之间二层逻辑隔离，拥有较高的安全性和性能。Lindorm-cli部署在ECS实例上时，通过专有网络连接至Lindorm流引擎，可获得更高的安全性和更低的网络延迟。
公网	公网即互联网，当本地设备需要测试或管理Lindorm流引擎时，可在本地设备上部署Lindorm-cli，然后通过公网连接至Lindorm流引擎。 **说明** 通过公网连接不会产生流量费用，但存在一定的安全风险，推荐通过专有网络连接以获取更高的安全性。

操作步骤

1. 在ECS上执行以下命令下载Lindorm-cli压缩包。

   ```
   wget https://hbaseuepublic.oss-cn-beijing.aliyuncs.com/lindorm-stream-sqlline-0.1.0.tar.gz
   ```

2. 执行以下命令解压Lindorm-cli压缩包。

   ```
   tar -zxf lindorm-stream-sqlline-0.1.0.tar.gz
   ```

3. 进入 `lindorm-sqlline-0.1.0/bin` 目录，执行以下命令连接至Lindorm流引擎。

   ```
   ./lindorm-sqlline -url <连接地址> -u <用户名> -p <密码>
   ```

参数	示例	说明
连接地址	jdbc:streamsql:url=http://ld-bp17pwu1541ia****-proxy-stream.lindorm.rds.aliyuncs.com:30060	Lindorm流引擎的Lindorm Stream SQL连接地址，获取方法请参见查看连接地址。
用户名	root	如果您忘记用户名密码，可以通过Lindorm宽表引擎的集群管理系统修改密码，具体操作请参见修改用户密码。
密码	root	

4. 使用Lindorm流引擎。

   通过Lindorm流引擎将日志数据进行转化后实时同步至Lindorm宽表中，并对日志数据进行多维检索操作。本示例的操作流程如下：

准备工作

i. 通过Lindorm-cli连接Lindorm宽表引擎，并在Lindorm宽表中创建一张表，具体操作请参见通过Lindorm-cli连接并使用宽表引擎。

```
CREATE TABLE IF NOT EXISTS log (
  loglevel VARCHAR,
  thread VARCHAR,
  class VARCHAR,
  detail VARCHAR,
  timestamp BIGINT,
  primary key (timestamp)
);
```

ii. 通过Lindorm-cli连接Lindorm流引擎。

```
./lindorm-sqlline -url jdbc:streamsql:url=http://ld-bp17pwu1541ia****-proxy-stream.lindorm.rds.aliyu
ncs.com:30060 -u root -p root
```

操作步骤

i. 在Lindorm流引擎中创建数据流表。

```
CREATE STREAM IF NOT EXISTS originalData (
  `loglevel` STRING,
  `thread` STRING,
  `class` STRING,
  `detail` STRING,
  `timestamp` LONG
) WITH (
  value_format = 'json', //表示数据源的格式
  key_value = 'thread',  //表示分区键
  stream_topic = 'log_topic', //指定数据流表的物理数据存储在流存储的某个Topic上
  TIMESTAMP = 'timestamp'   //指定时间戳列
);
```

创建成功后，通过以下命令查看结果：

```
list streams;
```

执行结果如下：

```
+-----------------------------------------------------------------------------------------+
|                                        ShowStreams                                      |
+-----------------------------------------------------------------------------------------+
|  Stream Name  | Stream Topic | Format
-----------------------------------------
 originalData | log_topic    | JSON
-----------------------------------------
  |
+-----------------------------------------------------------------------------------------+
1 row selected (0.13 seconds)
```

ii. 创建Serving表。Serving表示用于提供查询能力，例如点查询、范围查询、多维检索、全文检索、向量检索、时序查询、时空查询、图查询、交互式分析等。Serving表对于Lindorm流引擎来说是一个外表，源数据存储在Lindorm宽表中，通过以下语句来关联Lindorm宽表。

```
CREATE External Table IF NOT EXISTS lindorm_log_table WITH (
  table_type = 'lindorm.table',
  table_name = 'log',
  endpoint = 'ld-bp17pwu1541ia****-proxy-lindorm.lindorm.rds.aliyuncs.com:30020'
);
```

> 说明
> - `table_type`：表示源数据存储在Lindorm宽表引擎中。
> - `table_name`：表示Lindorm宽表的表名。
> - `endpoint`：Lindorm宽表引擎中HBase兼容的专有网络连接地址。

创建成功后，通过以下命令查看结果：

```
list external tables;
```

执行结果如下：

```
+-----------------------------------------------------------------------------------------------+
|                                    ShowExternalTables                                         |
+-----------------------------------------------------------------------------------------------+
|  External Table Name
---------------------
 lindorm_log_table
---------------------
  |
+-----------------------------------------------------------------------------------------------+
1 row selected (0.024 seconds)
```

iii. 创建流处理链路。将数据流表中的日志数据同步写入Serving表中，同时过滤出loglevel等于error的日志。

```
CREATE CQ log_to_lindorm
Insert Into lindorm_log_table
Select loglevel, thread, class, detail, timestamp From originalData
Where loglevel = 'ERROR';
```

iv. 写入数据。通过Kafka API写入数据至指定的Topic（名称为log_topic）中，具体操作请参见通过开源Kafka客户端写入Lindorm流引擎数据。

v. 查询数据。通过以下两种方式都可以查询流处理链路的结果。

- 通过Lindorm-cli连接Lindorm宽表引擎并查询处理结果。`lindorm_log_table`为Serving表的名称。

  ```
  SELECT * FROM lindorm_log_table;
  ```

- 通过Lindorm宽表引擎的集群管理系统查询处理结果，具体操作请参见数据查询。

12.2.5. 通过开源Kafka客户端写入Lindorm流引擎数据

Lindorm流引擎完全兼容开源Kafka API，您可以通过Kafka API编写程序写入Lindorm流引擎数据，也可以通过开源的三方工具采集并写入Lindorm流引擎数据，例如FluentD、Debezium等。本文介绍通过开源Kafka客户端连接Lindorm流引擎并写入Lindorm流引擎数据的代码示例。

前提条件

- 已安装Java环境，要求安装JDK 1.7及以上版本。
- 已将客户端IP地址添加至Lindorm实例的白名单中，具体操作请参见设置白名单。
- 已获取Lindorm流引擎的Lindorm Stream Kafka地址，具体操作请参见查看连接地址。

> **说明** Lindorm流引擎的Lindorm Stream Kafka地址为专有网络地址，需确保应用程序部署的环境和Lindorm实例使用相同的专有网络ID。

操作步骤

1. 下载开源Kafka客户端。在pom.xml中配置Maven依赖，具体内容如下：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.10.2.2</version>
</dependency>
```

2. 连接Lindorm流引擎并写入Lindorm流引擎数据。完整的代码示例如下：

> **说明**
> - 写入的数据格式支持JSON、Avro和CSV。
> - 代码中Lindorm Stream Kafka地址为专有网络地址，获取方法请参见查看连接地址。

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.codehaus.jettison.json.JSONObject;
import java.util.Properties;
import java.util.concurrent.Future;
public class KafkaToLindormStreamDemo {
    public static void main(String[] args) {
        Properties props = new Properties();
        //设置Lindorm Stream Kafka地址，这个Lindorm Stream Kafka地址为专有网络地址，需确保应用程序部署的环境和Li
ndorm实例使用相同的专有网络ID。
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "Lindorm Stream Kafka地址");
       //指定数据流表的物理数据存储在某个Topic上
        String topic = "log_topic";
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.Str
ingSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.S
tringSerializer");
        KafkaProducer<String, String> producer = new KafkaProducer<String, String>(props);
        try {
            JSONObject json = new JSONObject();
            //写入流引擎数据
            json.put("timestamp", System.currentTimeMillis());
            json.put("loglevel", "ERROR");
            json.put("thread", "[ReportFinishedTask7-thread-4]");
            json.put("class", "engine.ImporterTaskManager(318)");
            json.put("detail", "Remove tasks fail: job name=e35318e5-52ea-48ab-ad2a-0144ffc6955e , task
name=prepare_e35318e5-52ea-48ab-ad2a-0144ffc6955e , runningTasks=0");
            Future<RecordMetadata> future = producer.send(
          new ProducerRecord<String, String>(topic, json.getString("thread") + json.getLong("timestamp")
,
              json.toString()));
            producer.flush();
            try {
                RecordMetadata recordMetadata = future.get();
                System.out.println("Produce ok:" + recordMetadata.toString());
            } catch (Throwable t) {
                System.out.println("Produce exception " + t.getMessage());
                t.printStackTrace();
            }
        } catch (Exception e) {
            System.out.println("Produce exception " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

12.3. 流引擎SQL语法介绍

12.3.1. 数据类型

本文介绍Lindorm流引擎SQL支持的数据类型。
Lindorm流引擎SQL支持的数据类型如下表：

数据类型	说明
STRING	变长字符串。
VARCHAR(n)	变长字符串，n是指字节长度。

数据类型	说明
BOOLEAN/BOOL	布尔类型，取值： • 0：表示false。 • 1：表示true。
INTEGER/INT	常用的整数，存储空间为4字节，取值范围：-2147483648~2147483647。
LONG/BIGINT	大范围的整数，存储空间为8字节，取值范围：-9223372036854775808~9223372036854775807。
DOUBLE	IEEE标准，存储长度为8字节，取值包括：-1.7976931348623157E+308~-2.2250738585072014E-308, 0, 2.2250738585072014E-308~1.7976931348623157E+308。

12.3.2. DDL

12.3.2.1. CREATE STREAM

创建一张数据流表，表示一个无界的事件集合。数据流表仅支持新增数据，不支持更新数据和删除数据。数据流表中的最小单元为Record，Record有预定义的Schema。数据流表的物理数据存储在流存储的某个指定的Topic上，具体的存储格式可以在With语句中指定。

语法

```
CREATE STREAM (IF NOT EXISTS) table_name
                  ('(' tableElement (',' tableElement)* ')')
                  (WITH tableProperties)
tableElement
    : columnDefinition
    ;
columnDefinition
    : identifier type (NOT NULL) (COMMENT string) (WITH properties)
    ;
tableProperties
    : '(' tableProperty (',' tableProperty)* ')'
    ;
tableProperty
    : identifier EQ expression
    ;
```

参数说明

参数	是否必选	说明
table_name	是	数据流表的名称。
tableElement	是	定义数据流表的列，格式为：`列名 列的数据类型`。
tableProperties	是	指定数据流表的属性，包括以下配置项： • VALUE_FORMAT：必选项，指定数据源写入Lindorm流引擎的数据格式，包括CSV、Avro、JSON等格式。 • KEY_VALUE：必选项，指定数据流表的主键列。 • STREAM_TOPIC：可选项，指定数据流表的物理数据存储在流存储的某个Topic上。 • TIMESTAMP：可选项，指定时间戳列。

示例

```
CREATE STREAM users_original (
  registertime bigint,
  gender varchar,
  regionid varchar,
  userid varchar)
WITH (
    stream_topic='users',
    value_format='JSON',
    key_value = 'userid'
);
```

12.3.2.2. CREATE EXTERNAL TABLE

创建一张外表。外表是用来将Lindorm流引擎与Lindorm宽表或者时序表进行关联。创建外表不需要声明有哪些列，当需要列的Schema信息时，系统会实时读取被关联的Lindorm宽表或者时序表的源数据，从而保持源数据的一致性。

语法

```
CREATE EXTERNAL TABLE (IF NOT EXISTS) table_name
             (WITH tableProperties)
tableProperties
    : '(' tableProperty (',' tableProperty)* ')'
    ;
tableProperty
    : identifier EQ expression
    ;
```

参数说明

参数	是否必选	说明
table_name	是	外表的名称。
tablePropertys	是	将Lindorm流引擎与Lindorm宽表或者时序表进行关联，需要配置以下参数： • TABLE_TYPE：lindorm.table表示关联的是Lindorm宽表。lindorm.tsdb表示关联的是Lindorm时序表。 • TABLE_NAME：关联的Lindorm宽表或者Lindorm时序表的表名。 • ENDPOINT：如果关联的是Lindorm宽表，需要配置为Lindorm宽表引擎的HBase兼容连接地址。如果关联的是Lindorm时序表，需要配置为Lindorm时序引擎的连接地址。

示例

```
CREATE EXTERNAL TABLE IF NOT EXISTS user_external
WITH (
  `table_type` = 'lindorm.table',
  `table_name` = 'test',
  `endpoint` = 'ld-bp17pwu1541ia****-proxy-lindorm.lindorm.rds.aliyuncs.com:30020'
);
```

12.3.2.3. DROP STREAM

删除数据流表。

语法

```
DROP STREAM (IF EXISTS) table_name;
```

参数

参数	是否必选	说明
table_name	是	数据流表的名称。

示例

```
DROP STREAM users_original;
```

12.3.2.4. DROP EXTERNAL TABLE

删除外表。删除外表后会取消Lindorm流引擎与Lindorm宽表或者时序表的关联关系。

语法

```
DROP EXTERNAL TABLE (IF EXISTS) table_name;
```

参数说明

参数	是否必选	说明
table_name	是	外表的名称。

示例

```
DROP EXTERNAL TABLE external_users_original;
```

12.3.2.5. SHOW STREAMS

查询Lindorm流引擎中数据流表的列表信息。

语法

> **说明** 查询Lindorm流引擎中数据流表的列表信息也可以使用 `LIST STREAMS` 。

```
SHOW STREAMS;
```

示例

```
Show Streams;
```

返回结果如下：

```
+------------------------------------------------------------------------------------------+
|                                        ShowStreams                                       |
+------------------------------------------------------------------------------------------+
|  Stream Name  | Stream Topic | Format
-----------------------------------------
 originalData | log_topic    | JSON
-----------------------------------------
  |
+------------------------------------------------------------------------------------------+
1 row selected (0.13 seconds)
```

12.3.2.6. SHOW EXTERNAL TABLES

查询Lindorm流引擎中外表的列表信息。

语法

> **说明** 查询Lindorm流引擎中外表的列表信息也可以使用 `LIST EXTERNAL TABLES` 。

```
SHOW EXTERNAL TABLES;
```

示例

```
SHOW EXTERNAL TABLES;
```

返回结果如下：

```
+-----------------------------------------------------------------------------------------------+
|                                    ShowExternalTables                                         |
+-----------------------------------------------------------------------------------------------+
|  External Table Name
---------------------
 lindorm_log_table
---------------------
  |
+-----------------------------------------------------------------------------------------------+
1 row selected (0.024 seconds)
```

12.3.2.7. DESCRIBE

查询Lindorm流引擎中数据流表或者外表的列信息。

语法

```
DESCRIBE table_name;
```

参数说明

参数	是否必选	说明
table_name	是	数据流表或者外表的名称。

示例

```
DESCRIBE USERS_ORIGINAL;
```

返回结果如下：

```
-------------------------------------------------------------------------------------------------+
|  Field     | Type
---------------------------------------
 ROWTIME   | BIGINT           (system)
 ROWKEY    | VARCHAR(STRING)  (system)
 loglevel  | VARCHAR(STRING)
 thread    | VARCHAR(STRING)
 class     | VARCHAR(STRING)
 detail    | VARCHAR(STRING)
 timestamp | BIGINT
---------------------------------------
```

12.3.3. DQL

12.3.3.1. SELECT

查询数据流表中的符合条件的列的数据。

语法

```
SELECT setQuantifier selectItem (',' selectItem)*
      (FROM relation (',' relation)*)
      (WINDOW  windowExpression)
      (WHERE where=booleanExpression)
      (GROUP BY groupBy)
    ;
```

selectItem参数说明

参数值	说明	语句示例
*	查询所有列	`SELECT * FROM Test;`
columnName	查询指定列，指定列支持聚合函数和其他设置，具体请参见列的配置。	`SELECT gender FROM Test;`
AS	别名	`SELECT REGISTERTIME AS rtime FROM Test;`

列的配置

配置值	说明	语句示例
MAX	返回指定列的最大值。	`SELECT item_type,MAX(price) FROM orders GROUP BY item_type;`
MIN	返回指定列的最小值。	`SELECT item_type,MIN(price) FROM orders GROUP BY item_type;`
SUM	返回指定列的总和。	`SELECT item_type,SUM(price) FROM orders GROUP BY item_type;`
COUNT	返回指定列的个数。	`SELECT item_type,COUNT(item_id) FROM orders GROUP BY item_type;`
AVG	返回指定列的平均值，通过SUM(col)/COUNT(*) 实现。	`SELECT item_type, SUM(price)/COUNT(*) AS avg_price FROM orders GROUP BY item_type;`
LEN(column_name)	返回指定列的长度。	`SELECT LEN(column_name) FROM orders;`
IFNULL(value1,value2)	返回value1或者value2的值，需要遵循以下几点： • value1和value2的数据类型保持一致。 • 如果value1的值为空，则返回value2的值。 • 如果value1的值为不为空，则返回value1的值。	`SELECT IFNULL("Hello", "RUNOOB");`
TRIM(column_name)	删除字符串首尾的空格部分。	`SELECT TRIM(column_name) FROM orders;`
CONCAT(str1, str2,...)	返回拼接后的字符串。	`SELECT CONCAT("Hello", "RUNOOB");`
SUBSTRING（string，offset，len）	截取字符串的一部分，offset从1开始，len为截取长度。	`SELECT SUBSTRING("Hello",1 ,2);`

配置值	说明	语句示例
UCASE(column_name)	将列名转化为大写字符串。	`SELECT UCASE(column_name) FROM orders;`
LCASE(column_name)	将列名转化为小写字符串。	`SELECT LCASE(column_name) FROM orders;`

WHERE说明

WHERE值	说明	语句示例
=	等于	`SELECT* FROM Test WHERE gender = 'FEMALE';`
!=	不等于	`SELECT* FROM Test WHERE gender != 'FEMALE';`
<>	不等于	`SELECT* FROM Test WHERE REGISTERTIME <> 1;`
>	大于	`SELECT* FROM Test WHERE REGISTERTIME > 1;`
>=	大于等于	`SELECT* FROM Test WHERE REGISTERTIME >= 1;`
<	小于	`SELECT* FROM Test WHERE REGISTERTIME < 1;`
<=	小于等于	`SELECT* FROM Test WHERE REGISTERTIME <= 1;`
BETWEEN	选取介于两个值之间的数据范围内的值。	`SELECT* FROM Test WHERE REGISTERTIME BETWEEN 1 AND 3;`
LIKE	在WHERE子句中搜索列中的指定模式。	`SELECT* FROM Test WHERE USERID LIKE 'USER%';`
IS NULL	空值判断	`SELECT* FROM Test WHERE USERID IS NULL;`
IS NOT NULL	非空值判断	`SELECT* FROM Test WHERE USERID IS NOT NULL;`
AND	对两个或两个以上的条件记录数据进行过滤，如果第一个条件和第二个条件都满足，则AND运算符显示一条记录数据。	`SELECT* FROM Test WHERE GENDER = 'MALE' AND REGIONID = 'REGION_1';`
OR	对两个或两个以上的条件记录数据进行过滤，如果第一个条件和第二个条件中任何一个满足，则OR运算符显示一条记录数据。	`SELECT* FROM Test WHERE GENDER = 'MALE' OR REGIONID = 'REGION_1';`

GROUP BY说明

GROUP BY子句对查询结果进行分组，示例如下：

```
SELECT column_name, aggregate_function(column_name)
  FROM table_name
  WHERE column_name operator value
  GROUP BY column_name;
```

WINDOW说明

WINDOW子句允许您设置如何将具有相同主建的数据分组到窗口中，来进行聚合或连接等操作。窗口将向数据中添加了两个额外的系统列WINDOWSTART和WINDOWEND，提供了窗口边界。支持以下WINDOW类型：

- HOPPING window
  根据数据的时间戳将输入数据分组到固定大小的（可能）重叠的窗口。您必须为HOPPING window窗口指定窗口大小和滑动距离。以下语句表示如何创建HOPPING window的查询。

```
SELECT windowstart, windowend, item_id, SUM(quantity)
  FROM orders
  WINDOW HOPPING (SIZE 20 SECONDS, ADVANCE BY 5 SECONDS)
  GROUP BY item_id;
```

- TUMBLING window
  根据数据的时间戳将输入数据分组到固定大小、不重叠的窗口中。您必须指定滚动窗口的窗口大小。TUMBLING window是HOPPING window的一种特殊情况，其中窗口大小等于滑动距离。以下语句表示如何创建TUMBLING window的查询。

```
SELECT windowstart, windowend, item_id, SUM(quantity)
  FROM orders
  WINDOW TUMBLING (SIZE 20 SECONDS)
  GROUP BY item_id;
```

- Session window
  将输入数据分组到会话中，您必须为Session window窗口指定不活跃的时间间隔。例如，将不活跃的时间间隔设置为5分钟。对于给定的主键（例如alice），如果超过5分钟没有新的输入数据到达，则alice的当前会话将关闭，并且后续到达的alice数据都将标记为新会话的开始。以下语句表示如何创建Session window的查询。

```
SELECT windowstart, windowend, item_id, SUM(quantity)
  FROM orders
  WINDOW SESSION (20 SECONDS)
  GROUP BY item_id;
```

12.3.4. DML

12.3.4.1. CREATE CQ

连续查询（Continuous Query，简称CQ），对Lindorm流引擎中的实时数据进行计算（过滤、转换、增强、聚合），并将计算结果写入到另一个数据流表或者外表中，直到用户执行终止连续查询操作才会停止计算。本文介绍创建连续查询任务的语法。

语法

```
CREATE CQ cqName INSERT INTO tableName querySpecification
querySpecification
    : SELECT setQuantifier selectItem (',' selectItem)*
      (FROM relation (',' relation)*)
      (WINDOW  windowExpression)
      (WHERE where=booleanExpression)
      (GROUP BY groupBy)
    ;
```

参数说明

参数	是否必选	说明
cqName	是	创建连续查询任务的名称。
tableName	是	将计算结果写入数据流表或者外表的名称。
querySpecification	是	SELECT语法参数说明请参见SELECT。

示例

```
CREATE CQ GetErrors
INSERT INTO ErrorStream
SELECT log4j
FROM Log4ErrorWarningStream log4j WHERE log4j.level = 'ERROR';
```

12.3.4.2. LIST CQ

查看连续查询任务的详细信息。

语法

```
LIST CQ;
```

示例

```
LIST CQ;
```

返回结果如下：

```
Query ID                    | Stream Topic           | Query String
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
---------------
 CETAS_USERS_EXTERNAL_TABLE | USERS_EXTERNAL_TABLE | INSERT INTO ErrorStream SELECT log4j FROM Log4ErrorWarn
ingStream log4j WHERE log4j.level = 'ERROR'
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
---------------
```

12.3.4.3. TERMINATE CQ

终止正在运行的连续查询任务。

语法

```
TERMINATE CQ query;
```

参数说明

参数	是否必选	说明
query	是	连续查询任务的名称，可以通过 `LIST CQ` 查询。

示例

```
TERMINATE CQ CETAS_USERS_EXTERNAL_TABLE;
```

12.4. 最佳实践

12.4.1. 对流引擎数据进行实时统计

云原生多模数据库Lindorm流引擎是面向流式数据处理的引擎，提供了流式数据的存储和轻计算功能。本文介绍使用Lindorm流引擎对交易数据进行实时统计，并将统计结果（即每小时的订单数量和交易金额）存储至Lindorm宽表中。

前提条件

- 已安装Java环境，要求安装JDK 1.7及以上版本。
- 已将客户端IP地址添加至Lindorm实例的白名单中，具体操作请参见设置白名单。
- 已获取Lindorm流引擎的连接地址，具体操作请参见查看连接地址。

操作步骤

1. 创建Lindorm宽表。通过Lindorm-cli连接Lindorm宽表引擎，并在Lindorm宽表中创建一张表，具体操作请参见通过Lindorm-cli连接并使用宽表引擎。

   > **说明** 创建的Lindorm宽表用于存储交易数据的统计结果。

```
CREATE TABLE IF NOT EXISTS order_stat (
  biz VARCHAR,
  WINDOWSTART LONG,
  WINDOWEND LONG,
  total_order_price DOUBLE,
  count LONG,
  primary key (biz,WINDOWSTART)
);
```

2. 创建数据流表。通过Lindorm-cli连接Lindorm流引擎，具体操作请参见通过Lindorm-cli连接并使用Lindorm流引擎。

```
CREATE STREAM IF NOT EXISTS orders (
  `biz` STRING,
  `order_id` STRING,
  `price` DOUBLE,
  `detail` STRING,
  `timestamp` LONG
) WITH (
  value_format = 'json',
  key_value = 'order_id',
  stream_topic = 'order_topic',
  TIMESTAMP = 'timestamp'
);
```

3. 创建Serving表。Serving表对于Lindorm流引擎来说是一个外表，交易数据存储在Lindorm宽表中，通过以下语句来关联Lindorm宽表。

```
CREATE External Table IF NOT EXISTS lindorm_order_stat WITH (
  table_type = 'lindorm.table',
  table_name = 'order_stat',
  endpoint = 'ld-bp17pwu1541ia****-proxy-lindorm.lindorm.rds.aliyuncs.com:30020'
);
```

> **说明**
>
> - `table_type`：表示源数据存储在Lindorm宽表引擎中。
> - `table_name`：表示Lindorm宽表的表名。
> - `endpoint`：Lindorm宽表引擎的HBase兼容连接地址。

4. 创建流处理任务，统计出每小时的订单数量和交易金额。

```
CREATE CQ lindorm_order_state
Insert Into lindorm_order_stat
SELECT biz, SUM(price) AS total_order_price , COUNT(*) AS count from orders window TUMBLING ( size 1 hou
r) GROUP BY biz;
```

5. 通过Kafka API写入数据至数据流表中指定的Topic（即order_topic）。

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.codehaus.jettison.json.JSONObject;
import java.util.Properties;
import java.util.Random;
import java.util.UUID;
import java.util.concurrent.Future;
public class KafkaToLindormStreamDemo {
  public static void main(String[] args) {
    Properties props = new Properties();
    //设置Lindorm Stream Kafka地址
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "ld-bp17pwu1541ia****-proxy-stream-public.lindorm
.rds.aliyuncs.com:30080");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringS
erializer");
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.Strin
gSerializer");
    KafkaProducer<String, String> producer = new KafkaProducer<String, String>(props);
    //构造一个Kafka消息队列。
    String topic = "order_topic";
    try {
      for (int i = 0; i < 100; i++) {
        JSONObject json = new JSONObject();
        json.put("biz", "biz1");
        json.put("order_id", UUID.randomUUID().toString());
        json.put("price", new Random().nextInt(1000));
        json.put("detail", "order detail");
        json.put("timestamp", System.currentTimeMillis() - i * 30 * 60 * 1000);
        Future<RecordMetadata> future = producer.send(
            new ProducerRecord<String, String>(topic, json.getString("order_id"),
                json.toString()));
        producer.flush();
        try {
          RecordMetadata recordMetadata = future.get();
          System.out.println("Produce ok:" + recordMetadata.toString());
        } catch (Throwable t) {
          System.out.println("Produce exception " + t.getMessage());
          t.printStackTrace();
        }
      }
    } catch (Exception e) {
      System.out.println("Produce exception " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

6. 通过Lindorm-cli连接Lindorm宽表引擎并查询统计结果。

```
SELECT * FROM order_stat;
```

返回结果如下：

```
+------+---------------+---------------+-------------------+-------+
| biz  |  WINDOWSTART  |   WINDOWEND   | total_order_price | count |
+------+---------------+---------------+-------------------+-------+
| biz1 | 1645869600000 | 1645873200000 | 694.000000        | 1     |
| biz1 | 1645873200000 | 1645876800000 | 1170.000000       | 2     |
| biz1 | 1645876800000 | 1645880400000 | 453.000000        | 2     |
| biz1 | 1645880400000 | 1645884000000 | 958.000000        | 2     |
| biz1 | 1645884000000 | 1645887600000 | 365.000000        | 2     |
| biz1 | 1645887600000 | 1645891200000 | 674.000000        | 2     |
| biz1 | 1645891200000 | 1645894800000 | 1467.000000       | 2     |
| biz1 | 1645894800000 | 1645898400000 | 1430.000000       | 2     |
| biz1 | 1645898400000 | 1645902000000 | 422.000000        | 2     |
| biz1 | 1645902000000 | 1645905600000 | 822.000000        | 2     |
| biz1 | 1645905600000 | 1645909200000 | 1521.000000       | 2     |
| biz1 | 1645909200000 | 1645912800000 | 930.000000        | 2     |
| biz1 | 1645912800000 | 1645916400000 | 1501.000000       | 2     |
```

13.Ganos时空引擎

Lindorm Ganos是阿里云推出的一款包含管理「空间几何数据」、「时空轨迹」、「专题栅格」、「遥感影像」的时空大数据引擎系统。系统兼容开源GeoMesa、GeoServer等生态，内置了高效的时空索引算法、空间拓扑几何算法、遥感影像处理算法等，结合阿里云Lindorm宽表引擎强大的分布式存储能力以及DAL Spark分析平台能力，可广泛应用于空间/时空/遥感大数据存储、查询、分析与数据挖掘场景。

- 提供时空几何能力支持空间几何、时空轨迹的数据表达和建模；
- 时空几何对象的创建、写入、索引、查询、删除等管理能力；
- 支持与DLA Spark的集成分析，提供兼容OGC标准的GeoSQL访问；

应用场景

- 互联网出行

互联网出行涉及到运力的调度、拼车、供需预测、热力图等业务。以供需预测为例，基于对历史轨迹数据的分析，并结合实时订单数据，预测当前订单密集区域的分布，提高接单概率并减少司机空驶时间。这背后涉及到大量时空型数据和业务信息的快速读取，并结合业务算法进行预测，利用Lindorm Ganos可有力支持该业务场景。

- 传感网与实时GIS

在环保、气象、水利、航海航空监测等领域，需要通过各种传感器获取天、空、地、海不同地理现象、事件、要素的全生命周期多尺度监测指标，比如污染监测、水位监测、降雨量监测、航标监测等。Lindorm Ganos可以为构建实时GIS大数据应用提供稳定、可靠、弹性、免运维的PaaS服务，为地理国情常态化监测和智慧城市建设提供基础平台。

- IoT

IoT行业产生的部分数据兼具时序和空间特征。以车联网为例，海量的车辆终端在不断地产生轨迹数据，轨迹数据包含了时间和空间位置。利用Lindorm Ganos，实时监测车辆的行驶轨迹、是否偏航、是否进入某个限制区域等。除了实时监控外，还可以进行实时时空查询，如查询某段时间的轨迹，某段时间进入该区域的车辆等。结合大数据分析框架（如Spark）还可以进行穿越分析、区域分布热力图等。

- 智慧物流与外卖递送

在物流与外卖等领域，需要实时监控车辆、骑手的位置，以便进行可靠的时间预测等服务。车辆和骑手的位置需要实时上报，云端需要处理高并发写入并进行实时路径规划、偏航监测等计算，背后都需要大量的时空计算。

Lindorm Ganos优势

- 公有云原生架构的空间、时空、遥感一体化NoSQL大数据引擎产品，开箱即用；
- PB级存储、高并发写入、百亿级时空查询秒级响应；
- 支持数据冷热分离存储和数据高效压缩算法，海量数据存储成本低；
- 与云上DLA Spark无缝集成，快速搭建空间大数据仓库和空间大数据分析平台；
- 基于OGC标准设计，便于系统间的集成与互操作；











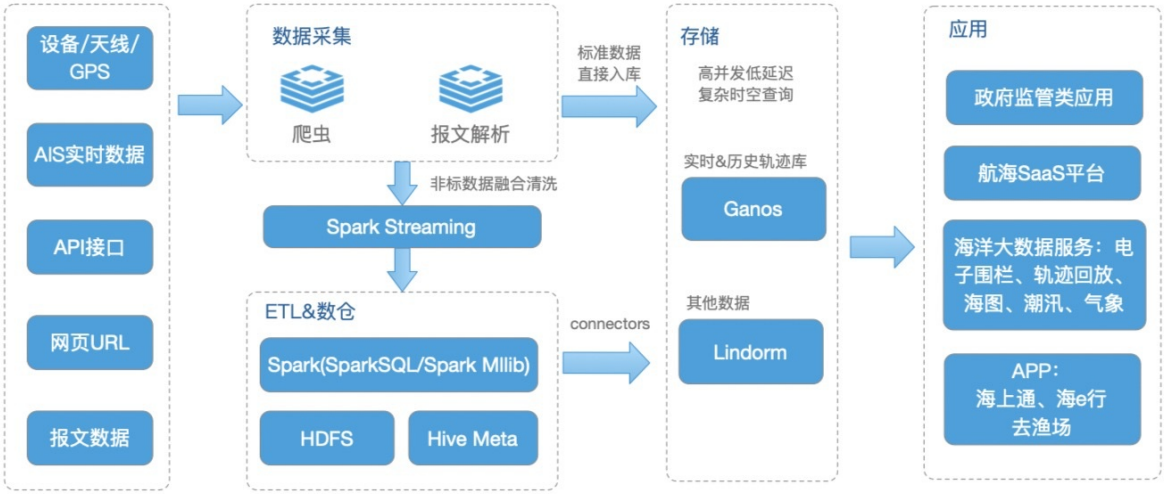




典型案例&架构

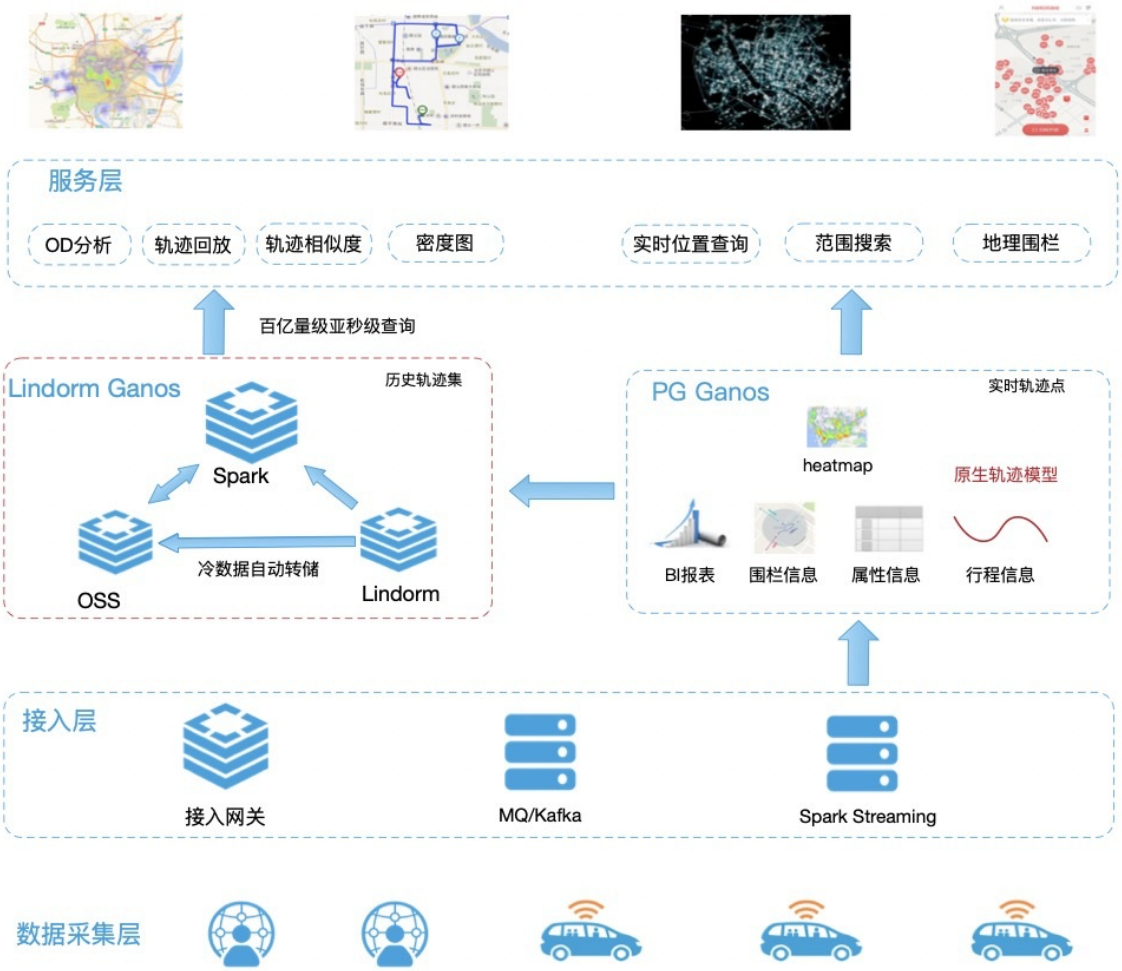
1、航海船舶大数据平台

- 场景：全球AIS船舶实时位置数据，每天上亿条轨迹写入，要求轨迹时空/时序查询秒级响应，具体包括：
  - 监管区域实时地理围栏判断
  - 轨迹区域回放（指定区域+指定时间段）
  - 轨迹回放（指定时间段查询）
- 架构说明
  - AIS等数据经过Spark平台清洗、融合完成后，构建数仓。
  - 对外提供查询的数据存储到Lindorm Ganos中，可供前端各类应用进行时空查询（如地理围栏、轨迹回放等）。
  - 整体系统的查询时间在毫秒级，数据清洗在分钟级。



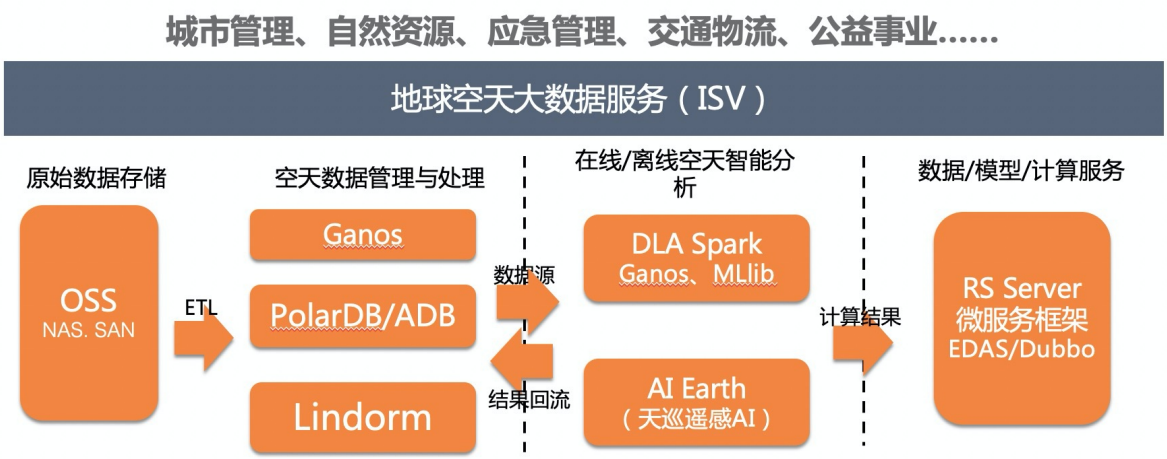
2、某位置服务平台

- 场景：实时采集车辆设备轨迹，40万+终端设备，3万+轨迹点/秒峰值；对外提供实时位置查询、地理围栏、轨迹相似度判断等服务。
- 架构说明：
  - Lindorm Ganos承载所有历史轨迹库的存储。
  - 当历史数据量较大时，可通过冷数据自动转储功能，将数据存放到oss中，减少70%以上的成本。
  - 百亿量级的轨迹查询（比如查询某段时间、某空间范围内经过的轨迹）只需亚秒级。
  - 与spark平台无缝衔接，可以提供类似轨迹相似度、OD分析等强大功能。



3、遥感大数据管理与智能服务平台

- 场景：遥感影像数据入库、查询、分析和遥感AI智能服务。
- 痛点：
  - 遥感影像数据体量大且增量快，存储扩展不灵活，成本高昂，管理复杂。
  - 传统静态切片方式管理显示速度快，但无法支撑分析场景和AI场景。
  - 多源空间数据融合查询分析困难。
- 架构说明：
  - 原始数据采用OSS高性价比存储，冷热分离。
  - 影像数据库采用SQL+NoSQL构建时空序列影像，容量无上限，存储与计算分离。
  - 原生提供基于“分块”理念的数据存储和组织方式，保留原始像元信息，有效支撑各类分析计算场景。
  - 与Spark集成，ETL与多源空间数据分析处理更便捷。



13.1. 引擎简介

Lindorm Ganos是阿里云推出的一款管理空间几何数据、时空轨迹的时空大数据引擎系统。
系统兼容开源GeoMesa、GeoServer等生态，内置了高效的时空索引算法、空间拓扑几何算法等，结合阿里云Lindorm宽表引擎强大的分布式存储能力以及DLA Spark分析平台能力，可广泛应用于空间或时空存储、查询、分析与数据挖掘场景。

- 提供时空几何能力支持空间几何、时空轨迹的数据表达和建模。
- 时空几何对象的创建、写入、索引、查询、删除等管理能力。
- 支持与DLA Spark的集成分析，提供兼容OGC标准的GeoSQL访问。

Lindorm Ganos可以作为DLA Ganos的数据层，借助DLA Ganos实现对遥感影像的处理，更多信息，请参见时空栅格。

应用场景

- 互联网出行
  适应于互联网出行高并发轨迹数据写入、历史轨迹存储、订单轨迹查询、时空范围查询等场景；向上可以支撑互联网出行涉及到运力的调度、拼车、供需预测、热力图等业务。
- 航海船舶大数据
  适应于航海船舶AIS数据的实时写入、历史船舶轨迹存储、时空区域回放（时空范围查询）等场景。
- IoT
  IoT行业产生的部分数据兼具时序和空间特征。以车联网为例，海量的车辆终端在不断地产生轨迹数据，轨迹数据包含了时间和空间位置。利用Lindorm Ganos，实时监测车辆的行驶轨迹、是否偏航、是否进入某个限制区域等。除了实时监控外，还可以进行实时时空查询，如查询某段时间的轨迹，某段时间进入该区域的车辆等。结合大数据分析框架（如Spark）还可以进行穿越分析、区域分布热力图等。
- 智慧物流与外卖递送
  在物流与外卖等领域，需要实时监控车辆、骑手的位置，以便进行可靠的时间预测等服务。车辆和骑手的位置需要实时上报，云端需要处理高并发写入并进行实时路径规划、偏航监测等计算。

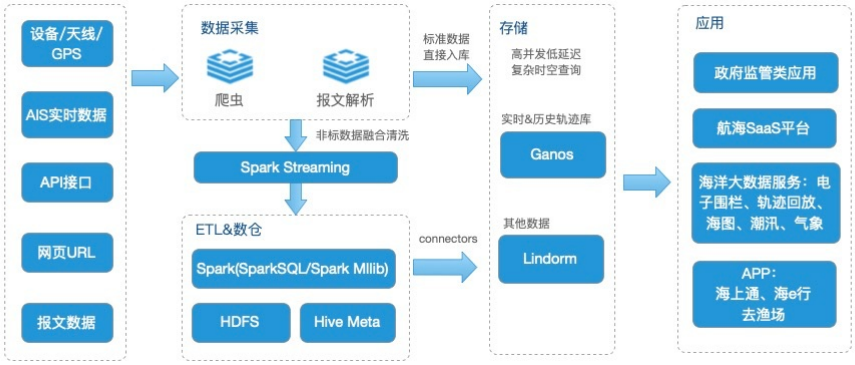
Lindorm Ganos优势

- 公有云原生架构的空间、时空的NoSQL大数据引擎产品，开箱即用。
- PB级存储、高并发写入、百亿级时空查询秒级响应。
- 支持数据冷热分离存储和数据高效压缩算法，海量数据存储成本低。
- 与云上DLA Spark无缝集成，快速搭建空间大数据仓库和空间大数据分析平台。
- 兼容OGC标准。

典型案例&架构

- 航海船舶大数据平台

- 场景
  全球AIS船舶实时位置数据，每天上亿条轨迹写入，要求轨迹时空或时序查询秒级响应，具体包括：
  - 监管区域实时地理围栏判断。
  - 轨迹区域回放（指定区域+指定时间段）。
  - 轨迹回放（指定时间段查询）。
- 架构说明
  - AIS等数据经过Spark平台清洗、融合完成后，构建数仓。
  - 清洗后的数据存储到Lindorm Ganos中，可供前端各类应用进行时空查询（如地理围栏、轨迹回放等）。
  - 整体系统的查询时间在毫秒级，数据清洗在分钟级。

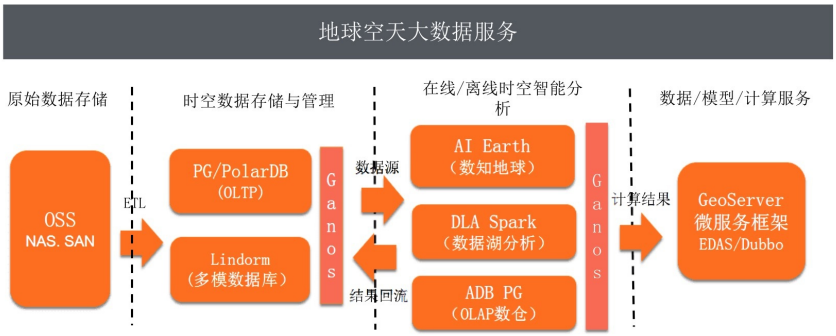


- 某位置服务平台
  - 场景
    实时采集车辆设备轨迹，40万+终端设备，3万+轨迹点/秒峰值；对外提供实时位置查询、地理围栏、轨迹相似度判断等服务。
  - 架构说明
    - Lindorm Ganos承载所有历史轨迹库的存储。
    - 当历史数据量较大时，可通过冷数据自动转储功能，将数据存放到OSS中，减少70%以上的成本。
    - 百亿量级的轨迹查询（比如查询某段时间、某空间范围内经过的轨迹）只需亚秒级。
    - 与Spark平台集成，可以提供轨迹的相关分析功能。



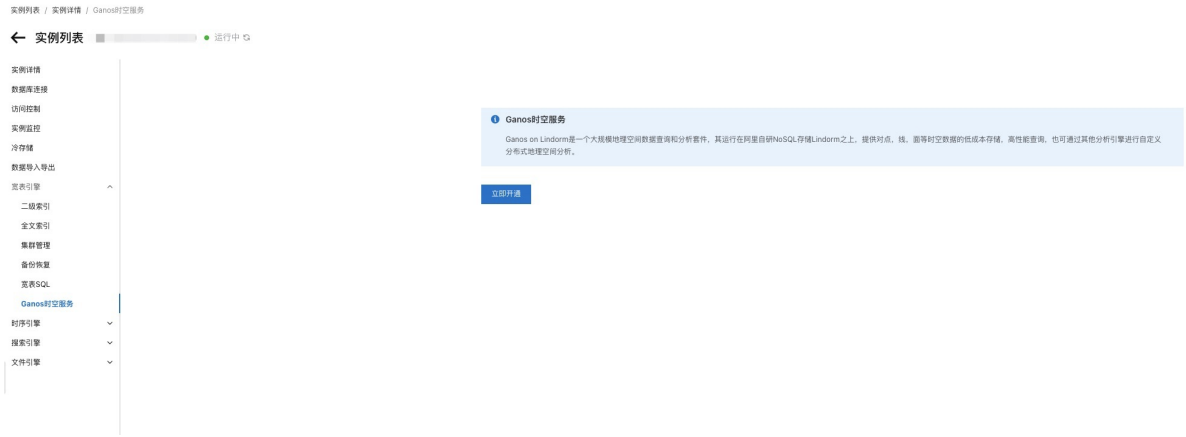
- 遥感大数据管理与智能服务平台
  - 场景
    遥感影像数据入库、查询、分析和遥感AI智能服务。
  - 痛点：
    - 遥感影像数据体量大且增量快，存储扩展不灵活，成本高昂，管理复杂。
    - 传统静态切片方式管理显示速度快，但由于其丢失了元数据信息，无法支撑分析场景和AI场景。
    - 多源空间数据融合查询分析困难。

- 架构说明
  - 原始数据采用OSS、NAS、SAN等存储，冷热分离。
  - 影像数据库使用PolarDB存储元数据信息，Lindorm存放瓦片信息数据，具有极高扩展性。
  - 原生提供基于“分块”理念的数据存储和组织方式，保留原始像元信息，有效支撑各类分析计算场景。
  - 与Spark集成，ETL与多源空间数据分析处理更便捷。



13.2. 开通指南

Lindorm Ganos作为Lindorm宽表引擎的一个组件，**本身不收取任何额外费用**，只收取Lindorm宽表引擎费用。用户在购买Lindorm宽表引擎之后，即可在控制台中开通。



13.3. 时空几何

13.3.1. 基本概念

本文介绍在开发过程中涉及到的一些概念。
Lindorm Ganos中所称的时空几何，其范畴包含以下几方面：

- 时空几何对象
  - 矢量数据，如点、线、面状要素
  - 在矢量数据基础上结合时间属性，组成的时空数据（或时空轨迹数据）
- 针对时空几何对象的相关操作，如时空关系判断
- 时空索引

Lindorm Ganos能够提供优异的查询性能，其背后的机制在于为时空数据建立了高效的时空索引。在Lindorm Ganos中，时空索引以HBase的Rowkey形式存在，请参见创建索引表。

- 时空关系

所谓的时空关系是指两个时空几何对象之间的时间&空间的相对位置。典型的时空关系包括：相交、相离、覆盖、包含等。 在现实场景中，常说的「地理围栏判断」是指一个面状要素表示的地理围栏与目标对象（点、线、面）之间的关系。如果目标对象在地理围栏之内，则称之为包含；在面状要素之外，则称之为相离。 Lindorm Ganos借助CQL工具来描述时空关系，请参见查询时空对象。

- OGC

OGC全称是开放地理空间信息联盟（Open Geospatial Consortium），是一个非盈利的国际标准组织，它制定了数据模型和相关操作的一系列标准，GIS厂商按照这个标准进行开发可保证空间数据的互操作。

- GeoTools

Lindorm Ganos的SDK是基于GeoTools实现的。GeoTools是一个遵循OGC标准，用于处理地理空间数据的工具包，实现了OGC标准的数据模型和接口，很多地理工具都基于GeoTools开发，详情参考：https://geotools.org/。

- Geometry

在OGC的定义中，Geometry用来表示一个空间对象，例如空间点对象、空间线对象、空间面对象。Geometry只包含空间对象的位置信息，并不包含其附带的属性信息。 GeoTools提供了工具来帮助构建Geometry，因此在Lindorm Ganos开发过程中，可以较为方便的使用。

- SimpleFeature

简单要素。 通俗来讲，SimpleFeature包含Geometry以及其他属性信息。通常所说的一个轨迹点就是一个SimpleFeature，包含了该轨迹点的空间位置、时间信息以及其他属性信息，其中时间信息也是作为属性信息的一部分。

- CQL&ECQL

CQL全称为Common Query Language，是OGC为方便地理服务的查询而定义的查询语言。ECQL全称Extended Common Query Language，是CQL的扩展版，比CQL更强大。 一般来说，ECQL更多的是定义filter，类似于SQL语言的where子句，通过文本描述的方式来筛选出目标对象。参考「**查询时空对象**」一章。 在本手册中，为了简单起见，所说的CQL就是ECQL。

- WKT

WKT全称Well-known text，是OGC定义的一种用文本来描述空间对象的格式。例如点就可以写成POINT(0,0)，这在查询语句中经常使用，也容易理解，CQL&ECQL中也是用WKT来表示空间对象的。WKT的详细规范可参考：http://www.opengeospatial.org/standards/wkt-c

- WKB

WKB全称Well-known Binary，是OGC定义的一种通过序列化字节来描述几何对象的格式。与WKT相比，其优点在于数据较小，适宜传输。GeoTools提供了工具可用于WKB与WKT之间的转换。

13.3.2. 快速入门

在Lindorm Ganos开通后通过Ganos接口进行开发工作。

操作步骤

1. 代码示例下载，lindorm-ganos-sample。
2. 下载解压后进入lindorm-sample目录，执行以下语句。

   ```
   mvn install -Dmaven.test.skip=true -s ../settings.xml
   ```

   编译成功后，在target目录下生成 `lindorm-sample-1.0.0.jar` 。
3. 代码示例说明。以AIS数据为例，展示了对轨迹数据的写入、查询功能。
   - Main: 示例程序入口。
     通过JSON文件指定操作和参数，JSON文件可参照示例sample_template.json。
   - Writer：向Lindorm Ganos写入数据，支持两种写入模式：
     - WRITER：单线程写入。连接Lindorm Ganos，定义Schema，创建时空索引表，读取示例数据并写入。
     - PWRITER：多线程写入。连接Lindorm Ganos，定义Schema，创建时空索引表。指定线程数，每个线程随机生成指定范围内的n个AIS点，写入已创建的表中。需通过JSON文件指定参数。JSON文件可参照示例write_template.json。
   - Reader：查询接口，对Writer写入的AIS数据进行查询，支持两种查询模式：
     - READER：进行固定参数的属性查询，空间查询和时空查询等示例。
     - UDFQUERY：通过JSON文件指定参数，自定义选择空间/时空/属性查询，指定查询条件。JSON文件可参照示例query_template.json。
4. 使用示例。以下示例中 `zk_address` 的地址为Lindorm宽表引擎的**使用HBase Java API访问**地址。
   - **写入数据**

在sample_template.json中，设置写入参数。

```
{
  "operation" : "WRITER",
  "zk_address" : $your_zk_address,
  "catalog" : "example",
  "schema" : "ship"
}
```

执行示例写入。

```
java -jar lindorm-sample-1.0.0.jar path_to_sample/sample_template.json
```

默认创建schema为ship的4个索引表 (index on id, z2, z3, attr) 。若要创建指定的索引表，可以参见创建索引表。

- **查询数据**
  在sample_template.json中设置查询参数，使用示例代码默认的查询范围。

```
{
  "operation" : "READER",
  "zk_address" : $your_zk_address,
  "catalog" : "example",
  "schema" : "ship"
}
```

也可设置查询语句filter_string，指定查询范围。

```
{
  "operation" : "READER",
  "filter_string" : "bbox(geom, 20, 20, 40, 40)"
  "zk_address" : $your_zk_address,
  "catalog" : "example",
  "schema" : "ship"
}
```

执行示例查询。

```
java -jar lindorm-sample-1.0.0.jar path_to_sample/sample_template.json
```

- **并行写入数据**
  使用PWRITER。

```
{
  "operation" : "PWRITER",
  "zk_address" : $your_zk_address,
  "catalog" : "example",
  "schema" : "ship",
  "write_template_file" : "src_root/src/main/java/com/aliyun/apsaradb/lindorm/ganos/sample/write_templ
ate.json"
}
```

- **自定义查询**
  使用UDFQUERY。

```
{
  "operation" : "UDFQUERY",
  "zk_address" : $your_zk_address,
  "catalog" : "example",
  "schema" : "ship",
  "query_template_file" : "src_root/src/main/java/com/aliyun/apsaradb/lindorm/ganos/sample/query_templ
ate.json"
}
```

13.3.3. 创建索引表

Ganos内置了多个空间索引，用户只需在前端输入时空数据，并指定要建立的索引即可，不用再关心HBase的KV如何设计与构建，使用较为方便。因此，在数据写入之前，需要先定义好索引表结构。
Ganos目前支持五类索引，适应于不同的查询场景。**这些索引可以同时存在，也可以根据需求只创建某个索引**（如业务场景中只有周边范围查询，那么就只需要创建Z2索引，而不用创建其他索引）。

ID索引

适应于根据空间对象ID（称为FID）查询的场景，要求每个空间对象的FID必须唯一。

Z2/XZ2索引

适应于空间查询，如地理围栏判断、周边范围查询等；其中Z2是用于「点」对象；XZ2用于「线」、「面」对象。

Z3/XZ3/XZ2T索引

适应于时空查询，如某个空间范围以及时间段内的历史轨迹等；其中Z3是用于「点」对象；XZ3和XZ2T用于「线」、「面」对象，对于轨迹线模型查询，XZ2T索引对查询效率提升效果更佳。

XYZ索引

适应于含有经度、纬度、高程信息的三维索引，目前仅用于为「点」对象构建三维索引。

属性索引

适应于根据其他属性查询的场景。

建立索引的步骤

建立索引过程中，主要包含两个步骤：

1. 定义空间/时空数据类型，即SimpleFeatureType 。
    i. 类型schema名称，可以认为是索引表的别名。
    ii. schema具体内容，包含了属性，空间对象(Geometry)以及时间等列的定义。
    iii. 其他自定义信息，包括：
        a. 使用的压缩方式。
        b. 是否使用TWKB格式。
        c. 为某一列建立索引。
2. 根据定义好的空间/时空数据类型创建具体的对象。

创建索引示例

如前所述，Ganos的**索引可以同时存在，也可以根据需求只创建某个索引**（如业务场景中只有周边范围查询，那么就只需要创建Z2索引，而不用创建其他索引）。默认情况下是建立所有的索引，这样对所有的查询场景来说都能提供较高的性能，但会占用较多的存储空间。为了节省空间存储，可以根据查询场景选择合适的索引。Ganos内置了特定的Hints，用来在定义SimpleFeatureType的时候指定对应的索引，语法如下：

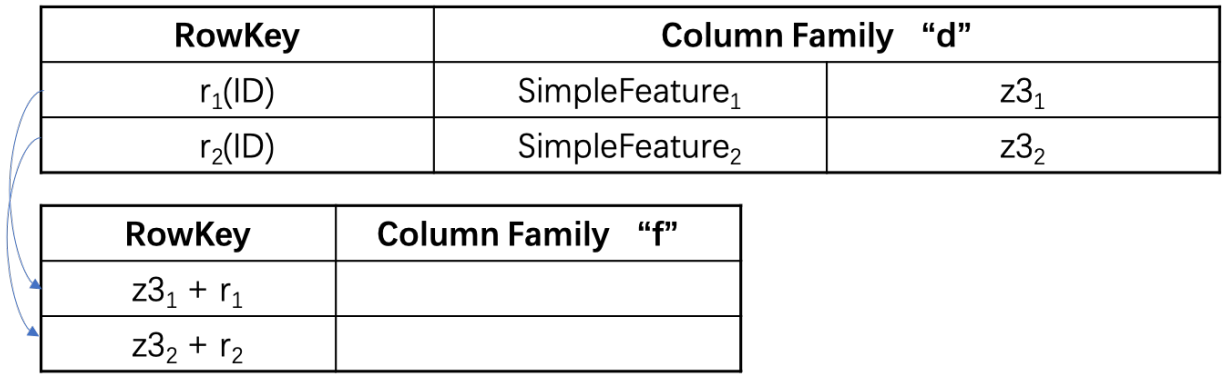
```
// sft为SimpleFeatureType的一个实例对象
sft.getUserData().put("geomesa.indices.enabled", "{index_name}:{col1}:{col2}:...,{index_name}:{col}");
```

- index_name取下面常量之一：id, attr, z2, z3, xz2, xz3, xyz。
- col为SimpleFeatureType中定义的列。
- 可以创建多个index，每个index之间用 `,` 分割。
- 可以将多个列添加到同一个索引中，索引和每个列之间用 `:` 分割。

下面这个示例中，一共创建了z3和attr两个索引，其中start列+dtg列为z3索引；end列+dtg列为z3索引；name列和dtg列为attr索引。

```
sft.getUserData().put("geomesa.indices.enabled", "z3:start:dtg,z3:end:dtg,attr:name:dtg");
```

时空二级索引
Lindorm Ganos利用Lindorm原生二级索引实现时空二级索引能力。常规二级索引表建立在主表的某个属性列上，但对序列化后的时空属性列直接建立二级索引是没有实际意义的。因此，需要先在主表上加入时空编码列（该列存储时空属性的空间填充曲线编码值），再对索引列建立二级索引。二级索引存储结构示例：



该示例中对以ID为RowKey的主表创建了Z3时空索引列，并在时空索引列上建立二级索引表。默认二级索引表中没有冗余列。冗余列支持待后续开放。

时空二级索引创建示例

在创建主表同时指定需要创建的时空二级索引类型

对已定义的SimpleFeatureType（示例中的sft）指定userData参数

```
// sft为SimpleFeatureType的一个实例对象
sft.getUserData().put("geomesa.secondary.enabled","true");
sft.getUserData().put("geomesa.primary.index","id");
sft.getUserData().put("geomesa.indices.enabled","id,z3");
```

调用createSchema创建表后，会生成如上图结构所示的ID主表和z3二级索引表

在已经存在的主表上建立二级索引

调用LindormDataStore.createIndex方法，在已创建且"geomesa.secondary.enabled"参数为true的主表上，创建二级索引

```
String index = "attr:dtg";
lindormDS.createIndex(schemaName, index);
```

注意：目前不支持创建新索引表的同时同步历史数据

13.3.4. 创建时空对象

在轨迹场景下，时空对象可简单理解为"轨迹点"，包含空间位置、时间、其他属性信息。在Lindorm Ganos SDK中，时空对象映射的就是SimpleFeature，定义好索引结构后，即可创建。

每个SimpleFeature由ID，Geometry、时间信息以及其他属性构成。在「**快速入门**」代码示例中封装了GanosClient类，包含索引创建、轨迹点创建等功能，用于简化用户代码量，建议直接使用。若要使用原生的GeoTools API，可通过SimpleFeatureBuilder类来创建SimpleFeature对象，如下：

```
String spec = "name:String,dtg:Date,*geom:Point:srid=4326;geomesa.logical.time='false'";
SimpleFeatureType sft = SimpleFeatureTypes.createType("mySft", spec);
SimpleFeatureBuilder sfBuilder = new SimpleFeatureBuilder(sft);
//builder.set("属性名", 属性值);
builder.set("geom", Geometry);  //设置空间对象，"geom"为固定写法
SimpleFeature feature = builder.buildFeature(object_id + "_" + date.getTime());
```

设置Geometry对象

在创建SimpleFeature环节中，需要设置geometry对象（见上面代码）。Geometry是SimpleFeature的一部分，存储要素的空间信息。这里所说的“要素”，包括点要素、线要素、面要素等，在时空轨迹的场景下可简单理解为“轨迹点”。GeoTools API提供了GeometryFactory工具类帮助用户创建Geometry对象，具体可以通过两种方式创建：

- 通过Coordinate对象

这种方式创建比较直观，该对象表示一个坐标点，推荐使用。

- 通过WKT文本创建

WKT(Well-known text)是一种空间对象的文本标记语言，如字符串"POINT (1 1)"表示一个点状要素，其坐标为1，1；字符串"LINESTRING(0 2, 2 0, 8 6)"表示一条线状要素，由三个坐标点组成；字符串"POLYGON((20 10, 30 0, 40 10, 30 20, 20 10))"表示一个面状要素，最首尾两个坐标点相同构成一个环。详细参考：https://en.wikipedia.org/wiki/Well-known_text

点要素

以下两种创建方式任选其一：

- 通过Coordinate对象创建

```
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory();
Coordinate coord = new Coordinate(1, 1);
Point point = geometryFactory.createPoint(coord);
```

- 通过WKT创建

```
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory();
 WKTReader reader = new WKTReader(geometryFactory);
 Point point = (Point) reader.read("POINT (1 1)");
```

线要素

以下两种创建方式任选其一：

- 通过Coordinate对象

```
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory();
Coordinate[] coords  =
 new Coordinate[] {new Coordinate(0, 2), new Coordinate(2, 0), new Coordinate(8, 6) };
LineString line = geometryFactory.createLineString(coordinates);
```

- 通过WKT描述：

```
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory();
WKTReader reader = new WKTReader( geometryFactory );
LineString line = (LineString) reader.read("LINESTRING(0 2, 2 0, 8 6)");
```

面要素

以下两种创建方式任选其一：

- 通过Coordinate对象

```
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory();
Coordinate[] coords  =
   new Coordinate[] {new Coordinate(4, 0), new Coordinate(2, 2),
                     new Coordinate(4, 4), new Coordinate(6, 2), new Coordinate(4, 0) };
LinearRing ring = geometryFactory.createLinearRing( coords );
LinearRing holes[] = null;
Polygon polygon = geometryFactory.createPolygon(ring, holes );
```

- 通过WKT描述：

```
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory( null );
WKTReader reader = new WKTReader( geometryFactory );
Polygon polygon = (Polygon) reader.read("POLYGON((20 10, 30 0, 40 10, 30 20, 20 10))");
```

设置其他属性

可以通过SimpleFeature的UserData字段设置其他自定义信息，比如Lindorm Ganos内置了诸多Hints，包括：

- 使用自定义FeautureID(可节省存储空间)

```
SimpleFeature feature =...
feature.getUserData().put(Hints.USE_PROVIDED_FID, java.lang.Boolean.TRUE);
```

13.3.5. 写入时空对象

在索引表创建完成、时空对象创建完成后，即可调用数据写入接口将数据写入到Lindorm Ganos中。

单条记录写入

Lindorm Ganos通过GeoTools API中的SimpleFeatureWriter写入单条数据，SimpleFeatureWriter支持事务，可以通过DataStore的getFeatureWriterAppend方法获取。

注：构造SimpleFeatureType以及SimpleFeature可参考「快速入门」

```
/**
     * 写入单条数据
     * @param sft SimpleFeatureType
     * @param feature 待写入的SimpleFeature
     * @return 返回1，否则抛出错误
     * @throws Exception
     */
public int upsert(SimpleFeatureType sft, SimpleFeature feature) throws Exception  {
        try{
            SimpleFeatureWriter writer=(SimpleFeatureWriter)ds.getFeatureWriterAppend(sft.getTypeName(), Tra
nsaction.AUTO_COMMIT);
            SimpleFeature toWrite=writer.next();
            toWrite.setAttributes(feature.getAttributes());
            toWrite.getUserData().putAll(feature.getUserData());
            writer.write();
            writer.close();
            return 1;
        } catch (Exception e){
            throw new Exception("写入数据错误" + e.getMessage());
        }
}
```

批量写入

Lindorm Ganos支持批量插入SimpleFeature，通过GeoTools API中的SimpleFeatureStore类实现。

注：构造SimpleFeatureType以及SimpleFeature可参考「快速入门」

```
   /**
     * 批量写入数据
     *
     * @param sft SimpleFeatureType
     * @param features 待写入的SimpleFeature对象集合
     * @return 返回实际写入成功的条数
     * @throws Exception
     */
    public int upsert(SimpleFeatureType sft, List<SimpleFeature> features) throws Exception  {
        try{
            SimpleFeatureStore featureStore = (SimpleFeatureStore) ds.getFeatureSource(sft.getTypeName());
            List<FeatureId> featureIds = featureStore.addFeatures(new ListFeatureCollection(sft,features));
            return featureIds.size();
        }  catch (Exception e){
            throw new Exception("写入数据错误" + e.getMessage());
        }
    }
```

13.3.6. 查询时空对象

Ganos的数据查询采用GeoTools Query对象来完成，Query中的查询条件是通过构建CQL查询语句来实现的。同时Query还有更多强大的功能，可以指定返回的属性列、对某一属性进行排序等。返回的数据是以SimpleFeatureCollection对象存在，可以通过迭代器遍历所有的SimpleFeature，进而解析出需要的数据。下面会分步介绍详细流程，所有介绍的流程均可参考「**快速入门**」代码示例GanosClient类。

构造查询条件

查询条件也称之为过滤器（Filter），采用CQL语言描述。CQL功能类似SQL语句的WHERE子句，根据这些条件可以过滤数据。

空间查询

- 空间查询支持如下谓词，表中的「Expression」表示WKT表示的空间对象（点、线、面）：

语法	描述
INTERSECTS(Expression , Expression)	判断两个空间对象是否**相交**，比如两条道路是否相交。
DISJOINT(Expression , Expression)	判断两个空间对象是否**相离**。
CONTAINS(Expression , Expression)	判断第一个空间对象是否**包含**第二个空间对象。
WITHIN(Expression , Expression)	与CONTAINS谓词相反，判断第一个对象是否被第二个空间对象包含。
TOUCHES(Expression , Expression)	两要素边界相交但内部不相交。
CROSSES(Expression , Expression)	判断两个空间要素是否相交，要求仅有部分交集而不是完全包含。
BBOX (Expression , Xmin , Ymin , Xmax , Ymax [, CRS])	空间对象[Expression]是否与[Xmin,Ymin,Xmax,Ymax]所组成的四边形相交。可选的CRS是空间参考系，包含SRS代码的字符串（例如，'EPSG：1234'。默认使用查询图层的CRS）。

- 案例1：获取某个空间范围（120E，30N，130E，40N）所包含的空间对象（餐馆、船舶、车辆等）。

```
//获取DataStore
DataStore ds = DataStoreFinder.getDataStore(params);
//设置查询条件
String schema = "schmaName";
String stFilter = "bbox(geom, 120,30,130,40)"
Query query = new Query(schema, ECQL.toFilter(stFilter));
//获取查询结果
SimpleFeatureCollection features=ds.getFeatureSource(schema).getFeatures(query);
```

- 案例2：获取周边5公里范围内的所有车辆。
  - 首先将周边5公里的范围构造成空间对象（面），假设为（46.9 48.9, 47.1 48.9, 47.1 49.1, 46.9 49.1, 46.9 48.9）。
  - 使用contains谓词将所有对象过滤出来。

```
//获取DataStore
DataStore ds = DataStoreFinder.getDataStore(params);
//设置查询条件
String schema = "schmaName";
String stFilter =  "contains('POLYGON ((46.9 48.9, 47.1 48.9, 47.1 49.1, 46.9 49.1, 46.9 48.9))', geom)"
Query query = new Query(schema, ECQL.toFilter(stFilter));
//获取查询结果
SimpleFeatureCollection features=ds.getFeatureSource(schema).getFeatures(query);
```

时空查询

除了空间查询场景以外，还存在大量的时空查询需求。如查询过去2小时内，位于（20E，30N，130E，40N）范围内所有的车辆轨迹。
Lindorm Ganos支持的时间查询谓词如下，表中的「Expression」为时间列名，「Time」为字符串表示的时间（UTC）：

语法	描述
Expression BEFORE Time	时间在Time之前。
Expression BEFORE OR DURING Time Period	时间在 Time Period之前或包含在 Time Period之中。
Expression DURING Time Period	时间在Time Period之中。

语法	描述
Expression DURING OR AFTER Time Period	时间在Time Period之中或之后。
Expression AFTER Time	时间在Time Period之后。

Lindorm Ganos支持的时间表达方式有多种：

语法	描述
Time / Time	由起始时间和结束时间定义的区间。
Duration / Time	在Time之前的时间区间。
Time / Duration	在Time之后的时间区间。

- 案例：如用户想查询位于（120E，30N，130E，40N）之中，时间介于2014-01-01T11:45:00与2014-01-01T12:15:00之间的车辆轨迹，那么需要将空间查询和时间查询结合在一起，如下：

```
//设置查询条件
String schema = "schmaName";
String stFilter = "bbox(geom, 120,30,130,40) AND dtg DURING 2014-01-01T11:45:00.000Z/2014-01-01T12:15:00.000Z";
Query query = new Query(schema, ECQL.toFilter(stFilter));
//获取查询结果
SimpleFeatureCollection features=ds.getFeatureSource(schema).getFeatures(query);
```

属性查询

- 比较运算符包括： =，<>，>，>=，<，<=，如要选取人口大于1500万的城市，条件为：PERSONS>15000000，其中PERSONS为人口数量的字段，后面不再单独说明。
- BETWEEN表示从一个范围到另一个范围，如PERSONS BETWEEN 1000000 AND 3000000。
- 比较运算符支持文本值，可在运算符=的右侧指定文本值，如 CITY_NAME='Beijing'，表示选择北京市；也可以使用LIKE操作符，与SQL中的用法一样，如 CITY_NAME LIKE 'N%'会选择所有以N开头的城市。
- 对两个属性进行比较，如 MALE > FEMALE，选择男性多余女性的城市。
- 算术表达式可以使用 +, -, *, /构成，如过滤条件 UNEMPLOY/(EMPLOYED + UNEMPLOY) > 0.07选择所有失业率大于7%的城市。
- 使用IN操作符，可选择属性在给定值范围内的，与SQL用法相同，如 ID IN ('cities.1', 'cities.12')，或查找名字在给定值内的城市，可使用 CITY_NAME IN ('Beijing','Shanghai', 'Guangzhou')。
- 可使用Geoserver中的任何过滤函数，如 strToLowerCase（CITY_NAME） like '%m%'，表示名字中包含m的城市，不区分M的大小写。
- 几何过滤，使用BBOX，比如BBOX(the_geom, -90, 40, -60, 45)，表示选择在（-90, 40, -60, 45）范围内的城市。

```
//设置查询条件
String schema = "schemaName";
String filter = "name = 'bob'";
Query query = new Query(schema, ECQL.toFilter(filter));
//获取查询结果
SimpleFeatureCollection features=ds.getFeatureSource(schema).getFeatures(query);
```

指定返回列名

用户可以通过配置Query对象参数指定具体返回哪些列，如：

```
//指定返回字段名
String[] returnFields={"geom"};
//设置查询条件
String schema = "schmaName";
String stFilter = "bbox(geom, 120,30,130,40)"
Query query = new Query(schema, ECQL.toFilter(stFilter));
query.setPropertyNames(returnFields);
//获取查询结果集
SimpleFeatureCollection features=ds.getFeatureSource(schema).getFeatures(query);
```

指定排序方式

用户可以通过构建SortBy对象参数指定具体返回哪些列，如：

```
String sortField="name" //指定排序字段名
FilterFactory2 ff = CommonFactoryFinder.getFilterFactory2();
SortBy[] sort = new SortBy[]{ff.sort(sortField, order)};
query.setSortBy(sort);
SimpleFeatureCollection features=ds.getFeatureSource(schema).getFeatures(query);
```

13.3.7. 删除时空对象

一般情况下，轨迹数据都是以“追加”的方式写入到数据库中的。但在一些场景中，也会有删除时空对象的需求，Lindorm Ganos支持按时间段、按时空范围、按时空对象ID三种删除方式。
Lindorm Ganos支持使用HBase TTL技术来删除过期的数据，目前支持直接在HBase Shell中修改时空索引表的TTL属性来实现。如果建立了多个索引表，那么需要对每张索引表均执行该操作。

例如，现有时空索引表xxx_z3和属性索引表xxx_attr，对5分钟之前'd'列的数据进行删除，那么需要执行语句：

```
Alter 'xxx_z3',{NAME=>'d',TTL=>300}
Alter 'xxx_attr',{NAME=>'d',TTL=>300}
```

根据时空范围删除

根据时空范围删除的方式，可以删除z2/xz2、z3/xz3这两类时空索引表，无法删除其他索引表的信息。如果此时还存在属性索引表，那么有可能造成时空索引表和属性索引表之间数据的不一致。建议当只有时空索引表时使用该方式进行删除。

具体步骤

- 构造filter，通过filter可以筛选出删除的对象，这个filter和查询时候构造的filter机制是一样的。
- 使用geotools的SimpleFeatureStore.removerFeatures接口删除符合filter的对象。

```
/**
 * 删除符合filter范围的数据
 * @param schema ： schema名称
 * @param filterString ： 字符串描述的filter，需要符合ECQL语法
 * @throws Exception
 */
public void removeByFilter(String schema, String filterString) throws Exception {
    SimpleFeatureStore fs = (SimpleFeatureStore) ds.getFeatureSource(schema);
    //如果filterString为空，表示全部删除
    if (StringUtils.isEmpty(filterString)){
        fs.removeFeatures(Filter.INCLUDE);
        return;
    }
    Filter filterToDelete = ECQL.toFilter(filterString);
    fs.removeFeatures(filterToDelete);
}
```

根据空间对象ID删除

根据空间对象ID删除的方式，将会删除ID索引表中的数据，无法删除其他索引表的信息。如果只有一个ID索引表，那么可以用该方式进行删除。

具体步骤：

使用geotools中的SimpleFeatureStore.removerFeatures接口删除指定的ID对象。

```
public void removeById(String schema,String simpleFeatureId) throws Exception{
    //这里展示是通过FilterFactory2来构造filter
    FilterFactory2 ff = CommonFactoryFinder.getFilterFactory2
    Filter filterToDelete = ff.id(ff.featureId(simpleFeatureId))
    SimpleFeatureStore fs = ds.getFeatureSource(schema);
    fs.removeFeatures(filterToDelete);
}
```

13.3.8. 与Spark集成分析

用户可以通过DLA Ganos实现Spark加载HBase中的时空数据并进行大规模时空分析操作。DLA Ganos是基于云原生数据湖分析（Data Lake Analytics，DLA）系统设计开发的，面向时空大数据存储与计算的数据引擎产品。基于DLA无服务器化（Serverless）数据湖分析服务与内置的Spark计算引擎，DLA Ganos提供了不同级别的数据分析模型，最基础的是GeometryRDD模型，用来实现Ganos数据中SimpleFeature与Spark中RDD模型的之间的转换。在GeometryRDD基础上，DLA Ganos基于SparkSQL设计了一系列用于空间数据表达的UDT与UDF或UDAF，允许用户使用类似SQL结构化查询语言进行数据的查询与分析。
详细内容请参考： 数据湖时空引擎Ganos

13.3.9. 与GeoServer集成

GeoServer可以发布Lindorm Ganos中数据源的矢量数据，操作步骤如下：

1、下载Lindorm Ganos的GeoServer插件，点击下载；

2、将该插件放到GeoServer的webapps/geoserver/WEB-INF/lib目录下；

3、启动GeoServer；

4、新增矢量数据源时，可以选择Lindorm(Ganos)；



13.4. 时空栅格

什么是时空栅格

栅格数据（Raster Data）是将地理空间分割成有规律的网格，每一个网格称为一个单元（像元或像素），并在各单元上赋予相应的属性值来表示实体的一种数据形式。
栅格数据通常有两种类型的栅格数据：专题数据和影像数据。

- 专题数据：每个栅格像元的值可以是一个测量值或分类值，如污染物浓度、降雨量、土地的权属类型或植被类型等。
- 影像数据：又称遥感影像或遥感图像（Remote Sensing Image），是通过地面遥感、航空遥感或航天遥感平台拍摄的，记录各种地物电磁波大小的胶片或照片，包含如航空影像和遥感卫星影像等。

每一幅栅格数据均带有时间属性和空间属性，我们称之为**时空栅格**。时空栅格还强调了栅格数据的时态特性，比如管理时间序列的系列栅格数据。

Lindorm + DLA Ganos的时空栅格方案

Lindorm支持时空栅格的数据存储和检索，除此之外，对时空栅格数据的处理还包括大量时空栅格计算等复杂场景，Lindorm与DLA Ganos结合可提供一套完成的解决方案。

请参考：DLA Ganos Raster

13.5. 时空轨迹

13.5.1. 功能简介

移动对象：在移动过程中产生带有时间戳的相关信息的对象。例如，飞机运行中，某一飞机（对象）在时间点 t 上有状态（飞行中、停靠等）、位置（经纬度）、高度以及速度等信息。

时空轨迹：移动对象数据库中记录对象连续位置变化信息的数据模型（详见轨迹存储模型基本概念），移动对象数据库为时空轨迹数据提供基本操作、查询和分析功能支持。

针对移动对象时空轨迹（以下简称轨迹）数据，Lindorm Ganos提供以下核心功能

低成本轨迹存储

- 轨迹点批量写入：支持带时间戳和自定义属性的二维轨迹点
- 轨迹存储模型：可采用点模型和线模型两类存储模型，其中线模型结合二级索引，可在提高查询效率的同时，相较点模型显著降低存储成本
- 轨迹分段存储：提供按点数/距离/经纬度/时间四种内置分段策略，从批量轨迹点构建分段线模型轨迹

高效率时空查询

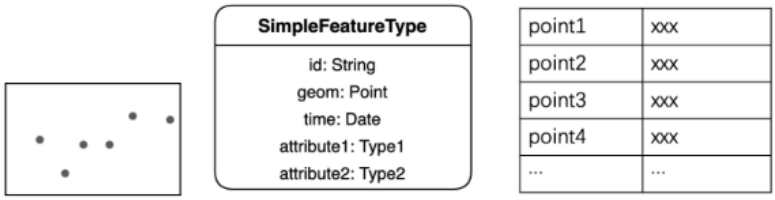
- 基本查询功能：ID+时间查询、属性查询、空间范围查询和时空范围查询
- 高效时空查询能力：通过对点模型构建z2/z3索引，或对线模型构建xz2/xz2t索引，实现高效率时空查询
- 多维度查询（点模型）：支持高效率的时空+属性查询，例如以时空索引作为主表rowkey，对其它属性构建二级索引。但二级索引构建不宜超过两个。

13.5.2. 基本概念

轨迹存储模型

移动对象采集的时空数据点在数据库中存储时，基本采用以下三种模型

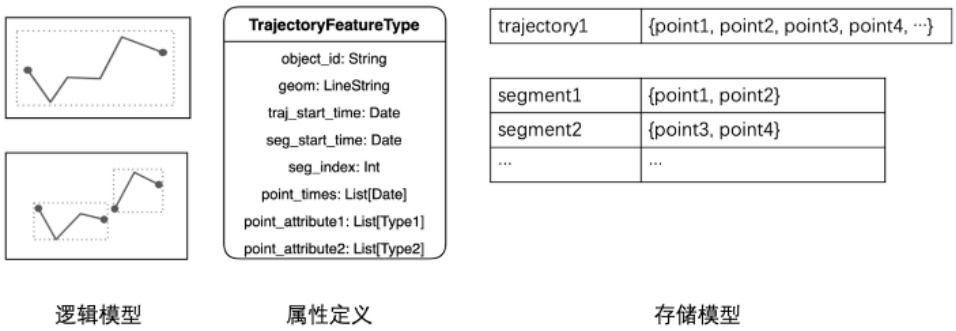
- 点模型：定义点模型各属性字段，每个点为一条记录。



点模型

- 轨迹线模型：整条轨迹为一条记录。

- 轨迹分段线模型：整条轨迹切分成多段，每个轨迹段为一条记录。



轨迹线/分段线模型

其中，seg_start_time为该轨迹段的起始时间，如果轨迹不分段则与traj_start_time相同，seg_index为该轨迹段的在整条轨迹中的编号，如果轨迹不分段，该字段为-1。

三种模型的优缺点和适用场景：

- 点模型利用索引进行实时查询的效率最高，但存储成本和二级索引代价高，适用于实时性要求较高的简单查询场景。
- 轨迹线模型时空索引效率不高，存储成本和二级索引代价低，适用于离线分析型场景。
- 轨迹分段线模型时空索引效率高于线模型，且存储成本和二级索引代价低，可同时实现低成本存储、实时和复杂分析查询，适用于对实时查询和离线分析都有需求的场景。

Lindorm Ganos同时支持三种模型的存储，批量轨迹点可以直接以点模型写入，也可以构建成线或分段线模型存储，用户可根据适用场景选择适合的模型。对于分段模型，Lindorm Ganos客户端提供基于距离差、角度差、时间间隔和点数的4种分段策略。线/分段线模型目前不支持点的追加和更新。

轨迹索引

轨迹模型引入了新的XZ2T和OTS索引（一般时空索引详见创建索引表），可提高轨迹查询效率。

- XZ2T索引：即time bin + XZ2，适用于轨迹线模型和分段线模型（非点类型的时空对象），相比XZ3索引查询效率更高。
- OTS索引：即移动对象object id + 轨迹起始时间 + 轨迹段标号，专用于轨迹分段线模型需要返回完整轨迹的查询情况。

轨迹二级索引存储

Lindorm Ganos支持对任意时空几何Schema创建二级索引，轨迹存储模型针对轨迹引入新的索引类型，采用Lindorm Ganos的时空二级索引机制（详见创建索引表），实现轨迹模型的二级索引支持。二级索引可以在保证查询效率的基础上有效降低存储成本，但要注意二级索引创建不宜超过两个。

13.5.3. 使用指南

本文介绍Ganos时空引擎功能的时空轨迹使用示例。

Java API（兼容GeoMesa的时空专用接口）

以轨迹分段线模型为例。

建表

1. 定义轨迹点模型SimpleFeatureType。

```
String pointSpec = "id:String,dtg:Date,*geom:Point:srid=4326";
SimpleFeatureType pointSFT = SimpleFeatureTypes.createType("point", pointSpec);
```

由于每个轨迹点也是具有属性的时空对象，在定义分段模型前，需要先定义点模型。

2. 创建轨迹线模型TrajectoryFeatureType。

```
TrajectoryBuilder trajBuilder = new TrajectoryBuilder("trajectory", pointSFT, "dtg", "id", true);
TrajectoryFeatureType trajFeatureType = trajBuilder.getTrajFeatureType();
```

使用Lindorm Ganos提供的帮助类TrajectoryBuilder，基于点模型创建轨迹线模型。创建TrajectoryBuilder时，需要指明pointSFT的时间和ID字段。

TrajectoryFeatureType有默认属性字段，参见基本概念（轨迹存储模型-线/分段线模型各属性字段定义）。默认的基本字段必须包括且不能修改，可以使用TrajectoryBuilder的addTrajProperties方法，增加新的轨迹属性字段。

```
TrajectoryBuilder trajBuilder = new TrajectoryBuilder("trajectory", pointSFT, "dtg", "id", true);
trajBuilder.addTrajProperties("speed:Double,traj_end_time:Date");
TrajectoryFeatureType trajFeatureType = trajBuilder.getTrajFeatureType();
```

3. 指定创建二级索引的必要信息。

```
trajFeatureType.getUserData().put("geomesa.secondary.enabled","true");
trajFeatureType.getUserData().put("geomesa.primary.index","id");
trajFeatureType.getUserData().put("geomesa.indices.enabled","id,xz2t");
```

通过设置TrajectoryFeatureType的userData指定创建二级索引的必要信息。本例指定ID为主表索引，创建针对时空属性的xz2t二级索引。

4. 创建表。

```
ds.createSchema(trajFeatureType);
```

调用Lindorm Ganos兼容的Geotools API：DataStore.createSchema创建表（DataStore获取方法可参见代码示例）。此时主表和二级索引表都已创建，二级索引表由Lindorm封装维护，对用户不可见。

写入轨迹数据

1. 创建轨迹点。

```
public SimpleFeature createFeature(SimpleFeatureType sft, String id, Object... values){
  SimpleFeatureBuilder sfBuilder = new SimpleFeatureBuilder(sft);
  SimpleFeature simpleFeature = sfBuilder.buildFeature(id,values);
  return simpleFeature;
}

List<SimpleFeature> points = new ArrayList<>();
for(String pointData: rawData) {
    // parse id and point values from pointData
    String pointID = ...;
    Object[] values = ...;

    points.add(createFeature(pointSFT, pointID, values);
}
```

根据前面的点模型定义，从原始数据解析轨迹点信息，创建轨迹点对象。

2. 定义轨迹分段策略。

```
SplitConfigs splitConfigs = new SplitConfigs(SplitConfigs.BY_COUNT(), new Object[]{50});
```

本例采用每段50个点划分一段的分段策略。

3. 创建轨迹段。

```
String objectID = "001";
List<SimpleFeature> trajFeatures = trajBuilder.createTrajSegments(objectID, features, splitConfigs);
```

指定objectID，使用TrajectoryBuilder.createTrajSegments方法，得到基于轨迹点List切分后的轨迹段。
注意如果前面使用addTrajProperties增加了轨迹段属性，需要在createTrajSegments时传入增加的轨迹段属性值。

```
Object[] extraTrajProperties = new Object[]{speed, trajEndTime};
List<SimpleFeature> trajFeatures = trajBuilder.createTrajSegments(objectID, features, splitConfigs, extraTra
jProperties);
```

4. 批量写入轨迹段。

```
SimpleFeatureStore featureStore = (SimpleFeatureStore) ds.getFeatureSource(trajFeatureType.getTypeName());
featureStore.addFeatures(new ListFeatureCollection(trajFeatureType, trajFeatures));
```

调用Lindorm Ganos兼容的GeoTools API：SimpleFeatureStore.addFeatures 批量写入轨迹段。

查询

Lindorm Ganos兼容GeoMesa支持的CQL语法，可参考查询时空对象。轨迹模型目前支持的查询有：

查询类型	示例
ID+时间查询	`object_id in ('001') and seg_start_time > 1549522511`
属性查询	`speed > 2.0` （点模型则查询点属性，线模型则只能查询整条轨迹/轨迹段的属性）
空间范围查询	`INTERSECTS (geom, POLYGON ( (-78 26,-73 26,-73 31,-78 31,-78 26) ) )`
时空范围查询	`INTERSECTS (geom, POLYGON ((-78 26,-73 26,-73 31,-78 31,-78 26))) AND traj_start_time >= 1549510271 AND traj_start_time <= 154958204`

查询步骤示例。

```
//创建查询语句
String ecqlPredicate = "INTERSECTS(geom, POLYGON ((-78 26,-73 26,-73 31,-78 31,-78 26))) AND traj_start_time >= 1549510271 AND traj_start_time <= 1549582040";

// 构建Query
String schemaName = "trajectory"; // named when creating table
Query query = new Query("trajectory", ECQL.toFilter(ecqlPredicate));

//获取查询结果
if(!getCompleteTrajectory) { // 如果只需要返回满足条件的轨迹段
    SimpleFeatureCollection featureCollection = ((LindormDataStore)ds).getFeatureSource("trajectory").getFeatures(query);
    SimpleFeatureIterator iterator = result.features();
    while (iterator.hasNext()) {
      iterator.next();
    }
} else { // 如果需要获取完整轨迹
    List<TrajectoryFeature> trajectories = ((LindormDataStore)ds).getFeatureSource("trajectory").getWholeTrajectoies(query);
}
```

13.5.4. 代码示例

本文介绍时空轨迹功能的使用示例。

1、示例下载

lindorm-ganos-mod-sample

下载解压后进入lindorm-sample目录。

```
mvn install -Dmaven.test.skip=true -s ../settings.xml
```

编译成功后，在target目录下生成 lindorm-mod-sample-1.0.0.jar。

2、使用示例

进入lindorm-sample目录。

```
java -jar target/lindorm-mod-sample-1.0.0.jar src/main/resources/template/mod_template.json
```

通过JSON文件指定操作和参数，JSON文件可参照示例mod_template.json。

2.1 写入数据

在mod_template.json中，设置写入参数。该示例指定AIS轨迹点数据集目录，读取数据并以分段二级索引模型存储轨迹，以ID为主索引，xz2t为二级索引创建表 "traj"并写入数据。分段策略为每50个点划分一段（目前支持BY_DURATION、BY_DISTANCE、BY_ANGLE、BY_COUNT四种分段策略，可以通过修改MOD.java中的SplitConfigs设置，使用其它分段策略）。

```
{
  "operation" : "WRITE",
  "zk_address" : "your_zk_address",
  "catalog" : "example",
  "schema" : "traj",
  "enable_secondary_index" : "true",
  "print_info": "false",
  "enabled_indices" : "id,xz2t",
  "primary_index" : "id",
  "thread_num": 8,
  "input_dir": "path_to_repository/lindorm-sample/src/main/resources/data/traj",
}
```

"enabled_indices" 指定需要创建的索引（索引类型参见创建索引表），注意需要设置"enable_secondary_index" 为true才能使用二级索引，并设置"primary_index"指定其中的主索引。"thread_num"设置写入数据时的并行线程数，默认为1。
然后执行sample写入。

```
java -jar target/lindorm-mod-sample-1.0.0.jar src/main/resources/template/mod_template.json
```

2.2 查询数据
在mod_template.json中设置查询参数，使用示例代码默认的查询范围。

```
{
  "operation" : "QUERY",
  "zk_address" : "your_zk_address",
  "catalog" : "example",
  "schema" : "traj",
  "predicate" : "INTERSECTS(geom, POLYGON ((-76 28, -74 28, -74 30, -76 30, -76 28))) and traj_start_time >=
1549510271 AND traj_start_time <= 1549583941",
  "return_complete_trajectory" : "false",
  "print_results" : "true"
}
```

"predicate"设置查询语句，"return_complete_trajectory" 指定是否返回完整轨迹（分段存储的轨迹在查询时默认返回符合查询条件的轨迹段，需设定该参数为true才会返回完整轨迹）。
执行sample查询。

```
java -jar target/lindorm-mod-sample-1.0.0.jar src/main/resources/template/mod_template.json
```

可以看到返回显示的查询结果。

14.数据导入导出

14.1. Lindorm数据生态大图

本文介绍Lindorm与关系型数据库、NoSQL数据库、大数据生态之间的数据导入导出能力。其中部分能力由LTS支持（Lindorm自研的数据通道服务）、部分能力由开源的Datax支持

数据导入

	源集群	目标集群	全量导入	增量导入
关系数据库	MySQL	Lindorm 宽表	支持 by LTS	支持 by LTS
	PolarDB	Lindorm 宽表	支持 by LTS	支持 by LTS
	PostgreSQL	Lindorm 宽表	支持 by Datax	NO
	Oracle	Lindorm 宽表	支持 by Datax	NO
	SQLServer	Lindorm 宽表	支持 by Datax	NO
NoSQL	HBase 1.x	Lindorm 宽表	支持 by LTS	支持 by LTS
	HBase 2.x	Lindorm 宽表	支持 by LTS	支持 by LTS
	云HBase单机版	Lindorm 宽表	支持 by LTS	NO
	云HBase增强版	Lindorm 宽表	支持 by LTS	支持 by LTS
	Cassandra	Lindorm 宽表	支持 by Datax	NO
	MongoDB	Lindorm 宽表	支持 by Datax	NO
大数据	MaxCompute	Lindorm 宽表	支持 by LTS	N/A
	Hive	Lindorm 宽表	即将支持	N/A
	Kafka	Lindorm 宽表	N/A	即将支持
文件	CSV	Lindorm 宽表	支持 by Datax	N/A
	ORC	Lindorm 宽表	即将支持	N/A
	Parquet	Lindorm 宽表	即将支持	N/A
	JSON	Lindorm 宽表	即将支持	N/A

数据导出

	源集群	目标集群	全量导出	增量导出
关系数据库	Lindorm 宽表	MySQL	支持 by Datax	NO
	Lindorm 宽表	PolarDB	NO	NO
	Lindorm 宽表	PostgreSQL	支持 by Datax	NO
	Lindorm 宽表	Oracle	支持 by Datax	NO
	Lindorm 宽表	SQLServer	支持 by Datax	NO

	源集群	目标集群	全量导出	增量导出
NoSQL	Lindorm 宽表	HBase 1.x	支持 by LTS	支持 by LTS
	Lindorm 宽表	HBase 2.x	支持 by LTS	支持 by LTS
	Lindorm 宽表	云HBase单机版	NO	NO
	Lindorm 宽表	云HBase增强版	支持 by LTS	支持 by LTS
	Lindorm 宽表	Cassandra	支持 by Datax	NO
	Lindorm 宽表	MongoDB	支持 by Datax	NO
大数据	Lindorm 宽表	MaxCompute	支持 by LTS	支持 by LTS
	Lindorm 宽表	DLA 数据湖	支持 by LTS	支持 by LTS
	Lindorm 宽表	HDFS	支持 by Datax	N/A
	Lindorm 宽表	OSS	支持 by Datax	N/A
	Lindorm 宽表	Kafka	N/A	支持 by LTS
文件	Lindorm 宽表	CSV	支持 by Datax	N/A
	Lindorm 宽表	ORC	NO	N/A
	Lindorm 宽表	Parquet	NO	N/A
	Lindorm 宽表	JSON	NO	N/A

14.2. LTS（原BDS）服务介绍

LTS（Lindorm Tunnel Service）是面向Lindorm业务场景特点深度定制的数据生态服务。支持简单易用的数据交换、处理、订阅等能力，满足用户的数据迁移、实时订阅、数湖转存、数仓回流、单元化多活、备份恢复等需求，实现面向Lindorm的一站式数据生态服务。

核心能力

- **云原生分布式**：LTS是一个基于ECS部署的分布式系统，具备良好的水平扩展能力，用户可以按需进行资源配置。
- **简单易用**：LTS支持一键配置数据迁移、导入、订阅、归档等任务。比如数据迁移任务，用户只需要选择源端、目的端和需要同步的列，LTS会自动完成schema复制、全量数据复制和增量数据复制。
- **安全可靠**：LTS充分考虑对源端和目标端系统在线影响最小化，以及兼容可能出现的故障。在任务启动前会有网络联通性、安全性等前置校验，运行过程中会实时监控同步延迟、目标集群存储水位等并有相关的限速和报警，任务结束后还会提供数据校验。
- **低成本**：LTS在HBase、Phoenix、Cassandra等开源系统上有深入的优化，提供物理文件级别的数据处理，比传统的数据复制效率提高10倍。同时在CPU、缓存、内存、网络IO上的整体优化使LTS提供一个高性价比的数据通道，减少客户数据流转、处理的成本。

功能列表

功能	场景	相关文档
HBase、Lindorm宽表之间数据迁移	新旧集群无缝迁移、集群升配、在线离线业务分离、主备容灾、异地多活。	全增量数据同步
RDS -> Lindorm	在线离线业务分离、历史库。	RDS全增量同步
MaxCompute/Hive -> Lindorm	离线数据查询加速，将数仓中的明细、指标等数据批量回流Lindorm提供在线查询。	待补充，目前可联系值班咨询

功能	场景	相关文档
Lindorm 数据导出 ODPS（MaxCompute）	支持历史数据和增量数据。	全量导出MaxCompute、增量归档MaxCompute
LogHub实时数据订阅	支持订阅LogHub实时数据到Lindorm。	日志服务（SLS）增量导入
Lindorm 数据订阅	支持Lindorm实时增量数据订阅。	Lindorm Stream

日志生命周期管理说明

- 开启日志订阅后，如果不消费数据，默认日志会保留48小时，超时后订阅关系自动取消，保留的数据自动删除。
- 什么场景会导致“不消费数据”：没有终止任务的前提下直接释放LTS集群，同步任务暂停。
- 使用日志订阅的任务类型：HBase/Lindorm增量同步、数据归档、数据备份和数据订阅。

典型场景

- 集群迁移
  - 使用范围
    - HBase迁移Lindorm。
    - 集群网络的切换，经典切换VPC。
    - 异地跨机房迁移。
    - 业务拆分。
  - 特点
    - 不停服数据迁移，同时搞定历史数据迁移、实时增量数据的同步。
    - 迁移过程中不会和源集群的HBase、Lindorm交互，只读取源集群的HDFS，尽可能减少对源集群在线业务的影响。
    - 文件层的数据拷贝比通常API层的数据迁移通常能节省50%以上的流量。
    - 高效性，单个节点迁移速度可达到100 MB/s，节点数支持水平扩展，能够满足支持TB、PB级别的数据迁移。
    - 稳定性，有完善的错误重试机制，实时监控任务速度和进度，支持任务失败报警。
    - 正确性，能够进行数据校验。
    - 支持自动同步schema，保证分区一致。
- 在线离线业务分离
  通过LTS，将在线业务数据实时同步到HDFS或者OSS等存储，结合Spark、MR等大数据组件进行数据分析，从而不影响在线业务的查询。
- 主备容灾
  通过LTS在主备集群之间实现实时数据的双向同步，当主集群出现问题，可以切换到备集群，尽可能减少对业务产生的影响，当主集群服务恢复之后，通过LTS同步服务补齐主集群遗漏的数据。
- RDS历史库建设
  对于交易订单等存储场景，随着数据量增大会对RDS造成性能压力，周期性的归档数据或者分库分表存在成本高、操作复杂等痛点。可以通过LTS将RDS数据实时同步到Lindorm宽表实现冷热数据分离，Lindorm宽表提供自动水平扩展、高并发查询、多维索引、轻量分析，Lindorm Streams提供变更数据有序订阅，LTS还支持将Lindorm 宽表数据同步到其它分析系统进行复杂数据分析。

14.3. 开通并登录LTS

LTS（Lindorm Tunnel Service）是云原生多模数据库Lindorm自研的数据通道服务，通过LTS完成数据导入导出、数据订阅、备份恢复等功能。本文介绍开通并登录LTS的操作步骤。

前提条件

- 已创建云原生多模数据库Lindorm实例，具体操作请参见创建实例。
- 已将客户端IP地址添加至云原生多模数据库Lindorm实例的白名单，具体操作请参见设置白名单。

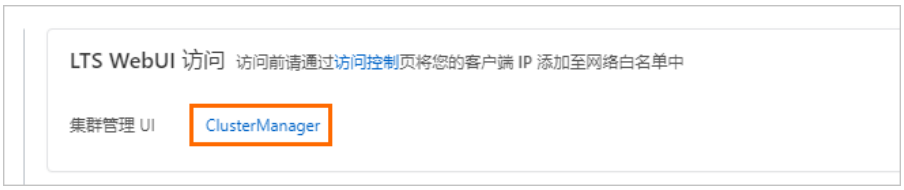
开通LTS

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。

3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**宽表引擎 > 数据订阅**。
5. 单击**这里**开通LTS引擎。
6. 在**变配**页面，配置以下参数并勾选服务协议。
   - BDS数据同步Core规格：选择LTS的Core规格，具体方法请参见Core选择。
   - BDS节点数量：选择LTS的节点数量。
7. 配置完成后，单击**立即购买**。

登录LTS

1. 登录Lindorm管理控制台。
2. 在页面左上角，选择实例所属的地域。
3. 在**实例列表**页，单击目标实例ID或者目标实例所在行**操作**列的**管理**。
4. 在左侧导航栏，选择**宽表引擎 > 数据订阅**。
5. （可选）如果实例的数据订阅功能未开通，请单击**立即开通**。
6. 在**LTS WebUI访问**区域，单击**ClusterManager**进入至LTS操作页面。



> **说明** 首次登录LTS服务时，单击页面右侧的**重置UI访问密码**，设置访问LTS服务的用户名和密码，规则如下：
> - 长度为2~30。
> - 包含大写字母、小写字母、特殊字符和数字中的一种或者多种。
> - 允许的特殊字符包括下划线（_）和短划线（-）。

14.4. 数据源管理

14.4.1. 添加DTS数据源

本文介绍如何在云原生多模数据库Lindorm控制台上添加DTS数据源。

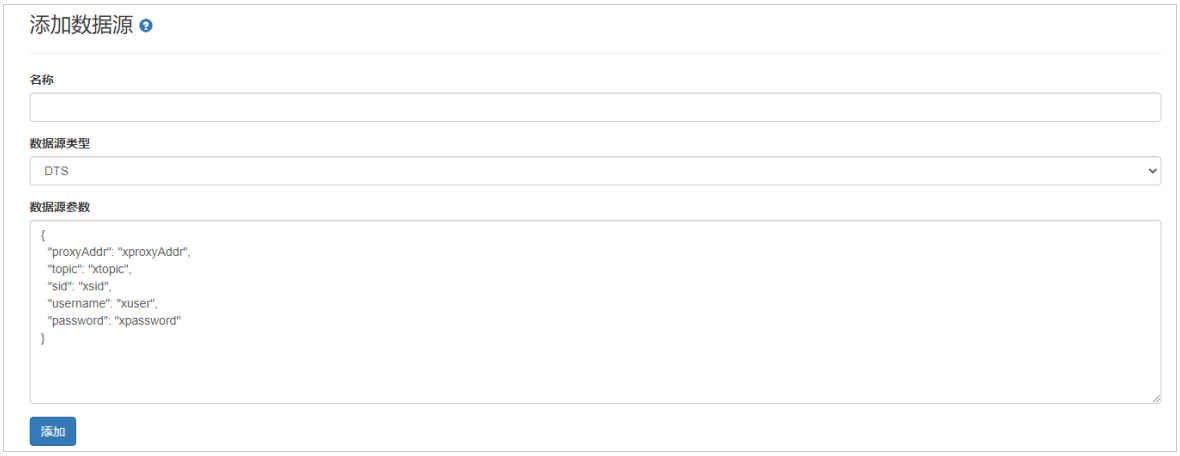
前提条件

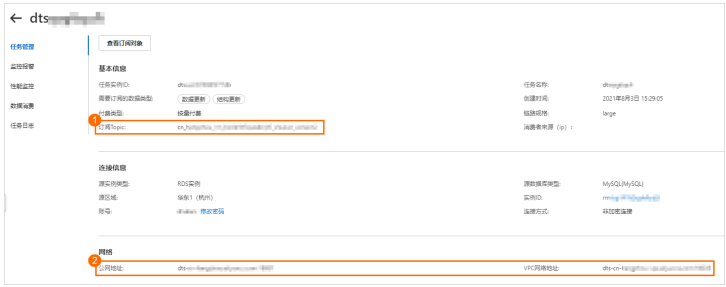
云原生多模数据库Lindorm实例已开通LTS服务，具体操作请参见开通LTS服务。

操作步骤

1. 已登录LTS服务，具体操作请参见登录LTS服务。
2. 在左侧导航栏，选择**数据源管理>添加数据源**。

3. 在**添加数据源**页面，配置以下参数。



参数	说明
名称	自定义DTS数据源的名称。
数据源类型	在下拉菜单中选择**DTS**。
数据源参数	通过数据传输服务DTS控制台获取数据源参数。 ○ 单击目标DTS任务在**任务管理**页面获取以下参数：  ■ topic：DTSt订阅Topic名称（图示中①）。 ■ proxyAddr：DTS订阅地址（图示中②）。 ○ 单击目标DTS任务在**数据消费**页面获取以下参数：  ■ sid：消费组ID（图示中③）。 ■ username：消费组的账号（图示中④）。 ■ password：消费组账号对应的密码，如果您忘记密码，可以单击**操作**列的**修改密码**重新设置密码。

4. 单击**添加**。

14.4.2. 添加HBase数据源

添加HBase数据源可以对HBase数据进行迁移或者同步操作，本文介绍如何添加HBase数据源。

HBase数据源类型和添加方式

HBase数据源类型	添加HBase数据源的方式

HBase数据源类型	添加HBase数据源的方式

云数据库HBase标准版	• 如果云数据库HBase标准版或者云数据库HBase增强版与LTS使用相同的专有网络ID，添加方法请参见通过云原生多模数据库Lindorm控制台添加HBase数据源。
	• 如果云数据库HBase标准版或者云数据库HBase增强版与LTS使用不同的专有网络ID，添加方式如下：
	  ○ 云数据库HBase标准版数据源的添加方式清参见通过LTS服务添加HBase数据源。
	  ○ 云数据库HBase增强版数据源的添加方式清参见添加Lindorm宽表数据源（兼容HBase）。

云数据库HBase增强版	
自建的HBase集群	请通过LTS服务添加HBase数据源。
EMR HBase集群	

通过云原生多模数据库Lindorm控制台添加HBase数据源

1. 登录Lindorm管理控制台。
2. 在**实例列表**页，单击引擎类型为**LTS**的实例ID。
3. 在左侧导航栏选择**数据源管理**。
4. 在**Hbase数据源**页面，单击**添加数据源**。
5. 在**添加数据源**对话框中配置以下信息。

配置项	说明
实例类型	选择**HBase增强版**或者**HBase标准版**。
实例地区	选择目标实例所属地域。
实例ID	选择目标实例ID。 **说明** 确保选择的云数据库HBase与LTS使用相同的专有网络ID。

6. 单击**确定**，状态为**已关联**表示HBase数据源已添加成功。

通过LTS服务添加HBase数据源

1. 开通并登录LTS服务，具体操作请参见开通并登录LTS。
2. 在左侧导航栏选择**数据源管理 > 添加数据源**。
3. 在**添加数据源**页面配置以下参数。

参数	说明
名称	自定义数据源名称。
数据源类型	根据HBase数据源类型的版本选择不同的数据源类型，包括以下选择： ○ **HBase1x**：表示HBase数据源版本为1.0版本。 ○ **HBase2x**：表示HBase数据源版本为2.0版本。 ○ **HBase098**：表示HBase数据源为自建的hbase098版本。 ○ **HBase094**：表示HBase数据源为自建的hbase094版本。

参数	说明
数据源参数	输入HBase数据源的相关参数。 `{ "clusterKey":"xxxzkhost1,xxxzkhost2,xxxzkhost3:2181:/xxxhbase", "hbaseDir":"/xxxhbase", "hdfsUri":"hdfs://xxxnn1:8020,xxxnn2:8020" }` ○ clusterKey：HBase数据源的连接信息，格式为：`ZK节点:ZK端口:HBase数据源在HDFS中的路径`。 ○ hbaseDir：HBase数据源在HDFS中的路径。 ○ hdfsUri：自建的HBase集群的HDFS连接地址。 ■ 如果HBase数据源为高可用实例，HDFS的连接地址格式为：`hdfs://namenode1:port,namenode2:port`。 ■ 如果HBase数据源为非高可用实例，HDFS的连接地址格式为：`hdfs://namenode:port`。
集群/etc/hosts(自建、EMR集群需要)	自建HBase集群或者EMR HBase集群需要配置。HBase所在集群的etc或者hosts。

4. 单击**添加**。

14.4.3. ODPS数据源

数据源添加步骤

1. 进入LTS操作页面，**数据源管理 -> ODPS数据源**。

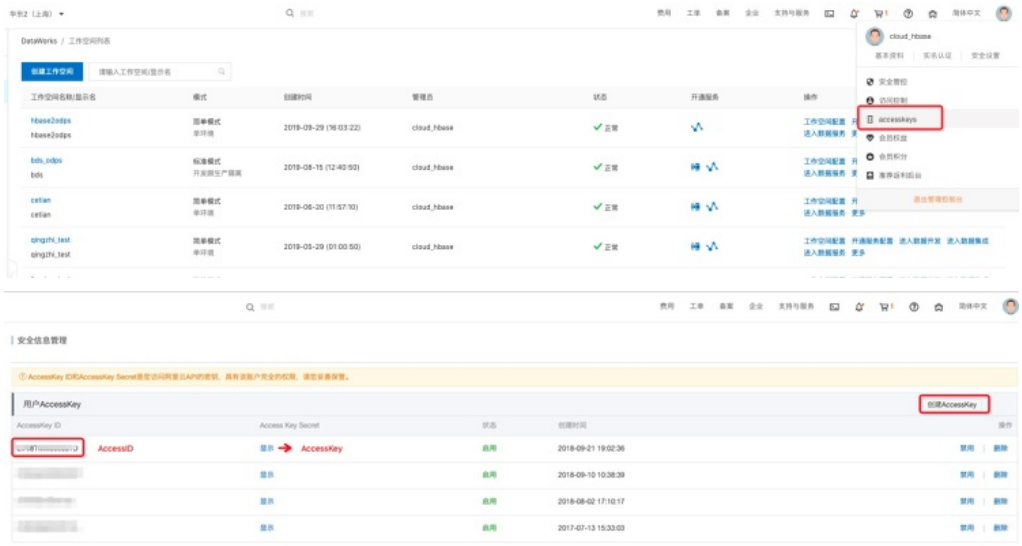


2. 录入ODPS的相关信息， 点击**创建**。



参数说明

- **获取 AccessId、AccessKey**



- **获取EndPoint 、TunnelEndPoint**
  根据ODPS所在地域， 查阅 配置EndPoint 获取VPC网络下的EndPoint地址。 测试案例中，ODPS在华东2，因此选择华东2的VPC网络下EndPoint地址。



- VPC网络下Region和服务连接对照表
  在VPC网络下访问MaxCompute，只能使用如下Endpoint和Tunnel Endpoint。

Region名称	所在城市	开服状态	VPC网络Endpoint	VPC网络Tunnel Endpoint
华东1	杭州	已开服	http://service.cn-hangzhou.maxcompute.aliyun-inc.com/api	http://dt.cn-hangzhou.maxcompute.aliyun-inc.com
华东2	上海	已开服	http://service.cn-shanghai.maxcompute.aliyun-inc.com/api	http://dt.cn-shanghai.maxcompute.aliyun-inc.com

MaxCompute已开服的Region和服务连接对照表
访问原则

- **获取ODPS Project**



14.4.4. RocketMQ数据源

数据源添加

1. 首先进入LTS操作页面， 数据源管理 -> RocketMQ数据源

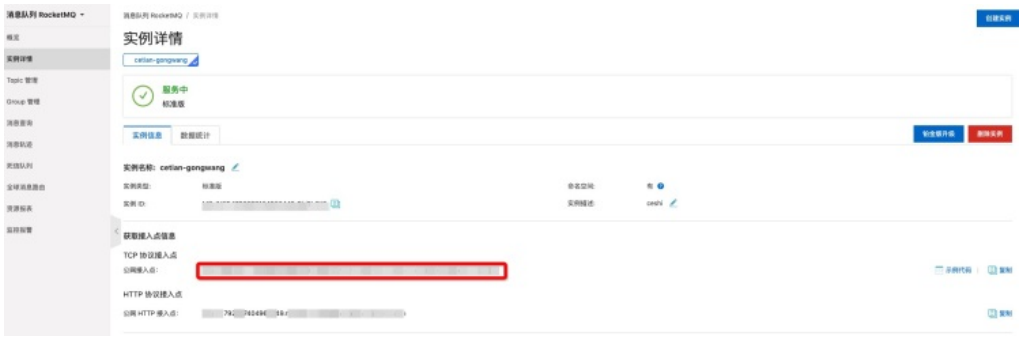


2. 录入MQ的相关信息， 点击创建，创建对应的MQ数据源。

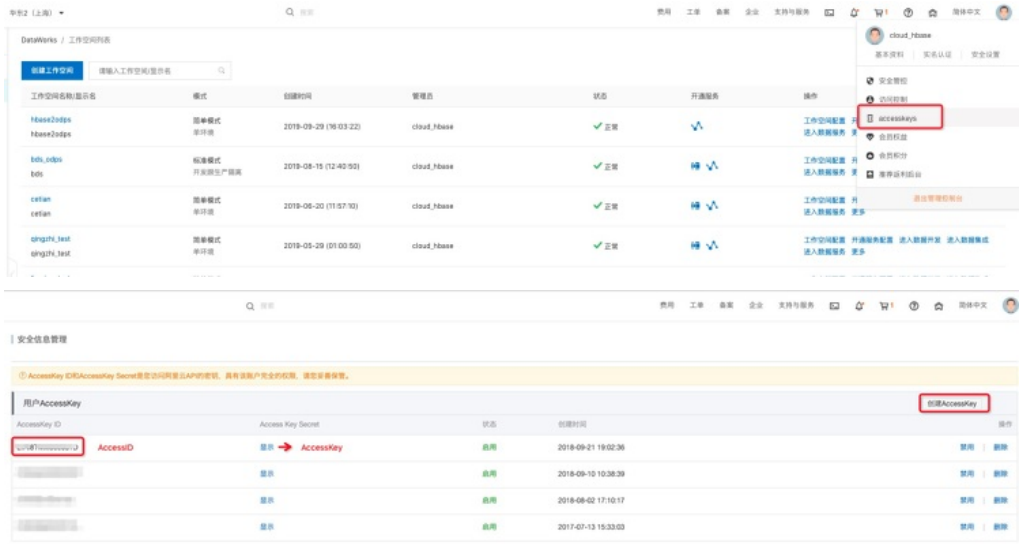


参数说明

- **获取NameSrvAddr**
  NameSrvAddr表示MQ的接入点，对于自建客户，请自行查询MQ的配置。对于云上的客户可以在MQ控制台获取。



- **获取 AccessId、AccessKey**



14.4.5. 添加Kafka数据源

本文介绍如何添加Kafka数据源。

前提条件

已登录LTS页面，具体操作请参见开通并登录LTS。

操作步骤

1. 进入LTS操作页面，在左侧导航栏选择**数据源管理 > 添加数据源**。
2. 在**添加数据源**页面设置相关参数。

参数	说明
名称	自定义名称，支持字母与数字。
数据源类型	选择**KAFKA**。
数据源参数	`{bootstrap.servers:"server1:port1,server2:port2"}` ○ `bootstrap.servers`：Kafka配置接入点，即控制台的实例详情页面显示的默认接入点，只支持相同VPC下默认接入点方式访问。 ○ `server1:port1`：填写格式为 `服务名:端口号`。

3. 单击**添加**。

14.4.6. 添加Lindorm宽表数据源

本文介绍如何添加Lindorm宽表数据源。

前提条件

已登录LTS页面，具体操作请参见开通并登录LTS。

添加Lindorm SQL数据源

1. 进入LTS操作页面，在左侧导航栏选择**数据源管理** > **添加数据源**。
2. 在**添加数据源**页面设置相关参数。

参数	说明
名称	自定义名称，支持字母与数字。
数据源类型	选择**LQL**。
数据源参数	`{ "clusterKey":"x.hbaseue.rds.aliyuncs.com:30020:$username:$password" }` ○ `x.hbaseue.rds.aliyuncs.com:30020`：填写Lindorm宽表引擎中**使用HBase Java API**专有网络访问的连接地址，获取方法请参见查看连接地址。 ○ `$username`：访问Lindorm宽表的用户名。 ○ `$password`：用户名对应的密码。

3. 单击**添加**。

添加Lindorm宽表数据源（兼容HBase）

1. 进入LTS操作页面，在左侧导航栏选择**数据源管理** > **添加数据源**。
2. 在**添加数据源**页面设置相关参数。

参数	说明
名称	自定义名称，支持字母与数字。
数据源类型	选择**Lindorm for HBase（增强版）**。
数据源参数	`{ "clusterKey":"{x.hbaseue.rds.aliyuncs.com:30020}:{username}:{password}" }` ○ `x.hbaseue.rds.aliyuncs.com:30020`：填写Lindorm宽表引擎中**使用HBase Java API**专有网络访问的连接地址，获取方法请参见查看连接地址。 ○ `username`：访问Lindorm宽表的用户名。 ○ `password`：用户名对应的密码。
集群/etc/hosts（自建、EMR集群需要）	自建HBase集群或者EMR HBase集群需要配置。HBase所在集群的etc或者hosts。

3. 单击**添加**。

添加Lindorm宽表数据源（兼容Cassandra）

1. 进入LTS操作页面，在左侧导航栏选择**数据源管理** > **添加数据源**。
2. 在**添加数据源**页面设置相关参数。

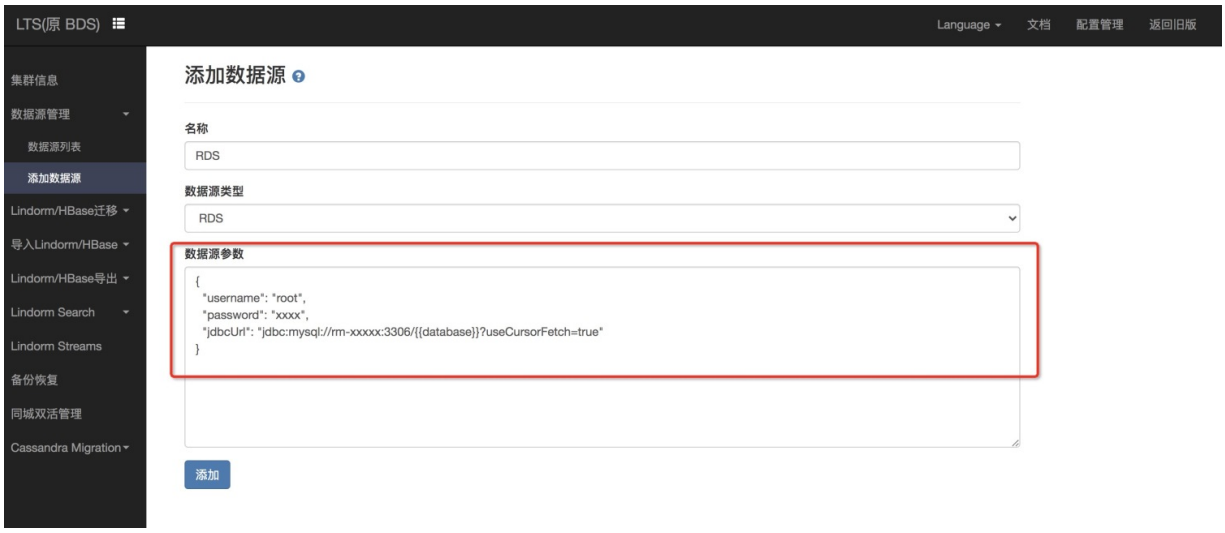
参数	说明
名称	自定义名称，支持字母与数字。
数据源类型	选择**Lindorm for Cassandra**。

数据源参数	`{`
	`"cqlAddress":"x.rds.aliyuncs.com:9042",`
	`"username":"",`
	`"password":"",`
	`"clusterKey":"x.hbaseue.rds.aliyuncs.com:30020:username:password"`
	`}`
	○ `x.rds.aliyuncs.com:9042`：填写Lindorm宽表引擎中**Cassandra兼容**专有网络访问的连接地址，获取方法请参见查看连接地址。
	○ `x.hbaseue.rds.aliyuncs.com:30020`：填写Lindorm宽表引擎中**使用HBase Java API**专有网络访问的连接地址，获取方法请参见查看连接地址。
	○ `username`：访问Lindorm宽表的用户名。
	○ `password`：用户名对应的密码。

3. 单击**添加**。

14.4.7. RDS数据源添加

添加RDS数据源



jdbcUrl：rm-xxxx替换成RDS实例域名,{{database}}替换成数据库名称

username：用户名

password：密码

14.4.8. 添加HDFS数据源

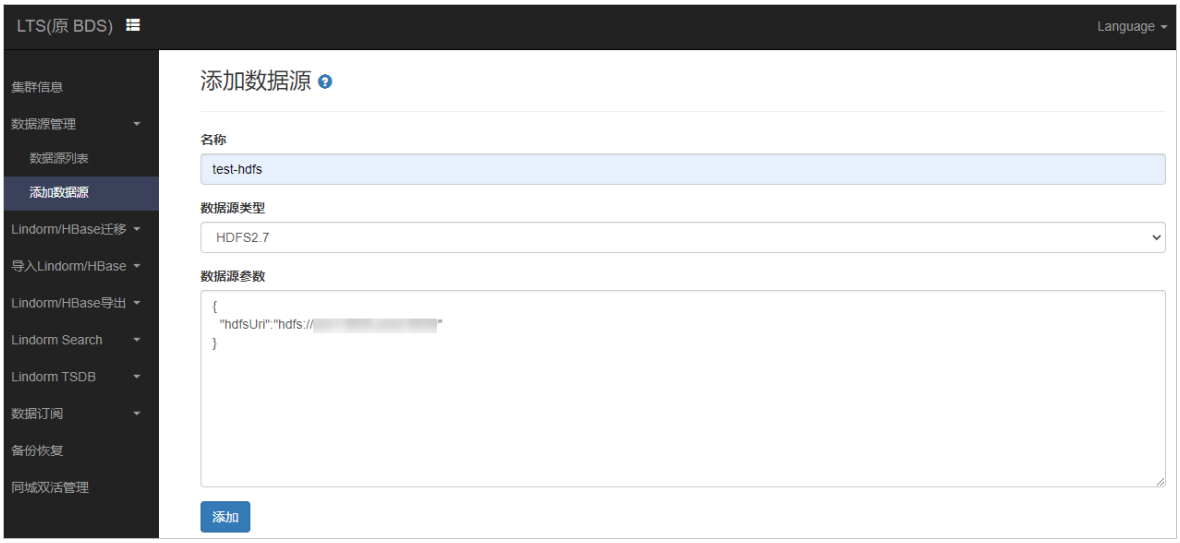
添加HDFS数据源用于实现云原生多模数据库Lindorm宽表引擎能够读取HDFS的数据，如果您要将HDFS数据导入到云原生多模数据库Lindorm宽表中，需要先添加HDFS数据源。本文介绍如何添加HDFS数据源。

前提条件

- 已购买LTS数据同步服务，具体操作请参见购买LTS。
- 已将客户端IP添加至白名单中，具体操作请参见设置白名单。

操作步骤

1. 访问LTS操作页面，具体操作请参见访问数据同步服务。
2. 在左侧导航栏中，选择**数据源管理 > 添加数据源**。



3. 在**添加数据源**页面，配置以下参数。
   - **名称**：输入数据源的名称。
   - **数据源类型**：固定选择**HDFS2.7**。
   - **数据源参数**：输入HDFS的namenode域名和端口，格式为 `hdfs://xnn1:port,xnn2:port` 。
4. 单击**添加**。

14.5. 实时数据订阅

14.5.1. 实时数据订阅概述

云原生多模数据库Lindorm支持实时数据订阅功能，实时数据订阅功能帮助您获取Lindorm的实时增量数据，您可以根据自身业务需求自由消费增量数据，搭建或实现多种业务场景。

背景信息

云原生多模数据库Lindorm数据订阅功能支持任何一个表的每一条数据变更，您可以在客户端实时有序的查看数据变更记录。当您开通某一张表的数据订阅功能后，其变更数据的操作就会被存储。为了确保数据消费的顺序和数据写入的顺序一致，数据订阅功能提供了主键级别保序，对于同一个主键的更新操作，会按照其更新的顺序存储和消费。每次对Lindorm表格的数据执行增删改操作时，数据订阅都会生成一个Stream Record键值对，键值对的键是这一行数据的主键，值是此次操作的详细信息（操作前的值，操作后的值，时间戳，操作类型）。

开通模式

开通云原生多模数据库Lindorm数据订阅功能有两种模式，Pull模式和Push模式。

- Pull模式：您可以使用开源的Kafka客户端直接订阅数据，服务端会暂时保存数据，您需要对保存的数据设置一个过期时间，如果超过这个时间数据会被删除。具体操作请参见通过Pull模式创建数据订阅通道。
- Push模式：云原生多模数据库Lindorm数据订阅功能直接将数据推送到Kafka客户端，服务端不会存储数据。具体操作请参见通过Push模式创建数据订阅。

使用限制

- 云原生多模数据库Lindorm数据订阅功能仅支持数据处理语义为至少一次（At-least-once）语义，即同一条数据可能因为内部重试而被重复消费。如果您需要设置为精确一次（Exactly Once）语义，请提交工单联系我们。
- 目前数据订阅的表格仅支持Lindorm SQL表格。如果您的表是HBase，请提交工单联系我们。
- 非SSD实例不建议开启数据订阅功能，由于数据订阅功能中导出整行数据的最新值或更新前的值时为了获取更新前的值会对Lindorm实例增加读操作，每一次写入操作都会增加一次读操作。如果您不需要OldImage/NewImage功能，仅需要变更Delta数据，请提交工单联系我们。
- 通过Pull模式创建数据订阅通道的订阅数据默认保存7天，与Lindorm共享底层的LDFS存储空间，因此您在开启数据订阅功能前需要先确认有充足的存储空间。

14.5.2. 通过Pull模式创建数据订阅通道

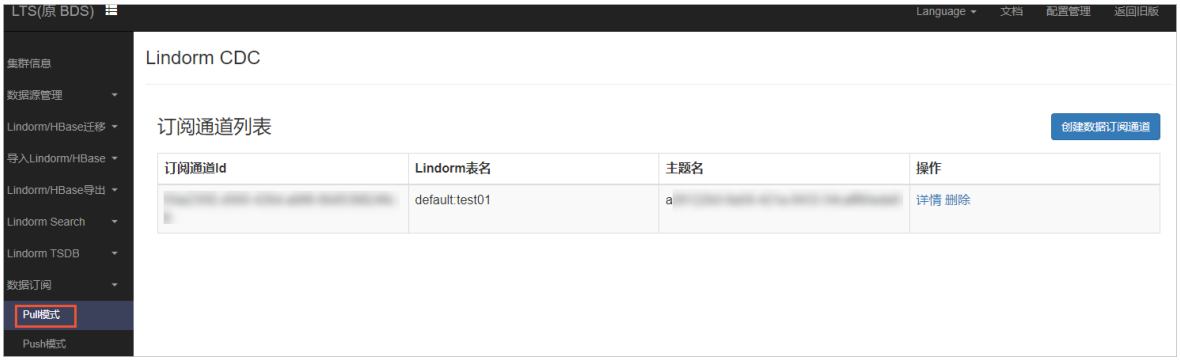
本文介绍通过Pull模式创建数据订阅功能，创建后订阅通道会实时拉取数据库实例的增量数据，并将增量数据保存在订阅通道中，您可以使用Lindorm提供的SDK从订阅通道中订阅增量数据并进行消费。同时，您可以在LTS页面进行订阅通道的创建、查看及删除等操作。

前提条件

已将客户端IP添加至白名单中，具体操作请参见设置白名单。

操作步骤

1. 进入LTS（原BDS）页面，在左侧导航栏中，选择**数据订阅 > Pull模式**。

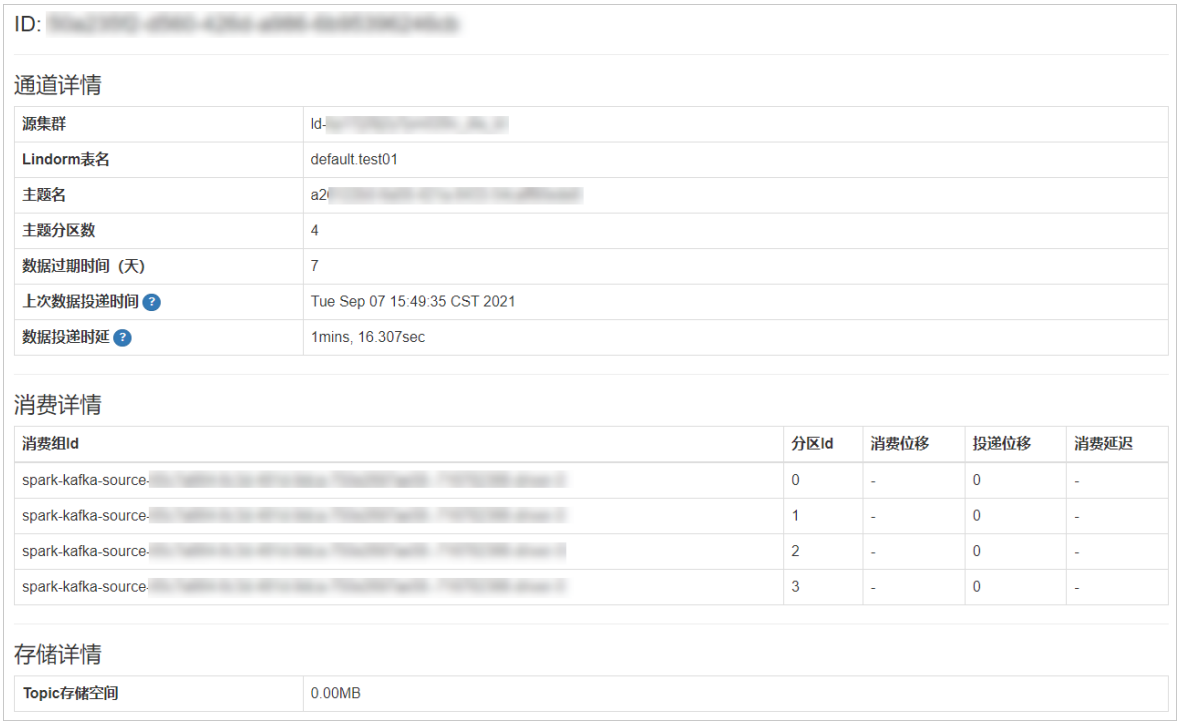


2. 单击**创建数据订阅通道**，并配置以下参数。



名称	描述
源集群	填写Lindorm实例ID。
Lindorm表名	选择需要创建数据订阅通道的Lindorm实例表，一条通道只能选择一张表格。
主题名	用于消费数据的主题名称。
数据过期时间（天）	表示数据可以保存的天数，默认为7天。
主题分区数	表示Kafka客户端为该主题设置多个分区，多分区可以并发消费数据，默认为4个分区。

3. 单击**提交**。
4. （可选）找到目标订阅通道，单击**操作**列的**详情**，可以查看数据订阅通道详情、消费详情和存储详情。


ID:

通道详情

源集群	Id-
Lindorm表名	default.test01
主题名	a2
主题分区数	4
数据过期时间（天）	7
上次数据投递时间	Tue Sep 07 15:49:35 CST 2021
数据投递时延	1mins, 16.307sec

消费详情

消费组Id	分区Id	消费位移	投递位移	消费延迟
spark-kafka-source-	0	-	0	-
spark-kafka-source-	1	-	0	-
spark-kafka-source-	2	-	0	-
spark-kafka-source-	3	-	0	-

存储详情

Topic存储空间	0.00MB

5. （可选）您可以通过以下代码在Kafka客户端对订阅数据进行消费。

```
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.kafka.clients.admin.AdminClientConfig;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.ByteArrayDeserializer;

import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

public class TestConsume {
  public static void main(String[] args) throws Exception {
    // 创建订阅通道时填写的topic名称
    String topic = "test-topic";

    // 链接endpoint的配置项
    Properties props = new Properties();
    // 指定链接endpoint地址
    props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "ld-xxx:9092");
    // 指定Key序列化器，不可更改
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, ByteArrayDeserializer.class.getName());
    // 指定Value序列化器，不可更改
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, ByteArrayDeserializer.class.getName());
    // 指定消费组名称，在消费时会自动创建
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "group-id-0");

    // 创建消费者
    KafkaConsumer<byte[], byte[]> consumer = new KafkaConsumer<>(props);
    // 订阅主题
    consumer.subscribe(Arrays.asList(topic));

    // 用消费者拉取数据
    ConsumerRecords<byte[], byte[]> records = consumer.poll(Duration.ofMillis(10000));
    for (ConsumerRecord<byte[], byte[]> record : records) {
      // 查看数据内容
      System.out.println("key: " + Bytes.toString(record.key()));
      System.out.println("value: " + Bytes.toString(record.value()));
    }
    // 提交当前消费位移
    consumer.commitSync();
    // 关闭消费者
    consumer.close();
  }
}
```

> **说明**
> 数据消费格式说明请参见数据消费格式。

14.5.3. 使用Flink SQL消费实时变更数据示例

云原生多模数据库Lindorm支持实时数据订阅功能，对实时变更数据进行计算，可以为实时监控，实时报表和流数据分析功能提供有效数据。通过Flink Kafka Connector可以从底层存储消费订阅数据，同时Flink Kafka Connector对订阅数据进行解析时支持Debezium格式。

背景信息

云原生多模数据库Lindorm的实时数据订阅功能包括Push模式和Pull模式。Push模式支持将实时变更数据投递至下游系统（例如TT、DD或MetaQ），同时可以结合其他计算平台对增量数据进行计算，构建一套完整的实时数据系统，但是这种模式存在日志堆积、开发周期长等问题。Pull模式的实时数据订阅功能底层使用Lindorm Streams Storage存储订阅消息，读写方式兼容Kafka客户端，因此可以支持Flink通过Kafka Connector的方式订阅消费。

前提条件

已创建数据订阅通道，具体操作，请参见通过Pull模式创建数据订阅通道。

Table API接入方式

Table API是Flink提供的API接口，支持使用Flink SQL进行数据查询。

- Lindorm CDC的变更消息采用Debezium Format格式。
- Table API支持Debezium-Json格式，相关内容请参见解析说明。

如果您在开源Flink上消费数据，需要配置以下Maven依赖。

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.11</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-clients_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table</artifactId>
  <version>1.13.2</version>
  <type>pom</type>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-common</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-json</artifactId>
  <version>1.13.2</version>
</dependency>
```

消费数据示例

1. 使用以下语句创建Lindorm源表格。

```
CREATE TABLE `lindorm_table` (
    `c1` INT,
  `col1` VARCHAR,
  `col2` VARCHAR,
  `col3` VARCHAR,
  PRIMARY KEY ( `c1` )
)
```

2. 创建数据订阅通道后，通过以下Java代码创建表格消费数据。

> **说明** 创建表格消费数据需要获取以下配置内容。
>
> - properties.bootstrap.servers：表示云原生多模数据库Lindorm宽表提供的消费接入点，您可以通过云原生多模数据库Lindorm控制台获取，在**实例列表**页面，左侧导航栏中选择**宽表引擎 > 数据订阅**，在**链接信息**区域中获取Lindorm宽表提供的消费接入点。
> - properties.group.id：表示消费组名称，您可以自定义。
> - topic：表示创建数据订阅通道时自定义的主题名。

```
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import static org.apache.flink.table.api.Expressions.$;
public class TestDebeziumFlink {
  public static void main(String[] args) {
    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    StreamTableEnvironment tableEnvironment = StreamTableEnvironment.create(env);
    // topic created by subscription
    String topicName = "test-topic";
    // replace with cdc endpoint
    String bootstrapServer = "localhost:9092";
    // consumer group id
    String groupID = "test-flink";
    // create table connected to source
    tableEnvironment.executeSql("CREATE TABLE test_table (\n" +
        "    c1 INT,\n" +
        "    `f:col1` STRING,\n" +
        "    `f:col2` STRING,\n" +
        "    `f:col3` STRING\n" +
        ") WITH (\n" +
        "    'connector' = 'kafka',\n" +
        "    'topic'     = '"+ topicName + "',\n" +
        "    'properties.bootstrap.servers' = '" + bootstrapServer + "',\n" +
        "    'properties.group.id' = '" + groupID + "',\n" +
        "    'debezium-json.schema-include' = 'true',\n" +
        "    'format'    = 'debezium-json'\n" +
        ")");
    // create print table
    tableEnvironment.executeSql("CREATE TABLE test_table_copy (\n" +
        "    c1 INT,\n" +
        "    `f:col1` STRING,\n" +
        "    `f:col2` STRING,\n" +
        "    `f:col3` STRING\n" +
        ") WITH (\n" +
        "    'connector' = 'print'" +
        ")");
    Table testTable = tableEnvironment.from("test_table");
    testTable.select($("c1"), $("f:col1"), $("f:col2"), $("f:col3")).executeInsert("test_table_copy");
    tableEnvironment.executeSql("SELECT * FROM test_table").print();
  }
}
```

3. 结果显示示例。

```
7> +I[1001, yq, null, null]
8> +I[1002, row2, null, null]
+----+-------------+--------------------------------+--------------------------------+--------------------------------+
| op |          c1 |                         f:col1 |                         f:col2 |                         f:col3 |
+----+-------------+--------------------------------+--------------------------------+--------------------------------+
| +I |        1002 |                           row2 |                         (NULL) |                         (NULL) |
| +I |        1001 |                             yq |                         (NULL) |                         (NULL) |
7> -U[1001, yq, null, null]
7> +U[1001, yq2, yq-sleep, null]
| -U |        1001 |                             yq |                         (NULL) |                         (NULL) |
| +U |        1001 |                            yq2 |                       yq-sleep |                         (NULL) |
7> -D[1001, yq2, yq-sleep, null]
| -D |        1001 |                            yq2 |                       yq-sleep |                         (NULL) |
```

14.5.4. 通过Push模式创建数据订阅

本文介绍通过创建Lindorm Streams实现数据订阅功能，数据订阅功能的Push模式是将云原生多模数据库Lindorm表格的增量数据推送至下游消息队列Kafka，您可以根据自身业务需求自由消费增量数据，搭建或实现多种业务场景。

推送流程

数据订阅功能将云原生多模数据库Lindorm表格的增量数据消息主动推送至下游Kafka，推送流程如下。

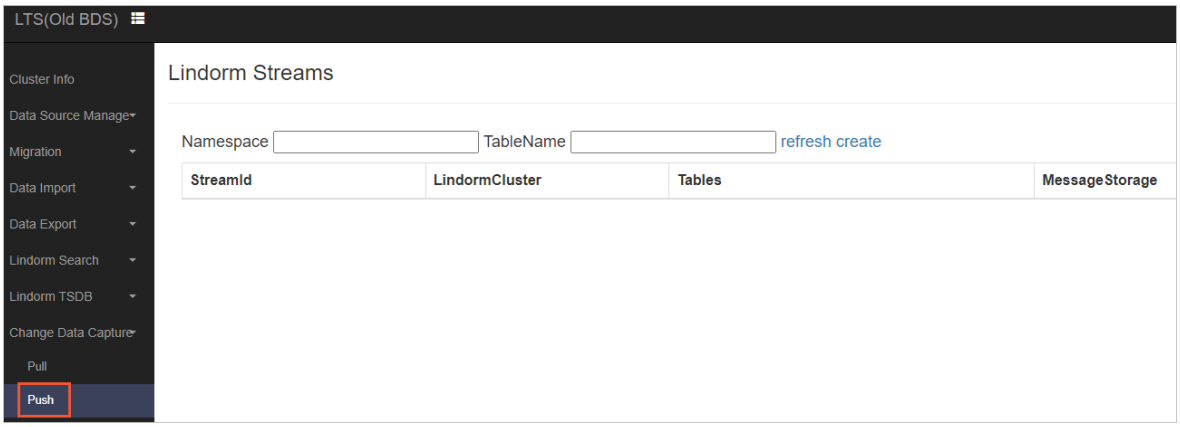


前提条件

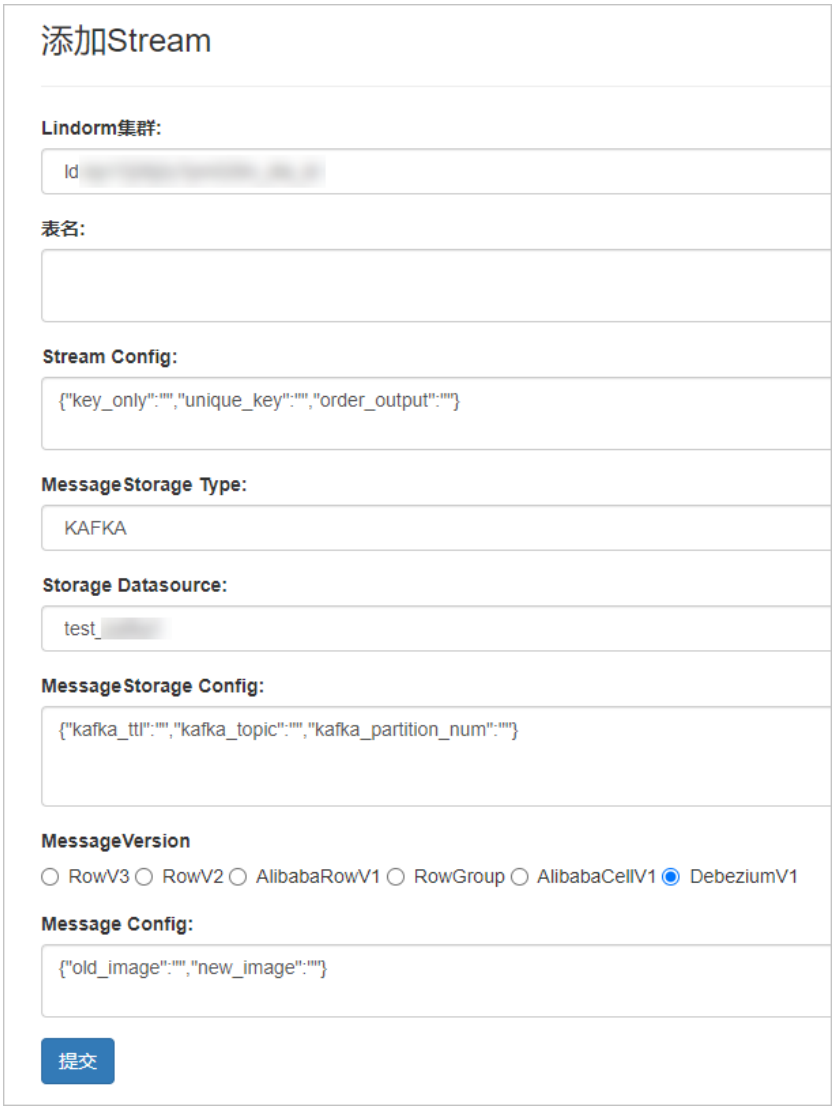
- 已将客户端IP添加至白名单中，具体操作请参见设置白名单。
- 确保源实例、目标实例和LTS服务网络互通，详细信息请参见网络互通。
- 已添加Lindorm宽表数据源，具体操作请参见Lindorm宽表数据源。
- 已添加Kafka数据源，具体操作请参见Kafka数据源。

创建Lindorm Streams

1. 进入LTS（原BDS）页面，在左侧导航栏中，选择**数据订阅 > Push模式**。



2. 单击create，并配置以下参数。



参数	描述
Lindorm集群	选择已添加Lindorm宽表数据源。
表名	需要填写数据订阅的表名，格式为：namespace.tablename，举例如下。 ◦ ns1.table1表示订阅空间名称为ns1中的表table1数据。 ◦ ns2.*表示订阅空间名称为ns2中的所有表数据。
Stream Config	◦ `key_only` 表示是否只输出主键，默认false。 ◦ `order_output` 表示是否开启key级别保序，默认false。
MessageStorage Type	选择KAFKA。
Storage Datasource	选择已添加Kafka数据源。
MessageStorage Config	`kafka_topic` 表示需要填写Kafka topic的名称。

参数	描述
MessageVersion	消费格式，默认格式为DebeziumV1。
Message Config	○ `old_image` 表示消息中是否包含当前行修改前的值，默认true。 ○ `new_image` 表示消息中是否包含修改后整行的值，默认true。

3. 单击**提交**。

14.5.5. 数据消费格式

本文介绍实时数据订阅功能的数据消费格式定义说明和示例，默认格式为Debezium Format V2.0。

数据消费定义说明

数据消费格式如下代码，字段说明如下表所示。

```
{
  "payload": {
    "op": "u", // (1)
    "ts_ms": 1465491411815, // (2)
    "before": { // (3)
      "id": 1004,
      "name": "Jane",
    },
    "after": { // (4)
      "id": 1004,
      "name": "Anne",
    },
    "source": { // (5)
      "version": "v1.0",
      "db": "ld-xxxx",
      "namespace": "default",
      "table": "customers",
      "ts_ms": 1465491411807
    }
  },
  "schema": { // (6)
  "type": "struct",
  "fields": [
     {
        "type": "string",
        "optional": false,
        "field": "op"
   }, {
        "type": "int64",
        "optional": false,
        "field": "ts_ms"
   }, {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
     }, {
            "type": "string",
            "optional": false,
            "field": "name"
          }
        ],
        "optional": true,
```

```
        "field": "before"
    }, {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
      }, {
            "type": "string",
            "optional": false,
            "field": "name"
      }
        ],
        "optional": true,
        "field": "after"
    }, {
        "type": "struct",
        "fields": [
          {
            "type": "string",
            "optional": false,
            "field": "version"
          }, {
            "type": "string",
            "optional": false,
            "field": "db"
      }, {
            "type": "string",
            "optional": false,
            "field": "namespace"
      }, {
            "type": "string",
            "optional": false,
            "field": "table"
      }, {
            "type": "int64",
            "optional": false,
            "field": "ts_ms"
      }
        ],
        "optional": false,
        "field": "source"
    }
     ],
   "optional": false
  },
}
```

Item	Field name	描述
1	payload.op	• c表示对数据进行新增操作。 • u表示对数据进行更新操作。 • d表示对数据进行删除操作。 • r表示对数据进行全量导出操作，暂时不会涉及到。
2	payload.ts_ms	表示写入Kafka的unix时间戳。
3	payload.before	表示导出整行数据的最新值。
4	payload.after	表示导出整行数据更新前的值。

5	payload.source	表示操作的额外信息，支持额外添加。 • version：消息对应Lindorm数据库的版本号。 • db：源集群。 • namespace：表的namespace。 • table：表名。 • ts_ms：记录更新Lindorm的时间。
6	schema	根据payload内容自动生成，对整个JSON的结构及所有字段类型进行说明。默认都包括schema信息。整个schema的结构是递归的。 • field：字段名称。 • type：field字段名对应的字段类型。 • name：schema字段所属模式名称。 • fields：递归说明当前字段包含的子字段内容。 • optional：该字段是否可选。

数据消费示例

在Lindorm数据库中创建如下schema。

```
CREATE TABLE customers (id varchar,first_name varchar,last_name varchar,constraint pk primary key(id))
```

- 插入数据的数据消费示例。

```
{
  "schema": {...},
  "payload": {
    "op": "c",
    "ts_ms": 1465491411815,
    "before": null,
    "after": {
      "id": "1004",
      "first_name": "Anne",
      "last_name": "Kretchmar",
    },
    "source": {
      "version": "v1.0",
      "db": "ld-xxxx",
      "namespace": "default",
      "table": "customers",
      "ts_ms": 1465491411807
    }
  }
}
```

- 更新数据的数据消费示例。

```
{
  "schema": {...},
  "payload": {
    "op": "u",
    "ts_ms": 1465491411815,
    "before": {
      "id": "1004",
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
    },
    "after": {
      "id": "1004",
      "first_name": "Anne",
      "last_name": "Kretchmar",
    },
    "source": {
      "version": "v1.0",
      "db": "ld-xxxx",
      "namespace": "default",
      "table": "customers",
      "ts_ms": 1465491411807
    }
  }
}
```

- 删除一行数据的数据消费示例。

```
{
  "schema": {...},
  "payload": {
    "op": "d",
    "ts_ms": 1465491411815,
    "before": {
      "id": "1004",
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
    },
    "after": null,
    "source": {
      "version": "v1.0",
      "db": "ld-xxxx",
      "namespace": "default",
      "table": "customers",
      "ts_ms": 1465491411807
    }
  }
}
```

- 删除列的数据消费示例。

```
{
  "schema": {...},
  "payload": {
    "op": "u",
    "ts_ms": 1465491411815,
    "before": {
      "id": "1004",
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
    },
    "after": {
      "id": "1004",
      "first_name": "Anne Marie",
    },
    "source": {
      "version": "v1.0",
      "db": "ld-xxxx",
      "namespace": "default",
      "table": "customers",
      "ts_ms": 1465491411807
    }
  }
}
```

14.6. 宽表引擎数据迁移

14.6.1. 不停机迁移

14.6.1.1. 全增量数据同步

本文主要介绍全增量数据同步的使用场景，功能列表，迁移优势，使用限制以及创建数据同步任务等操作。

使用场景

- 大版本升级：HBase1.x升级HBase2.x。
- 跨地域迁移：例如从青岛机房迁移到北京机房。
- 集群升配：从4核8G迁移到8核16G集群。
- 业务拆分：将部分业务迁移到新的集群。

功能列表

- 支持094、098、1.x、2.x、Lindorm任意版本之间的不停机迁移。
- 支持表结构迁移、实时数据同步、全量数据迁移。
- 支持整库迁移、Namespace迁移、表级别迁移。
- 迁移支持表重命名。
- 迁移支持指定时间范围、Rowkey范围、指定列。
- 提供OpenAPI，支持调用API创建迁移任务。

迁移优势

- 不停服数据迁移，同时搞定历史数据迁移、实时增量数据的同步。
- 迁移过程中不会和源集群的HBASE交互，只读取源集群的HDFS，尽可能减少对源集群在线业务的影响。
- 文件层的数据拷贝比通常API层的数据迁移通常能节省50%以上的流量高效性。
- 单个节点迁移速度可达到150 MB/s，节点数支持水平扩展，能够满足支持TB、PB级别的数据迁移稳定性。
- 有完善的错误重试机制，实时监控任务速度和进度，支持任务失败报警。
- 自动同步Schema，保证分区一致。

使用限制

- 不支持开启Kerberos集群。
- 不支持单节点云HBase实例。
- 由于网络原因，不支持经典网络下的云HBase实例。
- 增量数据同步基于HBase WAL实现异步数据同步，Bulkload的数据和不写WAL的数据将不会被同步。

注意事项

- 迁移前请确认目标集群的HDFS容量，防止迁移过程中出现容量写满的情况。
- 增量同步提交前，推荐修改一下源集群的日志保留时间，给增量同步出错预留一些处理的时间（hbase-site.xml的hbase.master.logcleaner.ttl调大为12小时以上，重启HMaster）。
- 客户无需在目标集群创建表，BDS同步服务会自动创建和源集群一样的表，包括分区的信息。客户自建的目标表可能会和源表分区很不一致，这可能会导致迁移完成之后，目标表会进行频繁的split、compaction，如果表的数据量十分庞大，可能会导致这个过程非常耗时。
- 如果源表带有coprocessor，在创建目标表的时候需要确保目标集群包含coprocessor对应的JAR包。

增量同步中的日志管理

- 开启增量同步后，如果不消费数据，默认日志会保留48小时，超时后订阅关系自动取消，保留的数据自动删除。
- 什么场景会导致“不消费数据”：没有终止任务的前提下直接释放LTS集群；同步任务暂停；任务异常阻塞。

准备工作

- 检查源集群、目标集群、LTS的网络连通性。
- 添加HBase、Lindorm数据源。
- 登录LTS操作页面，具体操作请参见创建同步任务。

创建任务

1. 在左侧导航选择**Lindorm/HBase迁移 > 一键迁移**。
2. 单击**创建任务**。



- **任务名**：只支持英文字符、数字，非必填，默认任务ID作为任务名。
- 根据界面提示设置**源集群**和**目标集群**。
- 选择需要的操作。
  - **表结构迁移**：在目标集群创建表（schema、分区信息一致）， 目标集群表已存在会跳过。
  - **实时数据复制**：同步源集群实时增量的数据。
  - **历史数据迁移**：文件级别的全量文件物理迁移。
- **表映射**：填写表名，使用换行符分割。
- **高级配置**：可不填。

查看任务

在左侧导航选择**Lindorm/HBase迁移 > 一键迁移**查看任务。

查看任务详情

1. 在左侧导航选择**Lindorm/HBase迁移 > 一键迁移**。
2. 单击需要查看的任务名。查看对应任务的执行情况。

关于切流

1. 等待全量任务迁移完成，增量同步的时延比较小（几秒或者几百毫秒）。
2. 开启LTS数据抽样校验，对于大表的抽样校验，抽样比例不宜过大，避免对在线业务产生影响。
3. 业务验证。
4. 业务切流。

14.6.1.2. 网络打通

Lindorm宽表引擎数据迁移之前需要源实例和目标实例的网络互通，本文介绍网络互通的方法。

源实例网络环境	目标实例网络环境	LTS服务是否支持	网络如何互通
VPC	相同VPC	是	LTS控制台关联HBase、Lindorm实例。
VPC	不同的VPC	是	1. VPC之间需要建立一个高速通道。 2. HBase、Lindorm实例添加LTS实例访问白名单。 3. 开通实例HDFS（仅云HBase和Lindorm需要）。
经典网络	相同的经典网络	否	1. HBase、Lindorm实例添加LTS实例访问白名单。 2. 开放HDFS、HBASE/Lindorm的端口。
经典网络	VPC	是	1. 需要创建 VPC 的 ClassicLink，经典网络的ECS需要绑定对应的ClassicLink。 2. HBase、Lindorm实例添加LTS实例访问白名单。 3. 开通实例的HDFS（仅云HBase、Lindorm需要）。
经典网络	云HBase经典网络	否	1. 需要临时购买几台经典网络ECS。 2. HBase、Lindorm实例添加LTS实例访问白名单。 3. 开通实例HDFS （仅云HBase、Lindorm需要）。
云HBase经典网络	VPC	否	1. 需要临时购买几台经典网络ECS做中转，迁移完可以释放，ECS和VPC HBase/Lindorm属于一个地域（例如华东2、华东1）。 2. ECS和VPC打通走Classiclink。 3. HBase、Lindorm实例添加LTS实例访问白名单。 4. 开通实例HDFS （仅云HBase、Lindorm需要）。

迁移过程访问HDFS和HBase涉及的端口

类型	默认端口	说明
fs.default.name	8020	Namenode RPC 交互端口
dfs.http.address	50070	NameNode Web管理端口

类型	默认端口	说明
dfs.datanode.ipc.address	50020	Datanode的RPC服务器地址和端口
dfs.datanode.address	50010	datanode控制端口
zookeeper.property.clientPort	2181	zk的地址
hbase.regionserver.port	16020	HBase RegionServer绑定的端口
hbase.master.port	16000	HBase Master绑定的端口

14.6.1.3. 公网迁移专项

LTS集群默认不能访问公网，这给部分想通过公网将本地HBase迁移到云HBase的客户带来了不小的麻烦。本文主要介绍如何开通LTS集群访问公网的能力。

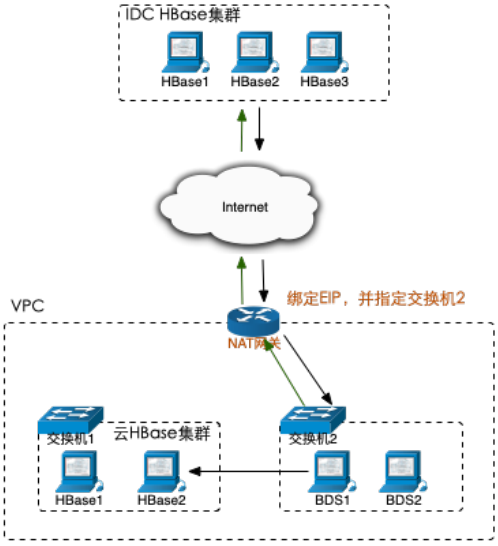
前提条件

- 本地集群对公网开放。
- 对LTS开通本地集群访问的权限。
- 购买LTS。

LTS打通公网流程

LTS默认情况下是不能访问公网的，为了打通LTS访问公网的能力主要分为一下几个步骤：

1. 对LTS所在VPC开通NAT网关。
2. 根据迁移的规模开通对应规格的公网EIP。
3. 绑定EIP到NAT网关，并指定LTS所在的交换机。

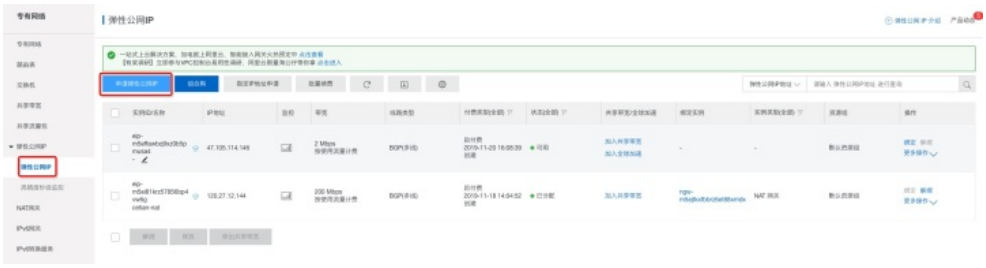


具体步骤

1. 针对LTS所在的VPC开通NAT网关，详见什么是NAT网关。



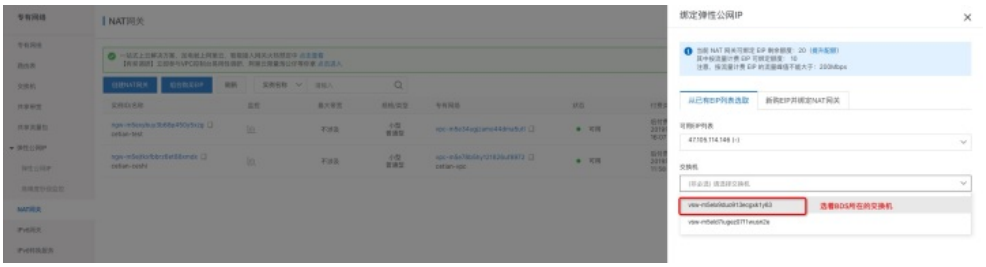
2. 创建对应规格的EIP。



3. NAT网关绑定EIP。



将EIP绑定到LTS所在的交换机上。



4. 绑定成功后，对应VPC交换机下的LTS就有了访问公网的能力。
5. 登录LTS页面，添加HBase数据源， 这里需要注意一下添加本地HBase数据源时需要填写etc/hosts，并把内网的IP替换成公网对外的IP。



6. 增量数据同步和历史数据迁移。

14.6.1.4. Phoenix全增量迁移

本文介绍Lindorm宽表引擎Phoenix的全增量数据迁移。

使用场景

- 跨地域迁移：从青岛机房迁移到北京机房。
- 集群升配：从4核8G迁移到8核16G集群。
- 业务拆分：将部分业务迁移到新的集群。

功能列表

- 仅支持同版本之间Phoenix迁移，不支持phoenix 4.x跨版本迁移到phoenix 5.x。
- 支持表结构迁移、实时数据同步、全量数据迁移。
- 支持整库迁移、Namespace迁移、表级别迁移。
- 迁移支持表重命名。
- 迁移支持指定时间范围、Rowkey范围、指定列。
- 提供OpenAPI，支持调用API创建迁移任务。

迁移优势

- 不停服数据迁移，同时搞定历史数据迁移、实时增量数据的同步。
- 迁移过程中不会和源集群的HB交互，只读取源集群的HDFS，尽可能减少对源集群在线业务的影响。
- 文件层的数据拷贝比通常API层的数据迁移通常能节省50%以上的流量高效性。
- 单个节点迁移速度可达到150 MB/s，节点数支持水平扩展，能够满足支持TB、PB级别的数据迁移稳定性。
- 有完善的错误重试机制，实时监控任务速度和进度，支持任务失败报警。
- 自动同步Schema，保证分区一致。

使用限制

- 不支持开启Kerberos集群。
- 不支持单节点云HBase实例。
- 由于网络原因，不支持经典网络下的云HBase实例。
- 增量数据同步基于HBase WAL实现异步数据同步， Bulkload的数据、不写WAL的数据将不会被同步。

准备工作

- 检查源集群、目标集群、LTS的网络连通性。
- 添加Phoenix数据源。
- 已开通LTS服务，具体操作请参见开通LTS。
- 已进入LTS（原BDS）页面，具体操作请参见同步服务页面。

创建任务

1. LTS（原BDS）页面中，在左侧导航栏选择**Lindorm/HBase迁移** > **Phoenix->Phoenix**。
2. 单击**创建任务**。
   - **任务名**：只允许英文字符、数字，非必填。默认任务ID作为任务名。
   - 根据界面提示设置**源集群**和**目标集群**。
   - 选择需要的操作。
     - **表结构迁移**：在目标集群创建表（schema、分区信息一致），目标集群表已存在会跳过。
     - **实时数据复制**：同步源集群实时增量的数据。
     - **历史数据迁移**：文件级别的全量文件物理迁移。
   - **同步或迁移的表**：填写表名，使用换行符分割。
   - **高级配置**：可不填。

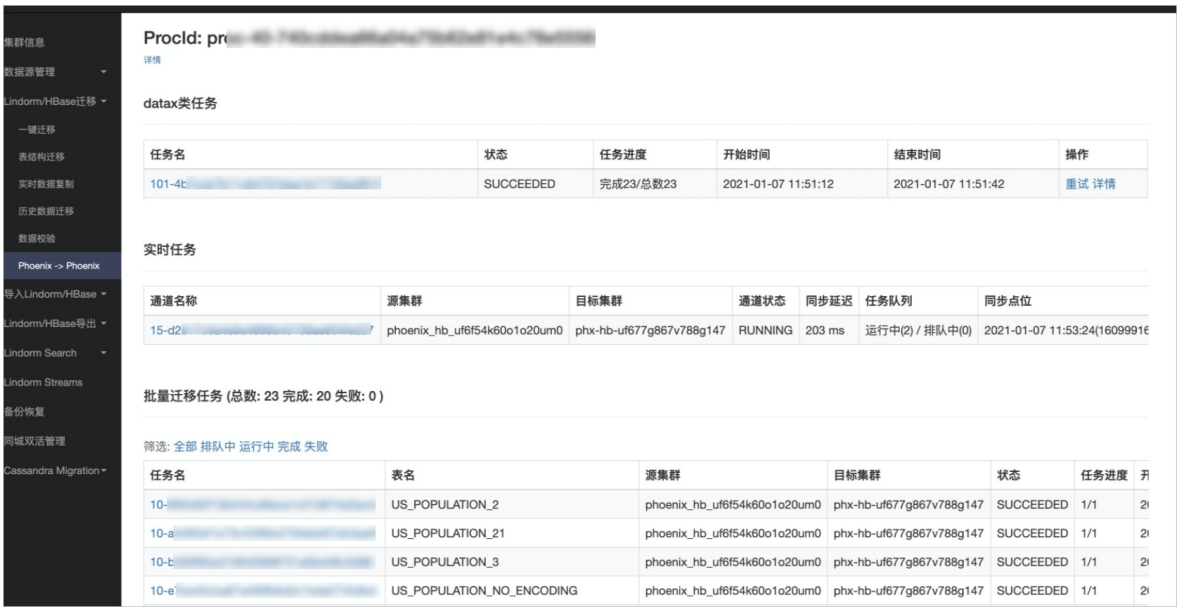
查看任务

LTS（原BDS）页面中，在左侧导航栏选择**Lindorm/HBase迁移** > **Phoenix->Phoenix**查看任务。



查看任务详情

1. LTS（原BDS）页面中，在左侧导航栏选择**Lindorm/HBase迁移 > Phoenix->Phoenix**。
2. 单击需要查看的任务名，查看对应任务的执行情况。



关于切流

1. 等待全量任务迁移完成，增量同步的时延比较小（几秒或者几百毫秒）。
2. 开启LTS数据抽样校验，对于大表的抽样校验，抽样比例不宜过大，避免对在线业务产生影响。
3. 业务验证。
4. 业务切流。

注意事项

- 迁移前请确认目标集群的HDFS容量，防止迁移过程中出现容量写满的情况。
- 增量同步提交前，建议您修改源集群的日志保留时间，为增量同步出错时预留处理时间（hbase-site.xml的hbase.master.logcleaner.ttl设置为12小时以上并重启HMaster）。
- 您无需在目标集群中创建表，BDS同步服务会自动创建和源集群相同的表，包括分区的信息。您自建的目标表可能会和源表分区很不一致，这可能会导致迁移完成之后，目标表会进行频繁的split和compaction操作，如果表的数据量十分庞大，可能会导致这个过程非常耗时。
- 如果源表带有coprocessor，在创建目标表的时候需要确保目标集群包含coprocessor对应的JAR包。

14.6.2. 数据导入

14.6.2.1. RDS全增量同步

本文主要介绍RDS全量同步和增量同步的操作过程。

使用场景

- RDS低成本历史库。
- RDS数据全量迁移至Lindorm。

前提条件

- 已登录LTS页面，具体操作请参见登录同步任务。
- LTS、云数据库HBase迁移集群、RDS实例的网络已连通（都在同一个VPC中可以跳过）。

功能列表

- RDS数据全增量一体同步到Lindorm宽表模型（兼容HBase、Cassandra访问）。
- RDS数据变换，请参见配置说明。
- RDS多表同步。

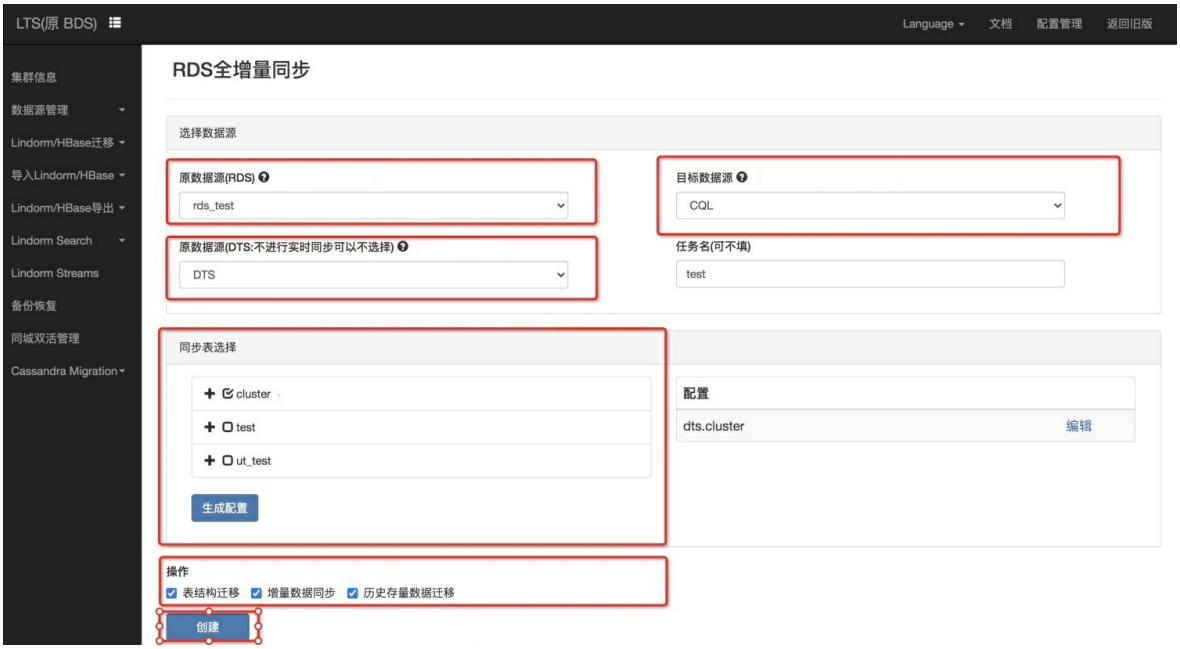
使用限制

- RDS全量同步支持数据源MySQL。
- RDS增量同步支持数据源DTS。
- 支持目标数据源LindormSQL，Lindorm宽表模型（兼容HBase、Cassandra访问）。

操作步骤

1. 在LTS操作页面中，单击**导入Lindorm/HBase > RDS全增量同步**。
2. 单击**创建任务**。
3. 选择RDS数据源、DTS数据源以及目标数据源。

> 说明
>
> - 添加Lindorm宽表数据源。
> - 添加RDS数据源。
> - 添加DTS数据源。



4. 单击**编辑**可以查看默认配置说明，也可以修改配置说明，请参见配置说明。
5. 选择要同步的表，单击**生成配置**。

> **说明**
> - RDS全增量同步先进行全量历史数据迁移，迁移完成后在进行增量数据迁移。
> - 导入CQL默认生成配置同RDS表字段和类型一一对应，需要更改可以手动编辑修改字段名和对应关系，详见配置说明。
> - 导入HBase/Lindorm默认生成列簇f，RDS中的字段会和f下列一一对应，同时rowkey为RDS主键字符串拼接。
> - 默认生成配置会跳过RDS删除操作如果保留需要手动修改配置，详见配置说明。

6. 单击**创建**。

配置说明

SQL表同步配置说明，具体语法请参见jtwig语法说明。

```
{
    "reader": {
        "querySql": [
            "select * from dts.cluster where id < 1000",//全量同步查询语句,一个语句对应一个读取线程
      "select * from dts.cluster where id >= 1000"//建议进行拆分提高速度和减小重试代价
        ]
    },
    "writer": {
        "columns": [
            {
                "name": "id", //目标表中字段名称
                "value": "id",//原表中字段名称
                "isPk": true , //是否是主键
                "type": "BIGINT" //可以不填写默认同RDS表类型一致
            },
      {
                "name": "cluster_id",
                "value": "cluster_id",
                "isPk": false
            },
      {
                "name": "id_and_cluster",
                "value": "{{concat(id, cluster_id)}}",//支持Jtwig域名对数据进行变换
                "isPk": true
            },
        ],
        "config": {
            "skipDelete": true //跳过删除操作
        }
    }
}
```

CQL表同步配置说明

```
{
    "reader": {
        "querySql": [
            "select * from dts.cluster where id < 1000",//全量同步查询语句,一个语句对应一个读取线程
      "select * from dts.cluster where id >= 1000"//建议进行拆分提高速度和减小重试代价
        ]
    },
    "writer": {
        "columns": [
            {
                "name": "id", //目标表中字段名称
                "value": "id",//原表中字段名称
                "isPk": true , //是否是主键
        "type": "bigint" //CQL中字段类型,可以不填写默认同RDS表类型一致
            },
      {
                "name": "cluster_id",
                "value": "cluster_id",
                "isPk": false
            },
      {
                "name": "id_and_cluster",
                "value": "{{concat(id, cluster_id)}}",//支持Jtwig域名对数据进行变换
                "isPk": true
            },
        ],
        "config": {
            "skipDelete": true //跳过删除操作
        }
    }
}
```

HBase API访问同步配置说明

```
{
    "reader": {
        "querySql": [
            "select * from dts.cluster where id < 1000",//全量同步查询语句,一个语句对应一个读取线程
      "select * from dts.cluster where id >= 1000"//建议进行拆分提高速度和减小重试代价
        ]
    },
    "writer": {
        "columns": [
            {
                "name": "f:id",//目标表中字段名称
                "value": "id", //原表中字段名称
        "isPk": false //不影响同步忽略
            },
            {
                "name": "f:cluster_id",
                "value": "cluster_id",
                "isPk": false
            },
      {
        "name": "f:id_and_cluster",
                "value": "{{concat(id, cluster_id)}}",//支持Jtwig域名对数据进行变换
       }
        ],
        "rowkey": {
            "value": "id" //hbase模型中rowkey由RDS哪些字段组成,支持Jtwig语法
        },
        "config": {
            "skipDelete": true//跳过删除操作
        },
        "table": {
            "name": "dts:cluster",// Lindorm/HBase中表名
            "parameter": {
                "compression": "ZSTD",//Lindorm/HBase中,新建表压缩算法,推荐使用ZSTD
        "split":["1", "5", "9", "b"] //指定splitkey,对新建表进行预分区
            }
        },
        "sourceTable": "dts.cluster"
    }
}
```

RDS和CQL类型对应关系。

RDS	CQL
TINYINT	boolean
SMALLINT	smallint
MEDIUMINT	int
INTEGER	int
BIGINT	bigint
FLOAT	float
DOUBLE	double
DECIMAL	decimal

RDS	CQL
DATE	date
TIME	time
YEAR	int
DATETIME	text
TIMESTAMP	timestamp
CHAR	varchar
VARCHAR	varchar
TINYBLOB	blob
TINYTEXT	text
BLOB	blob
TEXT	text
MEDIUMBLOB	blob
MEDIUMTEXT	text
LONGBLOB	blob
LONGTEXT	text

14.6.2.2. RDS全量导入

本文介绍通过LTS通道服务进行RDS全量导入。

前提条件

- 已购买LTS数据迁移同步服务，配置LTS操作页面账户密码。
- 已连接LTS和HBase迁移集群、RDS实例的网络。
- 已添加HBase集群数据源或者Phoenix数据源。
- 已添加RDS数据源。

适用版本

- **HBase**
  - 自建HBase1.x、2.x
  - EMR HBase
  - 标准版云HBase、增强版云HBase（集群版本）
- **Mysql**
  - 自建Mysql
  - RDS for Mysql

任务创建

1. 开通并登录LTS。
2. 进入LTS操作页面，在左侧导航栏中选择**导入Lindorm/HBase > RDS全量数据导入**。
3. 单击**创建迁移任务**，设置**RDS数据源**和**目标数据源(HBase、Phoenix)**，输入需要迁移的表Mapping信息。



4. 单击**创建同步通道**。
5. 查看任务进度。



6. 迁移完成之后， 查看HBase表。



参数说明

- **HBase表映射**

```
{
  "reader": {
    "querySql": [ // 根据querySql的数量进行任务拆分，分布式运行
      "select id, title, content from rds.test where id < 8",
      "select id, title, content from rds.test where id >= 8"
    ]
  },
  "writer": {
    "columns": [
      {
        "name": "f1:col1",
        "value": "{{ concat(title, id) }}"  // 将mysql中的数据title和id字段进行拼接，作为HBase f1:col1列的值
      },
      {
        "name": "f1:col2",
        "value": "content",
        "type": "string" // type 字段可选， 默认都按string类型类处理写入HBase
      },
      {
        "name": "f1:*" // mysql 没有匹配到的列会走默认的匹配
      }
    ],
    "rowkey": {
      "value": "{{ concat('idg', id) }}"
    },
    "tableName": "default:t1"
  }
}
```

- 支持简单的表达式，计算表达式为jtwig语法，如下：

```
{
  "name": "cf1:hhh",
  "value": "{{ concat(title, id) }}"
}
```

- 支持动态列，没有匹配到的列会走默认的匹配。

```
{
    "name": "cf1:*",
}
```

- **Phoenix表映射**

```
{
  "reader": {
    "querySql": [
      "select id, title, ts, datetime, date, time, b, f, d from rds.test where id < 8",
      "select id, title, ts, datetime, date, time, b, f, d from rds.test where id >= 8"
    ]
  },
  "writer": {
    "columns": [
      {
        "isPk": true,
        "name": "id"
      },
      {
        "name": "title",
        "value": "title" // mysql中title字段对应Phoenix中的title， 如果字段名相同可不填
      },
      {
        "name": "ts"
      },
      {
        "name": "datetime"
      },
      {
        "name": "date"
      },
      {
        "name": "time"
      },
      {
        "name": "b"
      },
      {
        "name": "f"
      },
      {
        "name": "d"
      }
    ],
    "tableName": "dtstest"
  }
}
```

14.6.2.3. RDS增量导入

本文介绍如何进行RDS增量导入。

前提条件

- 已购买LTS数据迁移同步服务，配置LTS操作页面账户密码。
- 已打通LTS和HBase迁移集群的网络。
- 已添加HBase集群数据源。
- 开通DTS订阅服务， 添加DTS数据源。

适用版本

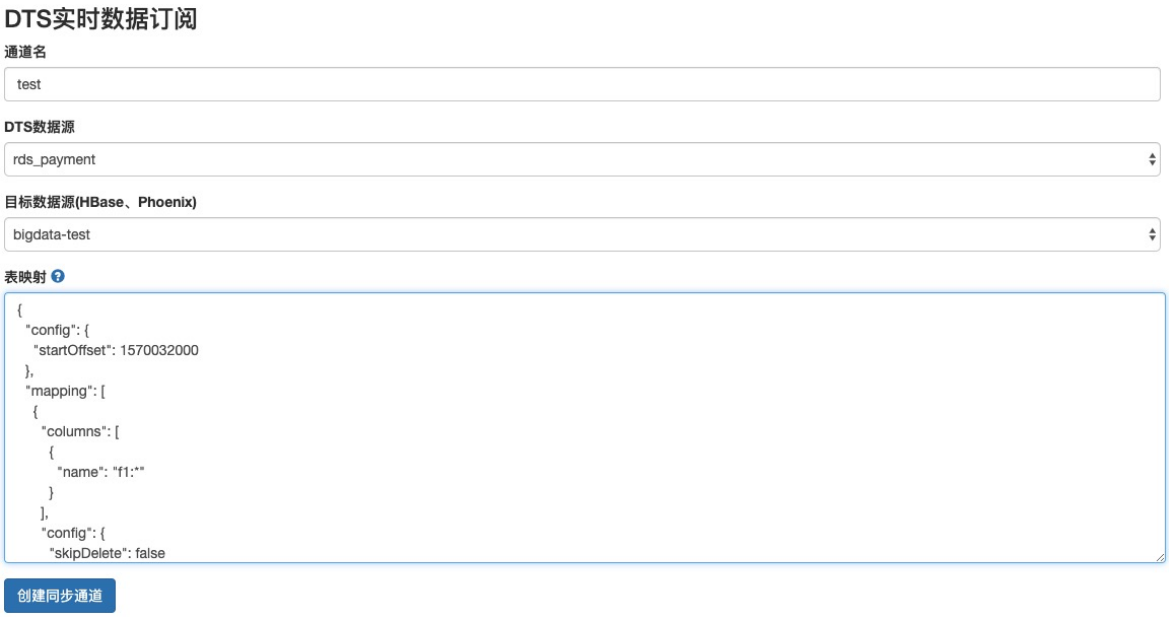
- 自建HBase1.x、2.x
- EMR HBase
- 标准版云HBase、增强版云HBase（单机版本、集群版本）
- 云HBase Phoenix

任务创建

1. 登录LTS操作页面，具体操作请参见登录LTS。
2. 在左侧导航栏中选择**导入Lindorm/HBase > RDS实时数据复制** 。



3. 单击**创建通道**，设置**RDS数据源**和**目标数据源(HBase、Phoenix)**，输入需要迁移的表Mapping信息。



4. 单击**创建同步通道**。

参数说明

- **HBase表映射**

```
{
  "mapping": [
    {
      "columns": [
        {
          "name": "cf1:hhh",
          "value": "{{ concat(title, id) }}"
        },
        {
          "name": "cf1:title",
          "value": "title"
        },
        {
          "name": "cf1:*"
        }
      ],
      "config": {
        "skipDelete": true
      },
      "rowkey": {
        "value": "{{ concat('idg', id) }}"
      },
      "srcTableName": "hhh_test.test",
      "targetTableName": "default:_test"
    }
  ]
}
```

参数	描述	必选
mapping[y].srcTableName	RDS源表表名。	是
mapping[y].targetTableName	HBase目标表名。	是
mapping[y].columns	RDS表和HBase表列的对应。	是
mapping[y].columns[x].name	对应的HBase的列名。	是
mapping[y].columns[x].value	对应的HBase列的计算表达式，该计算表达式为jtwig语法，当用户需要对源表的列进行简单计算得到rowkey时可以用。	是
mapping[y].config	表的同步策略。	否
mapping[y].rowkey	HBase表的rowkey的生成规则。	是

- 支持简单的表达式，如下

```
{
  "name": "cf1:hhh",
  "value": "{{ concat(title, id) }}"
}
```

- 支持动态列，没有匹配到的列会走默认的匹配。

```
{
    "name": "cf1:*",
}
```

- 支持指定订阅的起始时间（DTS订阅通道中包含指定时间戳以后的数据）。

```
{
  "config": {
      "startOffset":1569463200 // 秒
  },
  "mapping": [
      "srcTableName": "hhh_test.test",
      "targetTableName": "default:test",
      "columns": [
        {
          "name": "cf1:*"
        }
      ],
      "config": {
        "skipDelete": true
      },
      "rowkey": {
        "value": "{{ concat('idg', id) }}"
      }
    }
  ]
}
```

- DML支持情况

操作	支持	备注
INSERT	是	对应HBase的PUT
UPDATE	是	对应HBase的PUT
DELETE	是	用户可以配置是否同步源端的DELETE，默认不同步

- **Phoenix表映射**

```
{
  "mapping": [
    {
      "srcTableName": "hhh_test.phoenix_test",
      "targetTableName": "phoenix_test",
      "config": {
        "skipDelete": true
      },
      "columns": [
        {
          "name": "id",
          "isPk": true
        },
        {
          "name": "title",
          "value": "title"
        },
        {
          "name": "ts",
          "value": "ts"
        },
        {
          "name": "datetime",
          "value": "datetime"
        }
      ]
    }
  ]
}
```

参数	描述	必选
mapping[y].srcTableName	RDS源表表名	是
mapping[y].targetTableName	Phoenix目标表名	是
mapping[y].columns	RDS表和Phoenix表列的对应	是
mapping[y].columns[x].name	Phoenix的列名	是
mapping[y].columns[x].value	对应RDS列名	是
mapping[y].columns[x].isPk	指定主键列	是
mapping[y].config	表的同步策略	否
mapping[y].rowkey	HBase表的rowkey的生成规则	是

14.6.2.4. 导入日志服务增量数据

本文介绍通过云原生多模数据库Lindorm控制台将日志服务增量数据导入至Lindorm宽表。

前提条件

- 已登录LTS操作页面，具体操作请参见创建同步任务。
- 已添加Lindorm SQL数据源，具体操作请参见添加Lindorm SQL数据源。
- 已添加Loghub数据源。

支持导入的目标表类型

使用Lindorm SQL创建的表。

操作步骤

1. 在LTS操作页面，单击**导入Lindorm/HBase > SLS实时数据复制**。
2. 在**SLS实时数据复制**页面，单击**创建任务**。
3. 自定义**通道名称**，选择源集群和目标集群，填写同步或迁移的表。
4. 单击**创建**。创建成功后，查看通道详情。



参数说明

```
{
  "reader": {
    "columns": [
      "__client_ip__",
      "C_Source",
      "id",
      "name"
    ],
    "consumerSize": 2, // 指定有多少个consumer订阅loghub的数据，默认是1
    "logstore": "LTS-test"
  },
  "writer": {
    "columns": [
      {
        "name": "col1",
        "value": "{{ concat('xx', name) }}" // 支持表达式
      },
      {
        "name": "col2",
        "value": "__client_ip__" // 列名映射
      },
      {
        {
            "isPk":true,//是否是主键
            "name":"id",//主键不需要指定cf
            "value":"id"
        }
      }
    ]
    "tableName": "default.sls"
  }
}
```

支持简单jtwig语法。

```
{
  "name": "hhh",
  "value": "{{ concat(title, id) }}"
}
```

14.6.2.5. 通用批量导入服务

云原生多模数据库Lindorm支持Bulkload（批量快速导入数据）功能，可以更快更稳定的导入数据。本文介绍批量快速导入数据操作。

功能特性

批量快速导入数据功能支持数据文件旁路加载，不需要经过数据API写入链路并且不需要占用实例计算资源，批量快速导入数据与通过API导入数据相比有以下优势：

- 导入数据更快，速度可以提升10倍以上。
- 在线服务更稳定，不占用在线服务资源。
- 资源使用更灵活，在离线资源中分开使用更加灵活。
- 支持多种数据源导入，包括CSV、ORC、Parquet、MaxCompute等。
- 使用简单。您无需开发任何代码就可以实现数据的批量快速旁路加载。
- 成本低。LTS Bulkload基于Serverless Spark提供的云原生弹性能力，根据您的需求提供弹性计算资源按量收费，您无需长期配置计算资源，可以降低使用成本。

前提条件

- 已开通并登录LTS数据同步服务，具体操作请参见开通并登录LTS。
- 已开通Lindorm计算引擎，具体操作请参见开通与变配。
- LTS Bulkload功能目前处于公测中，开通此功能请提交工单或联系值班。

数据源支持

源数据源	目标数据源
MaxCompute Table	Lindorm宽表引擎
HDFS CSV	
HDFS Parquet	

提交方式

快速导入数据任务支持以下方式提交。

- 通过LTS操作页面提交
- 通过API接口提交

通过LTS操作页面提交

1. 登录LTS操作页面，具体操作参见开通并登录LTS。
2. 在左侧导航栏，选择**数据源管理 > 添加数据源**，添加以下数据源。
   - 添加ODPS数据源，具体操作请参见ODPS数据源。
   - 添加Lindorm宽表数据源，具体操作请参见Lindorm宽表数据源。
   - 添加HDFS数据源，具体操作请参见添加HDFS数据源。
3. 在左侧导航栏中，选择**导入Lindorm/HBase > Bulkload**。



4. 单击**创建任务**，配置以下参数。

配置项	参数	描述
选择数据源	源数据源	选择添加的ODPS数据源或者HDFS数据源。
	目标数据源	选择添加的Lindorm宽表数据源。
插件配置	Reader配置	○ 如果源数据源为ODPS数据源，读插件配置说明如下： ■ table：ODPS的表名。 ■ column：需要导入的ODPS列名。 ■ partition：非分区表为空，分区表配置分区信息。 ■ numPartitions：读取时的并发度。 ○ 如果源数据源为HDFS数据源，CSV文件的读插件配置说明如下： ■ filePath：CSV文件的所在目录。 ■ header：CSV文件是否包含header行。 ■ delimite：CSV文件分隔符。 ■ column：CSV文件中列名以及对应类型。 ○ 如果源数据源为HDFS数据源，Parquet文件的读插件配置说明如下： ■ filePath：Parquet文件的所在目录。 ■ column：Parquet文件中的列名。 说明 读插件配置示例请参见配置示例。

配置项	参数	描述
	Writer配置	○ namespace：Lindorm宽表的namespace。 ○ lindormTable：Lindorm宽表名称。 ○ compression：压缩算法，目前仅支持zstd，不使用压缩算法配置为none。 ○ columns：根据导入至目标表类型填写。 ■ 如果导入至Lindorm宽表，columns需要配置Lindorm SQL宽表的列名，和读配置中的column对应。 ■ 如果导入至Lindorm兼容HBase表，columns需要配置HBase表标准的列名，和读配置中的column对应。 说明 写插件配置示例请参见配置示例。
作业运行参数配置	Spark Driver规格	选择Spark Driver规格。
	Spark Executor规格	选择Spark Executor规格。
	Executor数量	输入Executor的数量。
	spark配置	输入Spark配置，可不填写。

5. 单击**创建**。
6. 在**Bulkload**页面单击**任务名**查看快速导入数据任务。
   - 单击**任务名**，可以查看Spark任务的UI界面。
   - 单击**详情**，可以查看Spark任务的执行日志。

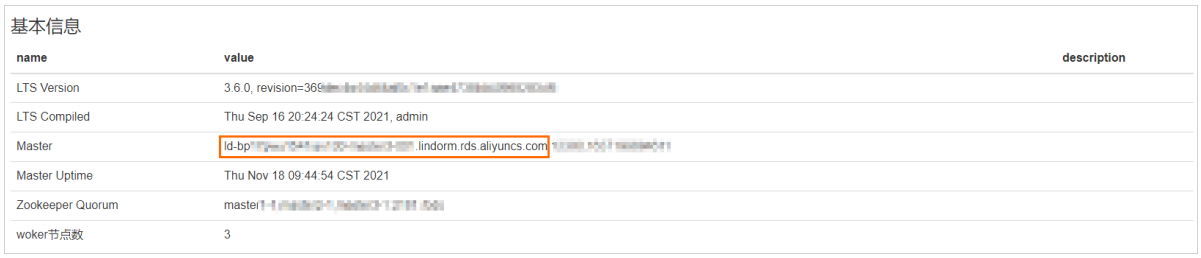


说明
源数据源迁移到Lindorm宽表，在Lindorm宽表不同分区的数据均匀情况下，数据容量为100 GB（压缩比为1:4）压缩后导入大概需要1个小时，根据具体情况不同可能会有区别。

通过API接口提交

提交任务接口

- 接口（POST）：`http://{BDSMaster}:12311/pro/proc/bulkload/create`，BDSMaster需要修改为Lindorm实例的Master hostname，可以登录Lindorm实例的LTS，在**集群信息**页面的**基本信息**区域获取。


基本信息

name	value	description
LTS Version	3.6.0, revision=369	
LTS Compiled	Thu Sep 16 20:24:24 CST 2021, admin	
Master	ld-bp .lindorm.rds.aliyuncs.com	
Master Uptime	Thu Nov 18 09:44:54 CST 2021	
Zookeeper Quorum	master	
woker节点数	3	

- 参数说明：

参数	说明
src	源数据源名称。
dst	目标数据源名称。
readerConfig	读插件配置信息，文件类型为JSON，读插件配置示例请参见配置示例。
writerConfig	写插件配置信息，文件类型为JSON，写插件配置示例请参见配置示例。
driverSpec	Driver的规格，包括small、medium、large和xlarge四种规格，推荐配置为large。
instances	Executor的数量。
filetype	如果源数据源为HDFS，需要填写文件类型为CSV或者Parquet。
sparkAdditionalParams	扩展参数可以不填。

- 示例：

```
curl -d "src=hdfs&dst=ld&readerConfig={\"filePath\":\"parquet/\",\"column\":[\"id\",\"intcol\",\"doublecol\",\"stringcol\",\"string1col\",\"decimalcol\"]}&writerConfig={\"columns\":[\"ROW||String\",\"f:intcol||Int\",\"f:doublecol||Double\",\"f:stringcol||String\",\"f:string1col||String\",\"f:decimalcol||Decimal\"],\"namespace\":\"default\",\"lindormTable\":\"bulkload_test\",\"compression\":\"zstd\"}&driverSpec=large&instances=5&fileType=Parquet" -H "Content-Type: application/x-www-form-urlencoded" -X POST http://{LTSMaster}:12311/pro/proc/bulkload/create
```

返回内容如下，message为任务ID。

```
{"success":"true","message":"proc-91-ff383c616e5242888b398e51359c****"}
```

获取任务信息

- 接口（GET）：`http://{LTSMaster}:12311/pro/proc/{procId}/info`，LTSMaster需要修改为Lindorm实例的Master hostname，可以登录Lindorm实例的LTS，在**集群信息**页面的**基本信息**区域获取。
- 参数说明：procId表示任务ID。
- 示例：

```
curl http://{LTSMaster}:12311/pro/proc/proc-91-ff383c616e5242888b398e51359c****/info
```

返回内容如下：

```
{
    "data":{
        "checkJobs":Array,
        "procId":"proc-91-ff383c616e5242888b398e51359c****",  //任务ID
        "incrJobs":Array,
        "procConfig":Object,
        "stage":"WAIT_FOR_SUCCESS",
        "fullJobs":Array,
        "mergeJobs":Array,
        "srcDS":"hdfs",    //源数据源
        "sinkDS":"ld-uf6el41jkba96****",  //目标数据源
        "state":"RUNNING",   //任务状态
        "schemaJob":Object,
        "procType":"SPARK_BULKLOAD"   //任务类型
    },
    "success":"true"
}
```

终止任务

- 接口（GET）：`http://{LTSMaster}:12311/pro/proc/{procId}/abort`，LTSMaster需要修改为Lindorm实例的Master hostname，可以登录Lindorm实例的LTS，在**集群信息**页面的**基本信息**区域获取。
- 参数说明：procId表示任务ID。
- 示例：

```
curl http://{LTSMaster}:12311/pro/proc/proc-91-ff383c616e5242888b398e51359c****/abort
```

返回内容如下：

```
{"success":"true","message":"ok"}
```

重试任务

- 接口（GET）：`http://{LTSMaster}:12311/pro/proc/{procId}/retry`，LTSMaster需要修改为Lindorm实例的Master hostname，可以登录Lindorm实例的LTS，在**集群信息**页面的**基本信息**区域获取。
- 参数说明：procId表示任务ID。
- 示例：

```
curl http://{LTSMaster}:12311/pro/proc/proc-91-ff383c616e5242888b398e51359c****/retry
```

返回结果如下：

```
{"success":"true","message":"ok"}
```

删除任务

- 接口（GET）：`http://{LTSMaster}:12311/pro/proc/{procId}/delete`，LTSMaster需要修改为Lindorm实例的Master hostname，可以登录Lindorm实例的LTS，在**集群信息**页面的**基本信息**区域获取。
- 参数说明：procId表示任务ID。
- 示例：

```
curl http://{LTSMaster}:12311/pro/proc/proc-91-ff383c616e5242888b398e51359c****/delete
```

返回结果如下：

```
{"success":"true","message":"ok"}
```

配置示例

- 读插件配置示例。

- 源数据源为ODPS的读插件配置示例。

```
{
  "table": "test",
  "column": [
    "id",
    "intcol",
    "doublecol",
    "stringcol",
    "string1col",
    "decimalcol"
  ],
  "partition": [
    "pt=1"
  ],
  "numPartitions":10
}
```

- 源数据源为HDFS，CSV文件的读插件配置示例。

```
{
  "filePath":"csv/",
  "header": false,
  "delimiter": ",",
  "column": [
    "id|string",
    "intcol|int",
    "doublecol|double",
    "stringcol|string",
    "string1col|string",
    "decimalcol|decimal"
  ]
}
```

- 源数据源为HDFS，Parquet文件的读插件配置示例。

```
{
  "filePath":"parquet/",
  "column": [   //parquet文件中的列名
    "id",
    "intcol",
    "doublecol",
    "stringcol",
    "string1col",
    "decimalcol"
  ]
}
```

- 写插件配置示例。
  - 导入到Lindorm SQL表格的写插件配置示例。

```
{
  "namespace": "default",
  "lindormTable": "xxx",
  "compression":"zstd",
  "columns": [
       "id",
       "intcol",
       "doublecol",
       "stringcol",
        "string1col",
        "decimalcol"
  ]
}
```

- 导入到Lindorm兼容HBase表格的写插件配置示例。

```
{
  "namespace": "default",
  "lindormTable": "xxx",
  "compression":"zstd",
  "columns": [
    "ROW||String",    //ROW代表rowkey,String表示类型
    "f:intcol||Int",  //列簇:列名||列类型
    "f:doublecol||Double",
    "f:stringcol||String",
    "f:string1col||String",
    "f:decimalcol||Decimal"
  ]
}
```

14.6.2.6. 将Cassandra数据导入至Lindorm宽表

本文介绍使用开源DataX配置同步任务将云数据库Cassandra全量数据导入至云原生多模数据库Lindorm宽表。

前提条件

已创建云原生多模数据库Lindorm实例，创建方法请参见创建实例。

使用限制

使用开源DataX配置同步任务实现数据导入的过程中，如果云数据库Cassandra的表涉及到以下四种对象，那么表的数据无法导入至Lindorm宽表。

- SASI索引（Cassandra的一种二级索引）
- 物化视图
- 轻量级事务
- Cassandra Counter数据类型

准备工作

- 如果开源DataX部署在ECS实例上，您需要确保云原生多模数据库Lindorm实例和ECS实例满足以下条件，以保证网络的连通性。
  - 所在地域相同，并建议所在可用区相同。
  - 网络类型相同。专有网络具有更高的安全性，建议使用专有网络。

    > **说明** 如果网络类型为**专有网络**，需确保使用相同的专有网络ID。

- 将客户端IP地址添加至云原生多模数据库Lindorm实例的白名单中，添加方法请参见设置白名单。
- 数据导入前，需要提前在云原生多模数据库Lindorm宽表中创建Namespace和表。
  - 创建Namespace的方法请参见创建Namespace。
  - 通过CREATE TABLE语法创建Lindorm宽表，更多语法说明请参见CREATE TABLE。

操作步骤

以下操作以开源DataX部署在ECS实例上为例。

1. 执行以下命令下载开源DataX安装包。

```
wget https://datax-opensource.oss-cn-hangzhou.aliyuncs.com/datax.tar.gz
```

2. 解压下载的开源DataX安装包。

```
tar zxvf datax.tar.gz
```

3. 执行以下命令创建同步任务文件，文件名为*JOB.json*。

```
touch JOB.json
```

4. 打开同步任务文件*JOB.json*。

```
vi JOB.json
```

5. 配置同步任务文件，同步任务文件的配置内容如下，根据下表的参数说明配置同步任务。

```
{
  "job": {
    "setting": {
      "speed": {
        "channel": 3
      }
    },
    "content": [
      {
        "reader": {
          "name": "cassandrareader",
          "parameter": {
            "host": "localhost",
            "port": 9042,
            "useSSL": false,
            "keyspace": "test",
            "table": "datax_src",
            "column": [
              "id",
              "name"
            ]
          }
        },
        "writer": {
          "name": "Lindormwriter",
          "parameter": {
            "host": "localhost",
            "port": 9042,
            "useSSL": false,
            "keyspace": "test",
            "table": "datax_dst",
            "column": [
              "id",
              "name"
            ]
          }
        }
      }
    ]
  }
}
```

参数	说明
channel	表示同步任务的并行度，通过设置数值来提高同步任务的运行速度。
host	◦ reader.parameter.host：云数据库Cassandra集群的连接地址。您可以通过云数据库Cassandra控制台获取，获取方法请参见获取连接地址。 ◦ writer.parameter.host：云原生多模数据库Lindorm宽表引擎的Cassandra兼容地址。您可以通过云原生多模数据Lindorm控制台获取，获取方法请参见获取连接地址。
port	端口号固定为9042。 ◦ reader.parameter.port：云数据库Cassandra集群的连接地址端口号。 ◦ writer.parameter.port：云原生多模数据库Lindorm宽表引擎的Cassandra兼容地址端口号。

参数	说明
useSSL	是否开启SSL（Secure Sockets Layer）加密。 ○ true：开启SSL加密。 ○ false：不开启SSL加密。
keyspace	○ reader.parameter.keyspace：云数据库Cassandra中需要导入的Keyspace名称。 ○ writer.parameter.keyspace：导入至Lindorm宽表的Namespace名称。
table	○ reader.parameter.table：云数据库Cassandra中需要导入的Table。 ○ writer.parameter.table：导入至Lindorm宽表的表名。
column	○ reader.parameter.column：云数据库Cassandra中需要导入的Table的列名。 ○ writer.parameter.column：导入至Lindorm宽表中指定表的列名。

6. 执行以下命令进入*bin*目录并运行同步任务文件。

```
cd <DATAX_HOME>/bin
python datax.py <JOB.json>
```

> **说明**
> ○ **DATAX_HOME**：表示开源DataX安装包解压后的存储路径。
> ○ **JOB.json**：表示同步任务文件的名称。

14.6.3. 数据导出

14.6.3.1. 全量导出MaxCompute

本文主要介绍如何在Hbase中进行全量导出MaxCompute操作。

前提条件

- 已购买LTS数据迁移同步服务，配置LTS操作页面账户密码，并登录LTS操作页面，具体操作请参见开通并登录LTS。
- 已添加HBase集群数据源。
- 已添加MaxCompute数据源。

功能列表

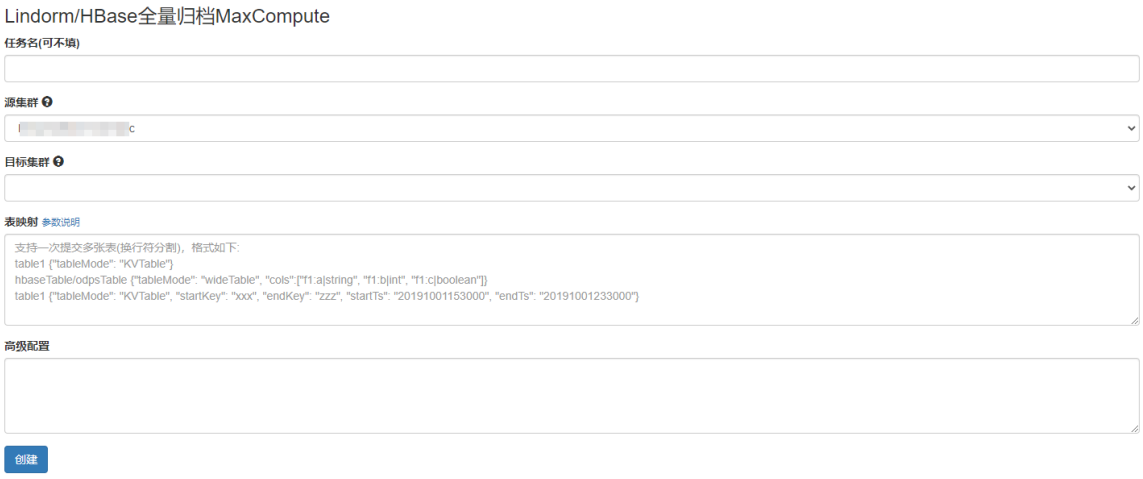
- 导出HBase全量数据到MaxCompute。
- 支持指定Rowkey、时间范围、支持指定对应列。

支持版本

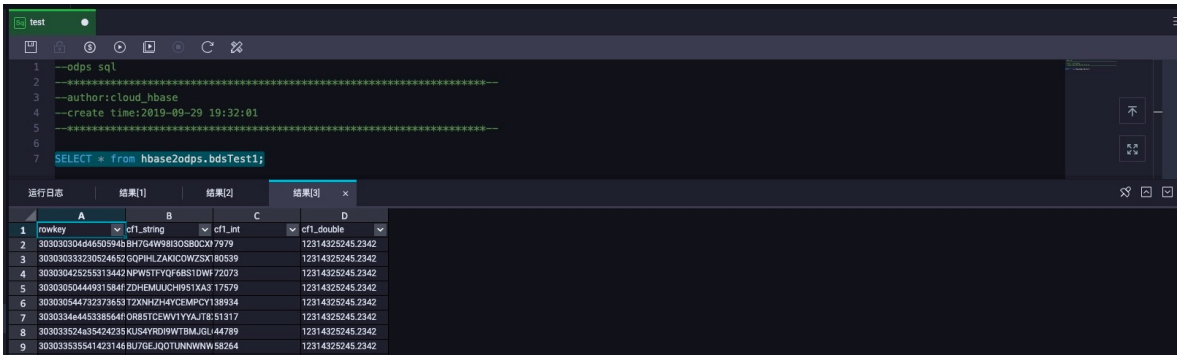
- 自建HBase1.x、2.x。
- EMR HBase。
- 标准版云HBase、增强版云HBase（集群版本）、Lindorm。

创建任务

1. 进入LTS操作页面，在左侧导航栏选择**Lindorm/HBase导出 > 历史数据导出MaxCompute**。
2. 在**Lindorm/HBase全量归档MaxCompute**页面单击**创建任务**。
3. 选择源集群和目标集群，输入需要迁移的表，单击**创建**。



4. 单击任务名，查看迁移进度。
5. 迁移完成之后，登录到MaxCompute便可以查询数据。



参数说明

```
hbaseTable/odpsTable {"tableMode": "wideTable", "cols":["f1:a", "f1:b", "f1:c"]}
hbaseTable/odpsTable {"tableMode": "wideTable", "cols":["f1:a|string", "f1:b|boolean"]}
hbaseTable/odpsTable {"tableMode": "wideTable", "cols":["f1:a|string", "f1:b|boolean"], "startTs": "20190101
000000", "endTs": "20200101000000", "startKey": "a", "endKey": "z"}
hbaseTable/odpsTable {"tableMode": "KVTable"}
```

HBase数据导出到MaxCompute，支持两种模式：KV模式、宽表模式。

KV模式

入参： `tablexxx {"tableMode": "KVTable"}`

结果输出：

	A	B	C	D	E	F
1	rowkey	version	op	family	qualifier	val
2	3737375a4c424f5033	1569404372597	Put	cf1	bigdecimal	000000040ddb105bd3
3	3737375a4c424f5033	1569404372597	Put	cf1	boolean	ff
4	3737375a4c424f5033	1569404372597	Put	cf1	double	4206efed87e9dfa4
5	3737375a4c424f5033	1569404372597	Put	cf1	float	4499e77f
6	3737375a4c424f5033	1569404372597	Put	cf1	int	0000641a
7	3737375a4c424f5033	1569404372597	Put	cf1	long	00000002dfdc1c3e
8	3737375a4c424f5033	1569404372597	Put	cf1	null	
9	3737375a4c424f5033	1569404372597	Put	cf1	short	0001
10	3737375a4c424f5033	1569404372597	Put	cf1	string	4b43524b5039315241
11	37373836555a52474f	1569404344057	Put	cf1	bigdecimal	000000040ddb105bd3
12	37373836555a52474f	1569404344057	Put	cf1	boolean	ff

> 说明
- 每一行代表HBase中的一个KV。
- rowkey：HBase表KV的rowkey。
- version：KV的版本。
- op：KV的类型（Put、DeleteFamily、DeleteColumn）。
- family：列族。
- qualifier：列名。
- value：KV的具体value（转化成HexString的格式）。

宽表模式

入参：

```
hbaseTable/odpsTable {"tableMode": "wideTable", "cols":["cf1:string|string", "cf1:int|int",
"cf1:double|double"]}
```

> 说明
宽表模式需要指定导出的列，例如 “cols” :[“f1:a” ， “f1:b” ， “f1:c”], 列值默认是转化成HexString的格式。可以指定列的类型，例如” cols” :[“f1:a|string” ， “f1:b|boolean” ， “f1:c|int”]。列支持的类型：string、int、long、short、decimal、double、float、boolean，默认类型 hex, 将bytes数组转化为HexString。

结果输出：

	A	B	C	D
1	rowkey	cf1_string	cf1_int	cf1_double
2	303030304d4650594b	BH7G4W98I3OSB0CX	7979	12314325245.2342
3	3030303332305246 52	GQPIHLZAKICOWZSX	80539	12314325245.2342
4	3030304252553134 42	NPW5TFYQF6BS1DWF	72073	12314325245.2342
5	3030305044493158 4f	ZDHEMUUCHI951XA3	17579	12314325245.2342
6	3030305447323736 53	T2XNHZH4YCEMPCY1	38934	12314325245.2342
7	3030334e445338564f	OR85TCEWV1YYAJT8	51317	12314325245.2342
8	303033524a354242 35	KUS4YRDI9WTBMJGL	44789	12314325245.2342
9	3030335355414231 46	BU7GEJQOTUNNWNW	58264	12314325245.2342
10	3030344630 4f495134	69PKNZ07IOY0KM5G	54219	12314325245.2342
11	303035414e554336 52	09U6DML9TUWXEKQ3	39945	12314325245.2342
12	3030354932303353 56	HY93A9Y0XQUSIIPT4	9360	12314325245.2342

指定导出数据的范围

入参： `table1 {"startKey": "xxx", "endKey": "zzz", "startTs": "20191001153000", "endTs": "20191001233000"}`

HBase导出MaxCompute不管是KV模式还是宽表模式，导出支持使用startKey、endKey来指定主键区间，以及使用startTs、endTs参数来指定时间区间。

指定导出到不同MaxCompute表

HBase导出MaxCompute支持HBase表和MaxCompute表不同表名，默认情况和HBase表名相同（hbase表名中的’-‘、’.’ 会被替换成’_’）例如： `hbaseTable/odpsTable {"tableMode": "KVTable"}` 。

14.6.3.2. 增量归档MaxCompute

本文主要介绍如何在Hbase中进行增量归档MaxCompute。

前提条件

1. 已开通LTS。
2. 已添加HBase数据源。
3. 已添加MaxCompute数据源。

支持版本

- 自建HBase1.x、2.x。
- EMR HBase。
- 标准版云HBase、增强版云HBase（集群版本）、Lindorm。

限制

- 实时数据归档基于HBase的日志，因此不能导出bulkload的数据。

日志生命周期

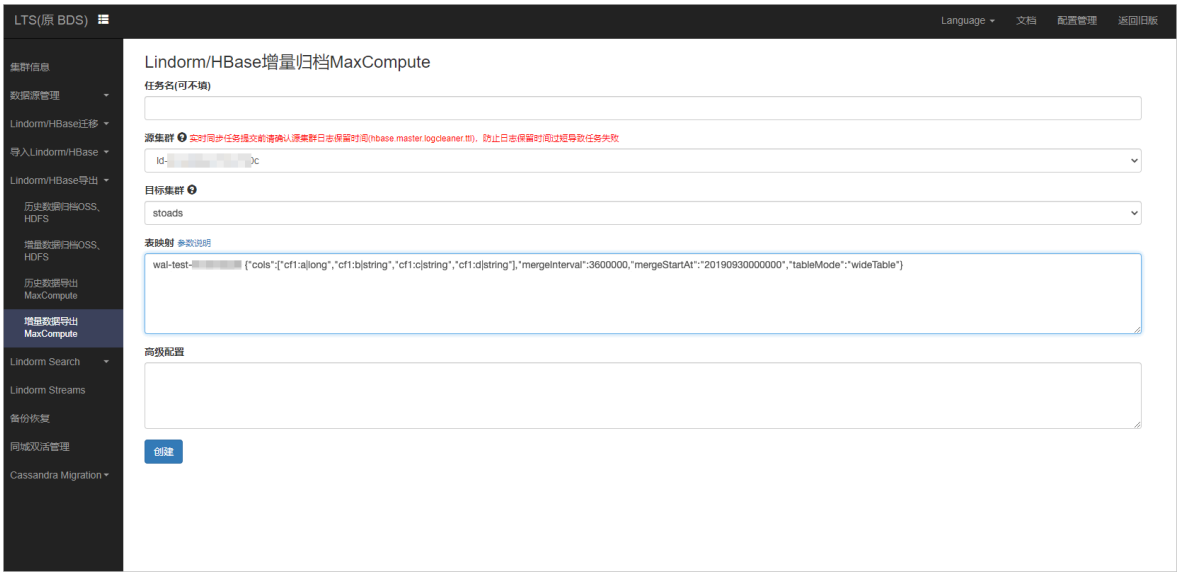
- 开启归档后，如果不消费数据，默认日志会保留48小时，超时后订阅关系自动取消，保留的数据自动删除
- 什么场景会导致“不消费数据”：没有终止任务的前提下直接释放LTS集群；同步任务暂停；

提交归档任务

1. 进入LTS操作页面，在左侧导航栏选择**Lindorm/HBase导出 > 增量数据导出MaxCompute**。



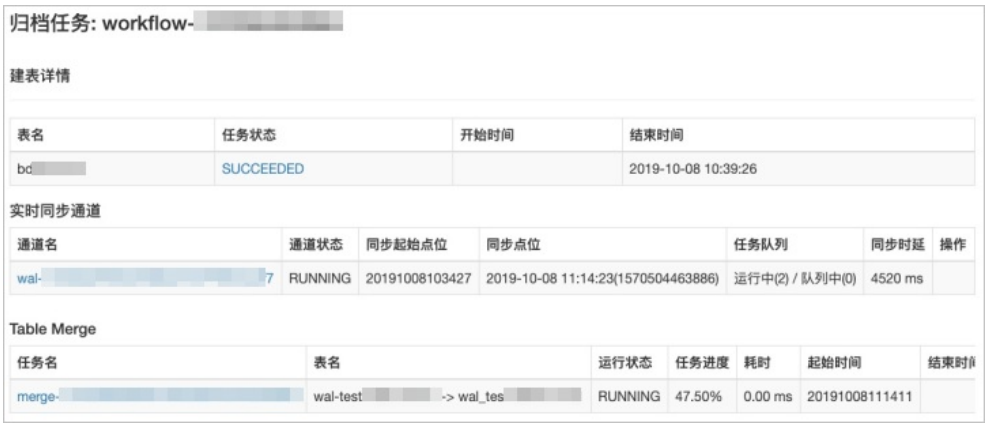
2. 单击**创建任务**， 进入创建页面，选择对应的HBase源集群和MaxCompute目标集群， 指定需要导出的HBase表。



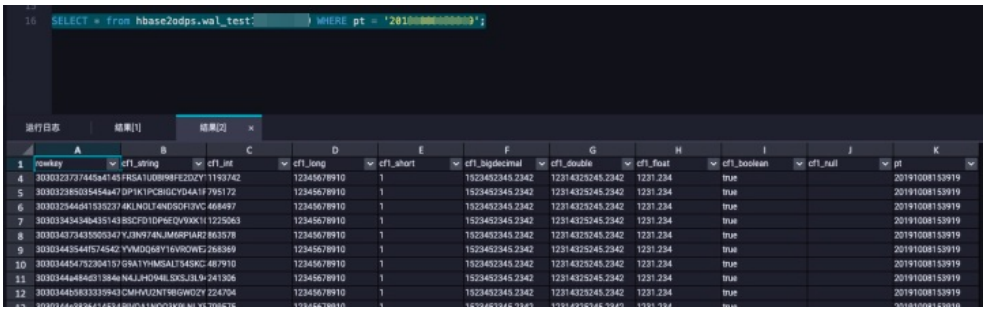
上图对HBase表wal-test进行实时数据归档到MaxCompute

- 归档的列包括cf1:a、cf1:b、cf1:c、cf1:d四列。
- mergeInterval 表示的是归档的间隔（单位毫秒）， 默认指定 86400000 （一天）。
- mergeStartAt（格式为yyyyMMddHHmmss）表示从2019年9月30号0点整作为起始时间（可以指定一个过去的值） 。

3. 查看表归档进度。实时同步通道展示的是日志同步的延迟、点位，Table Merge展现的是表Merge的任务，Merge完成即可在MaxCompute上查询最新分区的表。



4. 登录MaxCompute查询表数据。



参数说明

导出表格式如下：

```
hbaseTable/odpsTable {"cols": ["cf1:a|string", "cf1:b|int", "cf1:c|long", "cf1:d|short","cf1:e|decimal", "cf1:f|double","cf1:g|float","cf1:h|boolean","cf1:i"], "mergeInterval": 86400000, "mergeStartAt": "20191008100547"}
hbaseTable/odpsTable {"cols": ["cf1:a", "cf1:b", "cf1:c"],  "mergeStartAt": "20191008000000"}
hbaseTable {"mergeEnabled": false} // 不进行merge操作
```

导出的表包含三部分：{{hbaseTable}}，{{odpsTable}}，{{tbConf}}。{{hbaseTable}}：指定HBase表。{{odpsTable}}：可选，指定目标表名。默认情况和HBase相同 （ODPS表名不支持包含“.-“等字符，统一转化成“_”）。{{tbConf}}：指定表的归档行为。{{tbConf}}支持的参数如下表：

参数	说明	例子
cols	指定需要导出的列及列类型。默认转化格式为HexString。	“cols”：[“cf1:a”，“cf1:b”，“cf1:c”]
mergeEnabled	是否需要进行KV表转化成宽表， 默认为true。	“mergeEnabled”：false
mergeStartAt	merge的起始时间，可以指定一个过去的时间，格式是yyyyMMddHHmmss。	“mergeStartAt”：“20191008000000”
mergeInterval	merge的时间间隔，单位毫秒，默认是一天，即按天归档。	“mergeInterval”：86400000

14.6.4. 最佳实践

14.6.4.1. 低成本历史库

背景

在移动互联网高度发达的今天，每天都会有大量的业务数据产生，随着时间的积累和业务的发展数据量成指数级增长，同时历史数据的访问频率随着时间的推移却越来越低。这些数据全部存储在关系型数据库中会带来了一系列的问题。

挑战

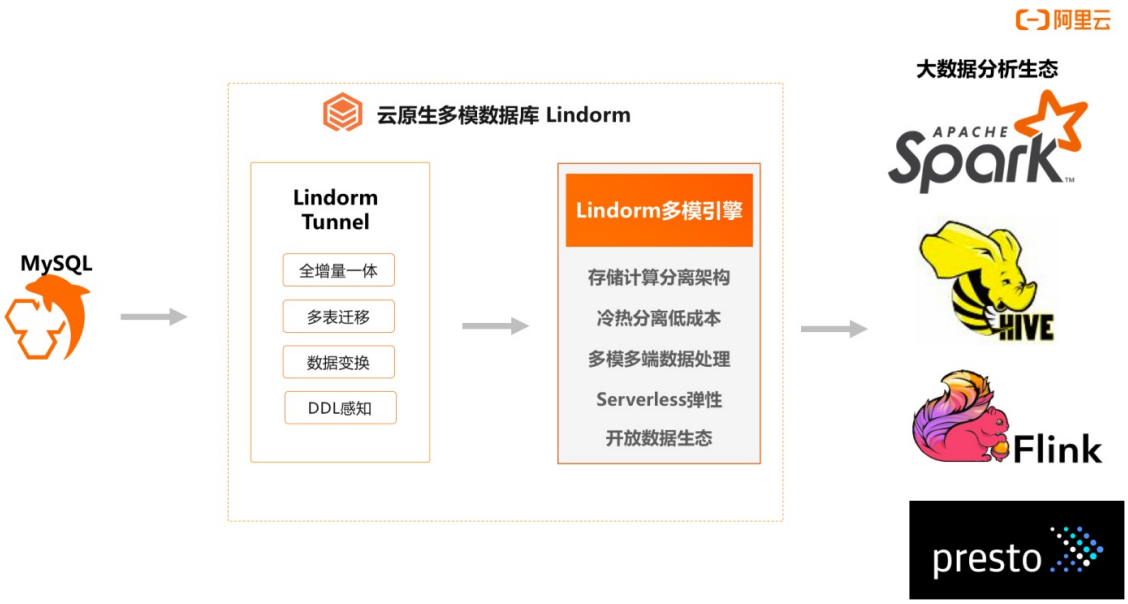
- **存储成本的升高**，存储成本和数据量成正比，数据量指数级增长带来了存储成本的指数级增加。
- **查询性能的下降**，单实例存储空间超过TB以后查询性能将会急剧下滑。
- **运维复杂度高**，通过分库分表的方式来解决数据量增加造成性能下降，给运维和业务开发增加了巨大的成本。

需求

- **存储成本可控**，历史数据的存储成本是在线数据的10分之一。
- **能够弹性的伸缩**， 计算和存储能力自动水平扩展，不再依赖分库分表解决运维难的痛点。
- **修改schema成本低**， 支持快速修改schema或者动态schema，解决历史库schema变更时间长的问题。
- **改造成本低**， 可以支持SQL进行访问。
- **满足实时查询需求**，例如消费账单、聊天记录等场景历史数据的查询RT要能够和在线数据的查询RT接近。
- **能够满足数据分析需求**，历史数据访问频度低，但是在某些业务场景下需要对全量数据进行挖掘分析，比如支付宝年账单。

云原生多模数据库Lindorm能够同时满足存储成本低廉、运维简单、弹性伸缩、性能稳定等需求可以和关系型数据库完美组合，是实时低成本历史库的最佳选择。

方案架构



- LTS数据同步服务对接MySQL等关系行数据库支持数据的全增量一体化同步，同时提供多表迁移、数据变化、DDL感知等企业级同步能力，帮助用户简单高效完成数据的迁移；
- 云原生多模数据库Lindorm提供海量数据的低成本存储能力（0.11元/GB/月）、弹性伸缩按需付费、多模多端处理能力满足用户多种场景下的数据存储需求，同时可以无缝对接spark、hive、flink、presto等开源分析生态满足用户对数据的复杂分析需求，最大程度挖掘数据价值；

方案优势

简单易用

- 数据搬迁配置可视化，分钟级完成迁移配置。
- 全增量一体化，最大程度降低用户使用成本。
- 支持多表迁移、数据变换等能力帮助用户轻松完成多表合并、字段组合变化等需求。
- 全面的监控报警，保证数据同步稳定性。

极致性价比

- 支持容量型**存储成本底至0.11元/GB/月**，在极致优化存储成本的同时通过系统内置缓冲加速层，让查询实时性仍有较大的保障，是实时历史库的最佳存储选择。
- 在性能上，Lindorm宽表引擎在吞吐延迟上做了非常多的突破，其**基准性能是开源HBase的7倍，**具体请参考测试结果；Lindorm时序引擎融入了许多创新型的高性能结构设计，其基准性能在目前的**信通院榜单中处于第一**的位置，大幅领先于其他专用时序数据库。
- **支持智能冷热分离**，针对数据随着时间线逐渐热变冷的场景，典型如监控、社交聊天、交易账单等，Lindorm内部将自动识别数据的冷热，并进行分离存储到高性能介质和低成本介质（两者之间的单价成本差可高达10:1），而用户读写访问保持完全透明，并且热数据的访问性能还能有所加速。
- 支持自适应压缩，针对数据的不同类型和特点，系统将自动选择混合的字典、前缀、Delta、熵编码等压缩算法，相比业界通用算法，**整体压缩率提升10%~30%。**

云原生弹性

- Lindorm基于**存储计算分离**的架构，支持计算资源、存储资源的独立弹性伸缩，最大程度避免资源浪费；
- **同时提供Serverless服务，实现按需即时弹性、按使用量付费的能力。**Lindorm Serverless基于多租户隔离、智能调度、弹性IaaS底座构建，具备企业级SLA保障，满足内部大部分业务的可用性要求，从而让一线同学大幅降低容量管理的运维负担，消除因流量波动导致的稳定性风险。

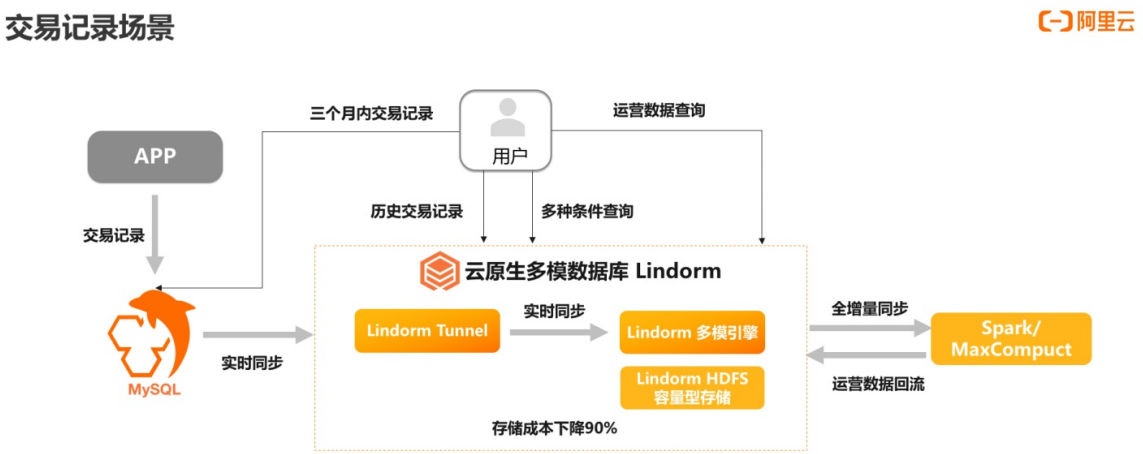
多模&检索

- **兼容HBase、Phoenix（SQL）、Cassandra（CQL）**等主流开源标准接口，最大程度降低用户改造成本;同时具备全局二级索引、多维检索、动态列、TTL等能力， 满足元数据、订单、账单、画像、社交、feed流、日志等多种场景需求。
- **支持一键开启搜索引擎**（兼容开源Solr标准接口）， 具备全文检索、聚合计算、复杂多维查询等能力，帮助加速检索查询，满足用户实时复杂分析需求。

大数据生态

- 无缝对接Spark、Hive、Flink、Presto等开源大数据生态产品，支持API访问和文件读取等多种对接方式，简单高效满足用户海量数据分析需求。

典型应用



- 用户交易记录通过APP写入MySQL，LTS将MySQL中实时同步到Lindorm，近三个月状态不断变化的记录查询MySQL，三个月以上历史交易记录查询Lindorm，历史交易记录存储在容量型存储中，存储成本下降90%以上。
- 对于用户复杂条件实时检索需求，例如按时间、地点、金额大小、交易备注内容等任意组合查询，可以结合Lindorm搜索引擎的全文检索、聚合计算、复杂多维查询等能力，不需要业务改造就能轻松满足用户需求。
- 通过LTS支持将Lindorm账单数据同步到Spark/MapCompuct等离线计算平台进行计算按照业务需求分析生成运营报表数据，然后回流到Lindorm供用户进行实时查询。

使用说明

- RDS全增量同步

- 存的起，看得见—云原生多模数据库Lindorm技术解析

14.7. 时序引擎数据迁移

14.7.1. 利用DataWorks将数据迁移到Lindorm时序引擎

背景

本文介绍通过DataWorks的数据集成功能，实现OpenTSDB到TSDB的数据迁移。DataWorks是阿里云数加重要的PaaS平台产品，提供数据集成、数据开发、数据服务、数据分析和数据治理等全方位的产品服务，一站式开发管理的界面，帮助企业专注于数据价值的挖掘和探索。本文档介绍的数据迁移主要涉及”数据开发”部分。不熟悉DataWorks的用户可以参阅DataWorks的产品文档，做进一步地了解。等全方位的产品服务，一站式开发管理的界面，帮助企业专注于数据价值的挖掘和探索。本文介绍的数据迁移中主要涉及数据开发的部分。

Quick Start

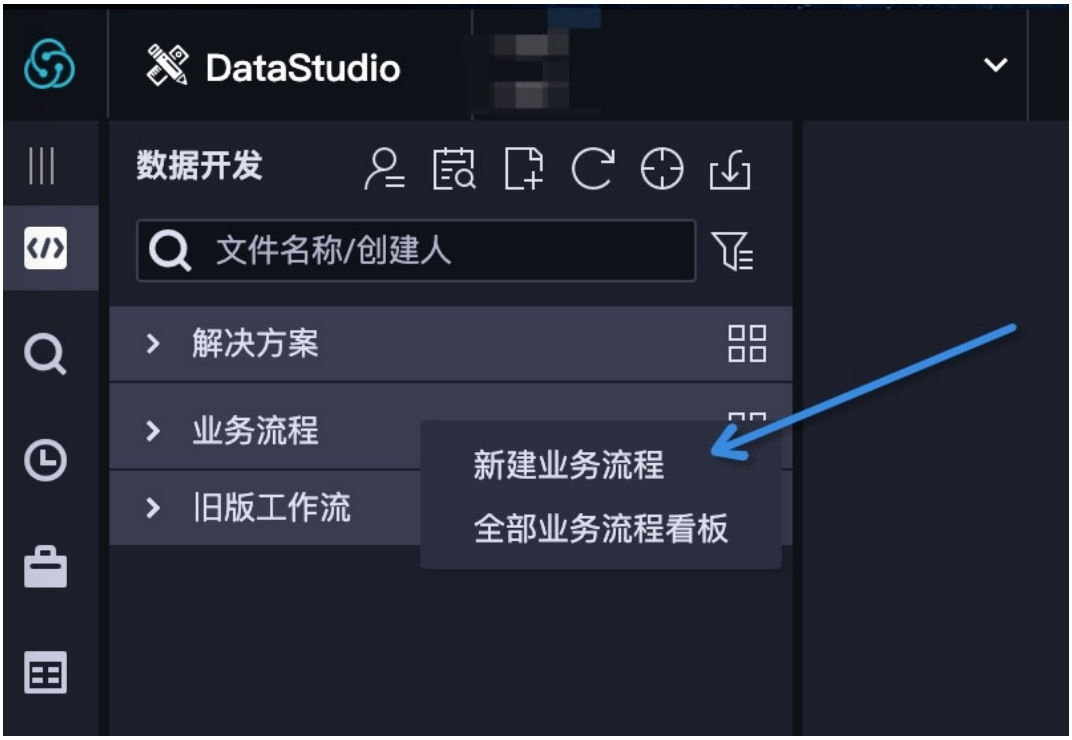
1. 入口
   首先，通过单击链接，我们可以访问DataWorks的管理控制台。如果您尚未在控制台创建工作空间，需要先按照《创建工作空间》文档创建好一个工作空间。创建完成之后，便可以在管理控制台看到工作空间列表。实际效果如下图所示：

   工作空间列表



2. 在”数据开发”中创建”同步节点”
   右击左上角”业务流程”按钮，再点击”新建业务流程”。实际效果如下图所示：

   新建业务流程



在弹窗中填入业务名称（例如：migration_from_opentsdb_to_tsdb）。实际效果如下图所示：

业务名称



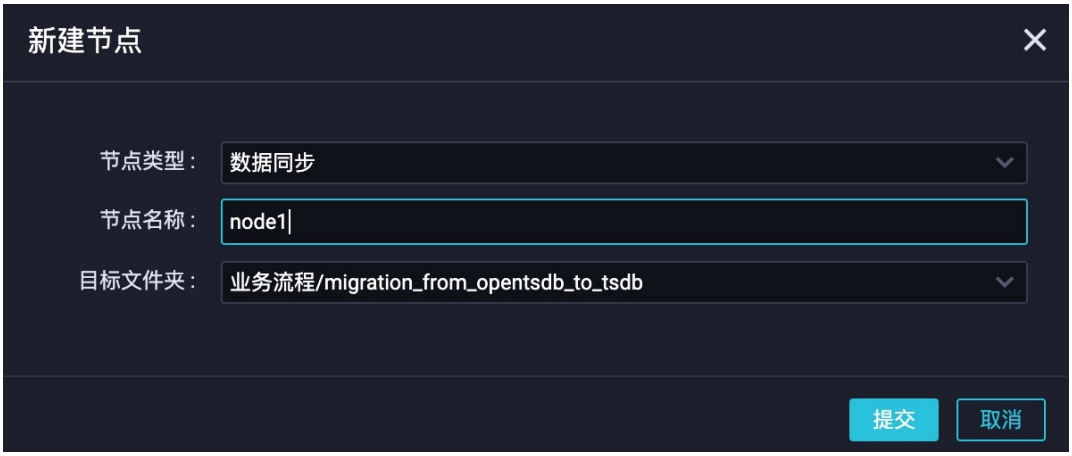
按照图 4 中的三个步骤，完成数据同步任务的创建：

创建数据同步任务



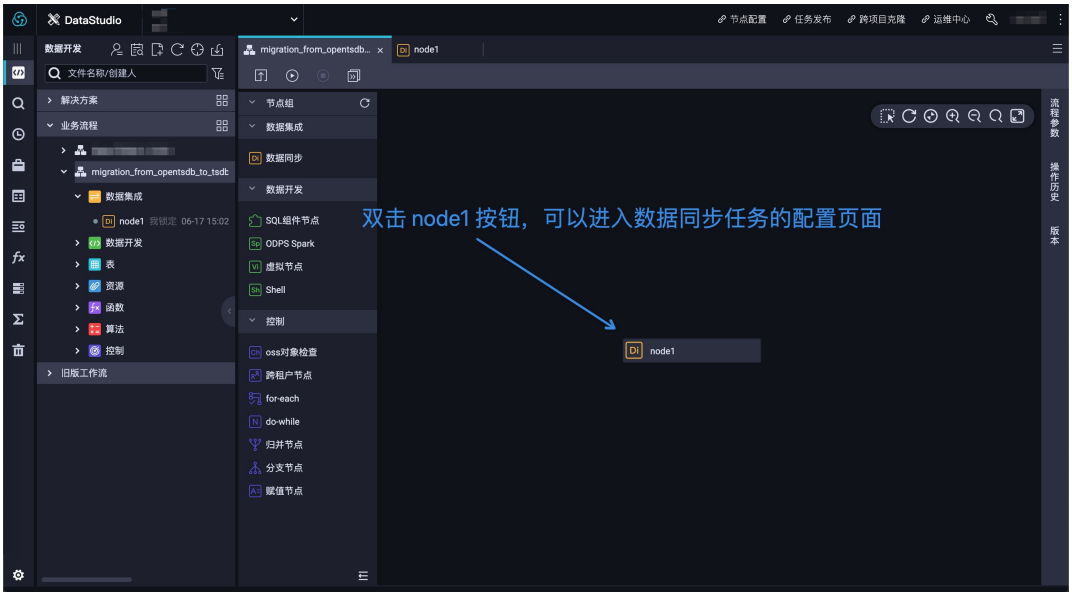
在弹窗中填入数据同步任务名（例如：node1）。实际效果如下图所示：

数据同步任务名



数据同步任务创建成功之后，可以在右边空白处看到 node1 节点。通过双击 node1 按钮，可以进入到数据同步任务的配置页面。实际效果如下图所示：

进入数据同步任务的配置页面



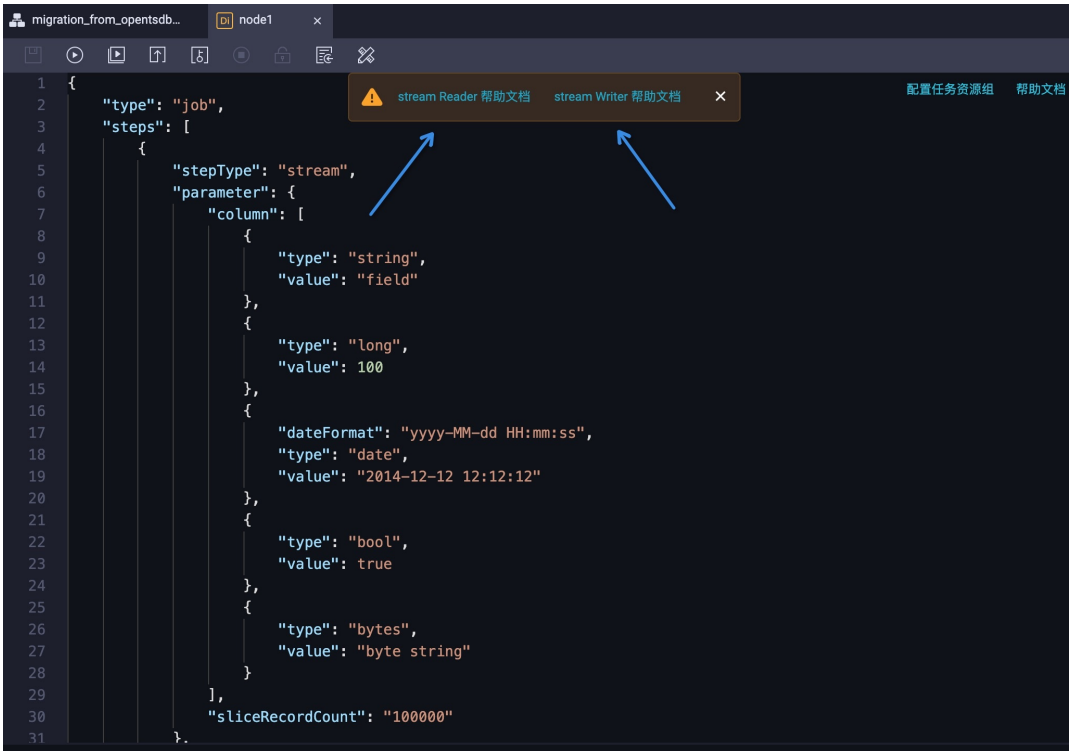
数据同步任务 node1 的默认编辑模式是”向导模式”，我们可以通过点击最右边的按钮，直接进入到脚本编辑模式。实际效果如下图所示：

脚本编辑模式



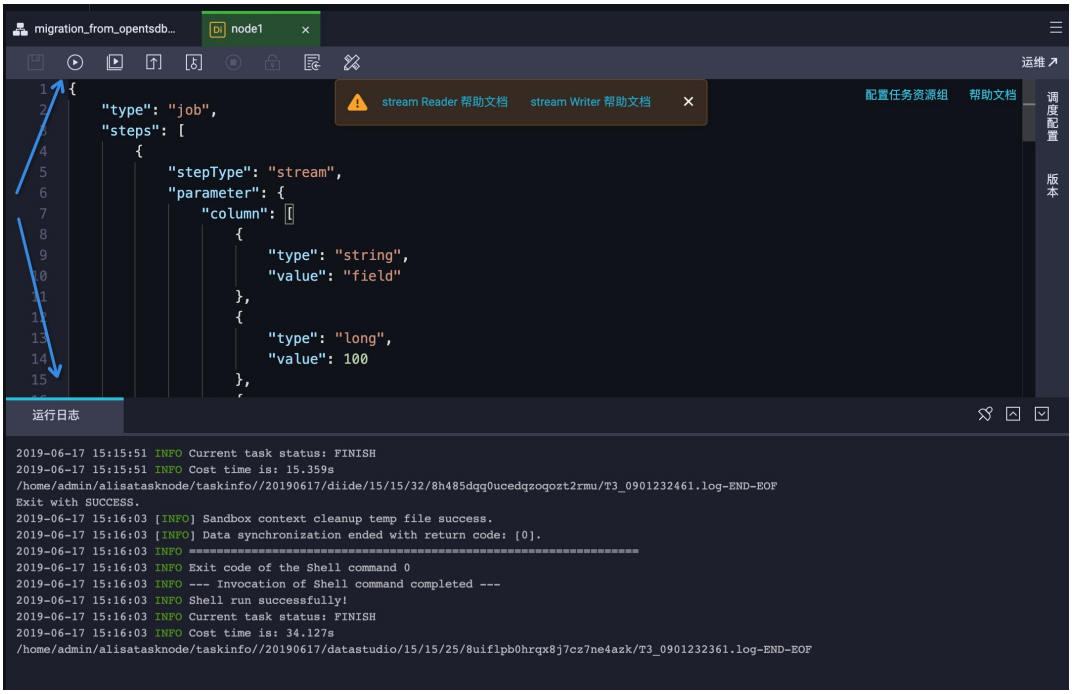
默认的同步任务是 Stream Reader 到 Stream Writer。大致流程是 Stream Reader 作为数据源，创建随机的字符串，而 Stream Writer 作为目标源，接受这些生成的字符串，并打印。更多相关的信息详见页面上方的帮助文档。具体效果如图下图所示：

默认同步任务



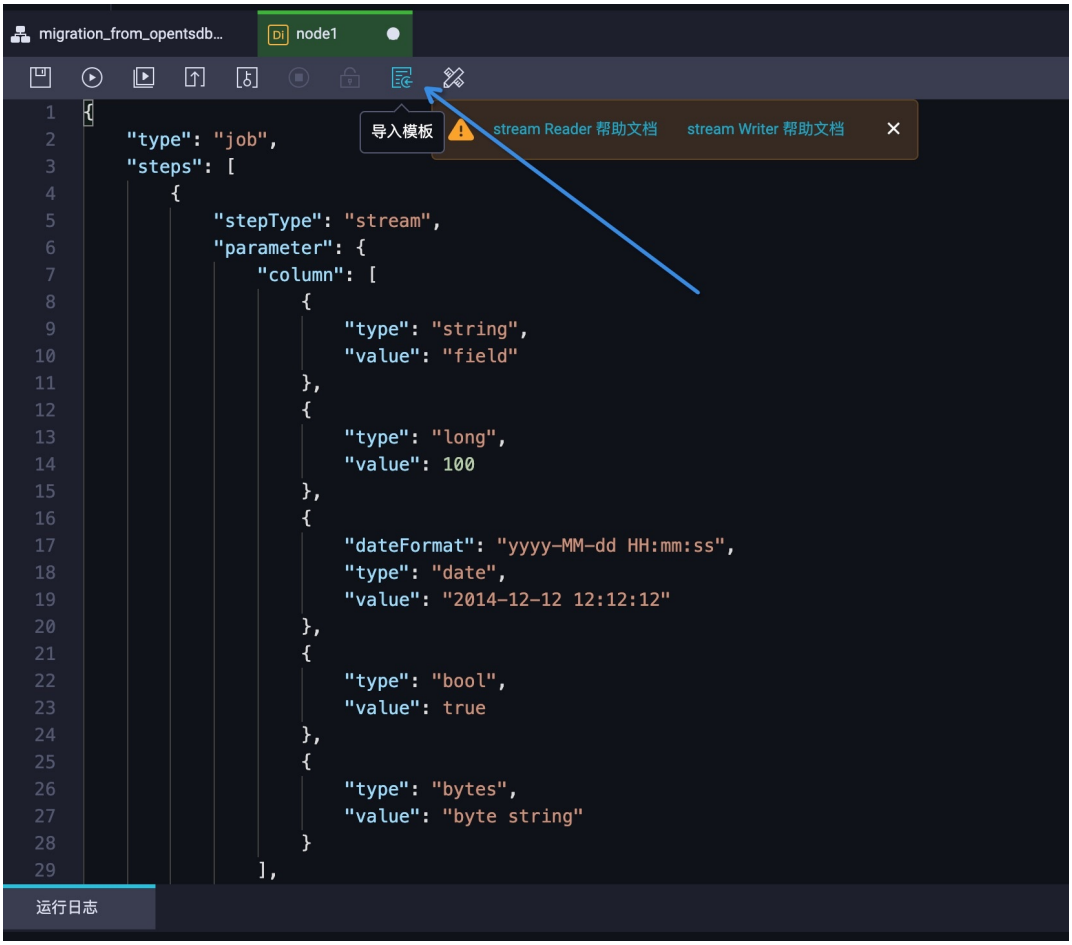
因为 Stream Reader 和 Stream Writer 同步任务不需要依赖任何外部资源，所以可以直接点击左上角的执行按钮，直观地查看任务的执行过程。实际效果如下图所示：

运行默认的同步任务



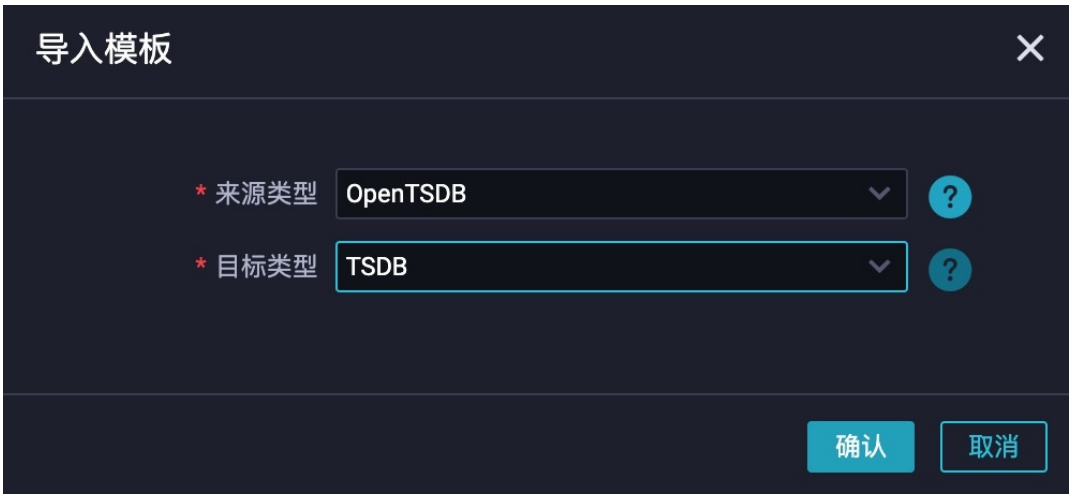
3. 修改配置
   现在，我们将默认的同步任务，修改为 OpenTSDB 到 TSDB 的同步任务。
   首先，我们通过点击下图中箭头所指按钮，开始导入配置模板。实际效果如下图所示：

   导入模板



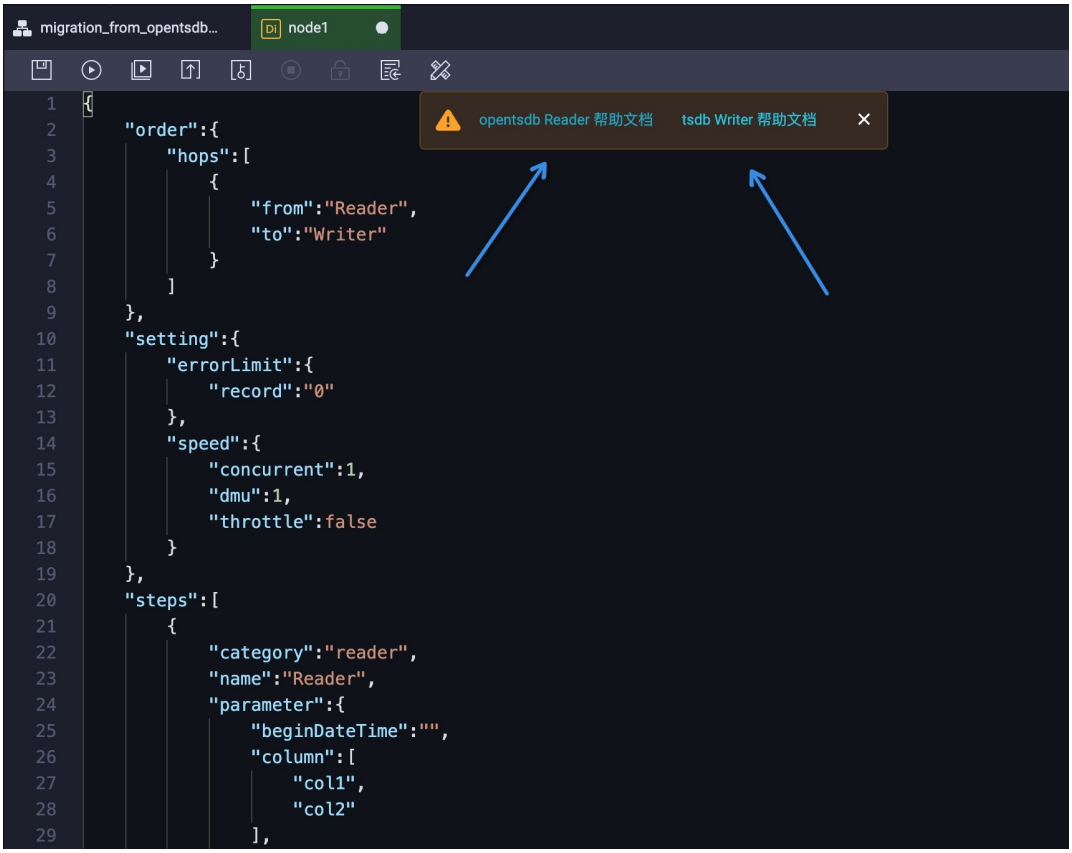
在弹窗中配置来源类型为 OpenTSDB，目标类型为 TSDB。实际效果如下图所示：

配置导入模板



单击确认之后，可以看到stepType已经调整为opentsdb和tsdb，而整个配置都已经自动调整为对应的配置项。效果如下图所示：

导入配置模板后的效果



此时，按照配置文档进行配置的调整。其中必填的参数只有 5 个，包括 OpenTSDB 的地址（endpoint）、需要同步的 metric（column）和同步的时间范围（beginDateTime & endDateTime），以及 TSDB 的地址（endpoint）。例如：

```
{
    "type": "job",
    "steps": [
        {
            "stepType": "opentsdb",
            "parameter": {
                "endpoint": "http://host:4242",
                "column": [
                    "m"
                ],
                "beginDateTime": "20190101000000",
                "endDateTime": "20190101030000"
            },
            "name": "Reader",
            "category": "reader"
        },
        {
            "stepType": "tsdb",
            "parameter": {
                "endpoint": "http://host:8242"
            },
            "name": "Writer",
            "category": "writer"
        }
    ],
    "version": "2.0",
    "order": {
        "hops": [
            {
                "from": "Reader",
                "to": "Writer"
            }
        ]
    },
    "setting": {
        "errorLimit": {
            "record": "0"
        },
        "speed": {
            "throttle": false,
            "concurrent": 1,
            "dmu": 1
        }
    }
}
```

4. 修改白名单
   使用 DataWorks 默认的资源组需要将 Region 对应的网段，增加到白名单中。以 OpenTSDB 到 TSDB 的数据迁移为例，需要给 OpenTSDB 和 TSDB 分别配置白名单。

i. 参考《DataWorks V2.0 > 数据集成 > 网络连通 > 添加白名单 》文档，依据 DataWorks 工作区所在的 Region，找到需要配置的白名单网段。这里我们以华东 2 为例：

不同的Region对应不同的DataWorks白名单



3. 根据项目所在的Region选择相应的白名单。
目前一部分数据源有白名单的限制，需要对数据集成的访问IP进行放行，比较常见的数据源如RDS、MongoDB和Redis等，需要在相应的控制台对这边的IP进行开放。一般添加白名单有以下两种情况：
- 同步任务运行在自定资源组上，要给自定资源组机器授权，将自定义机器内/外网IP添加数据源的白名单列表。
- 同步任务运行在默认资源组上，需要给底层运行机器授予访问权限，根据您选择DataWorks的Region来填写您需要添加的白名单，内容如下表所示。

region	白名单
华东1（杭州）	100.64.0.0/8,11.193.102.0/24,11.193.215.0/24,11.194.110.0/24,11.194.73.0/24,118.31.157.0/24,47.97.53.0/24,11.196.23.0/24,47.99.12.0/24,47.99.13.0/24,114.55.197.0/24,11.197.246.0/24,11.197.247.0/24
华东2（上海）	11.193.109.0/24,11.193.252.0/24,47.101.107.0/24,47.100.129.0/24,106.15.14.0/24,10.117.28.203,10.117.39.238,10.143.32.0/24,10.152.69.0/24,10.153.136.0/24,10.27.63.15,10.27.63.38,10.27.63.41,10.27.63.60,10.46.64.81,10.46.67.156,11.192.97.0/24,11.192.98.0/24,11.193.102.0/24,11.218.89.0/24,11.218.96.0/24,11.219.217.0/24,11.219.218.0/24,11.219.219.0/24,11.219.233.0/24,11.219.234.0/24,118.178.142.154,118.178.56.228,118.178.59.233,118.178.84.74,120.27.160.26,120.27.160.81,121.43.110.160,121.43.112.137,100.64.0.0/8
华南1（深圳）	100.106.46.0/24,100.106.49.0/24,10.152.27.0/24,10.152.28.0/24,11.192.91.0/24,11.192.96.0/24,11.193.103.0/24,100.64.0.0/8,120.76.104.0/24,120.76.91.0/24,120.78.45.0/24
香港	10.152.162.0/24,11.192.196.0/24,11.193.11.0/24,100.64.0.0/8,11.192.196.0/24,47.89.61.0/24,47.91.171.0/24,11.193.118.0/24,47.75.228.0/24
	100.106.10.0/24,100.106.35.0/24,10.151.234.0/24,10.

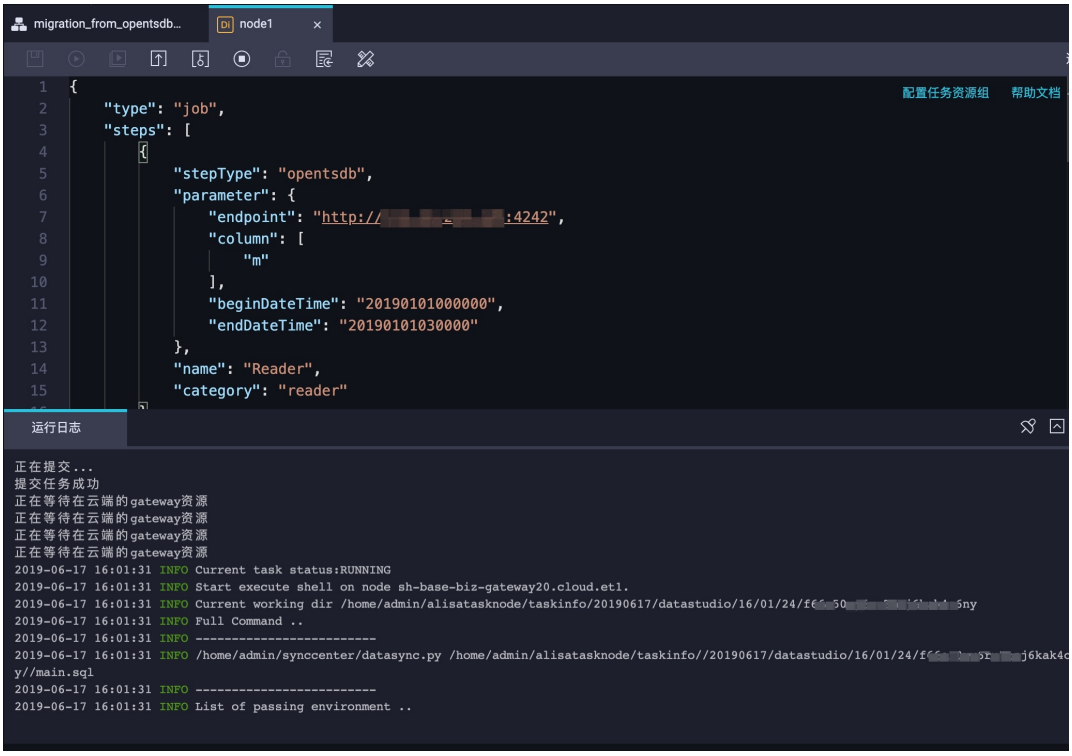
相关文档 · 添加安全组　　相关产品 DataWorks V2.0 MaxCompute

ii. 针对 ECS 上自建的 OpenTSDB 实例，请参考《ECS 安全组配置案例》和《VPC 常见问题》文档，将对应的网段配置到所有 OpenTSDB 实例所在的 ECS 的白名单列表中（包含底层存储 HBase 节点和引擎 TSD 节点）；

iii. 最后，参考设置白名单，将对应的网段配置到云上 TSDB 实例的白名单列表中即可。

5. 数据同步
点击运行按钮，即可启动数据同步任务。实际效果如下图所示：

运行 OpenTSDB 到 TSDB 的数据同步任务



6. 独享资源组
   默认使用 DataWorks 的公共资源组，可能会存在资源抢占的情况，所以并不会保证数据迁移任务的高性能。如果您对数据迁移性能要求很高，建议配置独享资源组。可以参考《DataWorks 独享资源》文档了解该特性的细节，同时，还可以参考《独享资源模式》文档完成购买和使用。

14.7.2. 从关系型数据库到Lindorm时序引擎的数据迁移

本文主要介绍如何使用阿里巴巴的开源工具DataX实现从关系型数据库到时序引擎的数据迁移。

背景信息

DataX是阿里巴巴集团内被广泛使用的离线数据同步工具，实现包括MySQL、Oracle、SQL Server、PostgreSQL、HDFS、Hive、ADS、HBase、TableStore（OTS）、MaxCompute（ODPS）、DRDS等各种异构数据源之间高效的数据同步功能。DataX相关使用介绍请参阅DataX的README文档。
下面将首先介绍DataX工具本身，以及以MySQL为例，实现从MySQL迁移到Lindorm时序引擎。本次迁移工作涉及到的两个插件（MySQL Reader和TSDB Writer）。

MySQL Reader

MySQL Reader是DataX的一个插件，实现了从MySQL读取数据。

TSDB Writer

TSDB Writer是DataX的一个插件，实现了将数据点写入到Lindorm数据库时序引擎中。

注意事项

确保与时序引擎的网络是连通的

因为TSDB Writer写入数据的方式是调用HTTP接口（ `/api/mput` 以及 `/api/put` ）来完成的，所以需要确保迁移任务的进程能正常访问到时序引擎暴露出来的HTTP接口的。否则，会报错Connect Exception异常。

确保与MySQL的网络是连通的

因为MySQL Reader读取数据是通过JDBC来完成的，所以需要确保迁移任务的进程能正常访问到MySQL暴露出来的JDBC接口的。否则，会报错Connect Exception异常。

准备工作

1. 环境准备。
   - Linux
   - JDK（1.8及以上）

- Python（推荐Python 2.6.x）
- MySQL（目前只支持兼容5.x，其他版本暂不兼容）
- Lindorm时序引擎

2. 下载DataX及其插件。
   单击链接进行下载。
3. 利用DataX自带的测试迁移任务，检查迁移流程能否走通。
   这里，我们先以最简单的Stream Reader到Stream Writer为例。因为这两个插件不依赖任何的外部环境，特别适合用来测试流程是否能走通。简单介绍下这两个插件，其中Stream Reader会随机地产生字符串，而Stream Writer则会将接受到的字符串输出到控制台，以此模拟一个最简单的数据迁移过程。

工具部署

将下载后的安装包，解压至某个目录（作为DATAX_HOME），即可运行测试迁移任务：

```
$ cd ${DATAX_HOME}
$ python bin/datax.py job/job.json
```

检查任务是否成功

下面是任务结束后的汇总信息，看到如下内容，即可表示迁移任务已成功完成：

```
任务启动时刻                    : 2019-04-26 11:18:07
任务结束时刻                    : 2019-04-26 11:18:17
任务总计耗时                    :                 10s
任务平均流量                    :          253.91KB/s
记录写入速度                    :          10000rec/s
读出记录总数                    :              100000
读写失败总数                    :                   0
```

配置和启动MySQL到时序引擎的迁移任务

通过上述Stream Reader到Stream Writer的迁移任务，我们就可以确保整个DataX流程是没有问题的。下面就可以真正开始MySQL Reader到TSDB Writer迁移任务了。

参数说明

接下来，我们先看下各个配置项的含义：

MySQL Reader相关

名称	类型	是否必选	描述	默认值	举例
jdbcUrl	String	是	数据库的JDBC连接信息	无	`jdbc:mysql://127.0.0.1:3306/datax`
username	String	是	数据源的用户名	无	`root`
password	String	是	数据源指定用户名的密码	无	`root`
table	String	是	需要同步的表	无	`book`
column	Array	是	表中需要同步的列名集合	`[]`	`["m"]`
splitPk	String	否	如果指定splitPk，表示用户希望使用splitPk代表的字段进行数据分片	空	`id`

TSDB Writer相关

名称	类型	是否必选	描述	默认值	举例
sourceDbType	String	否	目前支持TSDB和RDB两个取值。其中，时序引擎指OpenTSDB、InfluxDB、Prometheus、TimeScale等；而RDB指MySQL、Oracle、PostgreSQL、DRDS 等	`TSDB`	`RDB`
column	Array	是	表中需要同步的列名集合	`[]`	`["name","type","create_time","price"]`
columnType	Array	是	关系型数据库中表字段，映射到时序引擎中的类型。支持的类型：**timestamp**：该字段是个时间戳；**tag**：该字段是个tag；**metric_num**：该metric的value 数值类型；**metric_string**：该metric的value是字符串类型	`[]`	`["tag","tag","timestamp","metric_num"]`
endpoint	String	是	时序引擎的IP:PORT地址	无	`ld-xxxx:8242`
multiField	Bool	是	使用多值方式写入，推荐true	false	true
table	String	是	导入TSDB的表名	无	table
username	String	否	数据库用户名	无	tsdbuser
password	String	否	数据库密码	无	tsdbpsw
ignoreWriteError	Bool	是	忽略写入错误	false	true
database	String	否	导入TSDB的数据库名	default	default

配置迁移任务

配置一个从MySQL数据库同步抽取数据到时序引擎的任务，命名为 `mysql2tsdb.json` ，完整的配置信息如下（针对各个配置参数的详细说明，请看"参数说明"）：

```
{
  "job": {
    "content": [
      {
        "reader": {
          "name": "mysqlreader",
          "parameter": {
```

```
            "column": [
              "name",
              "type",
              "create_time",
              "price",
              "description",
              "available"
            ],
            "connection": [
              {
                "jdbcUrl": [
                  "jdbc:mysql://127.0.0.1:3306/datax"
                ],
                "table": [
                  "book"
                ]
              }
            ],
            "password": "xxx",
            "splitPk": "id",
            "username": "root"
          }
        },
        "writer": {
          "name": "tsdbwriter",
          "parameter": {
            "column": [
              "name",
              "type",
              "create_time",
              "price",
              "description",
              "available"
            ],
            "columnType": [
              "tag",
              "tag",
              "timestamp",
              "field_double",
              "field_string",
              "field_bool"
            ],
            "sourceDbType": "RDB",
            "endpoint": "http://localhost:8242",
            "table": "my_book",
            "multiField": "true",
            "username":"xxx",
            "password":"xxx",
            "ignoreWriteError":"false",
            "database":"default"
          }
        }
      }
    ],
    "setting": {
      "speed": {
        "channel": 1
      }
    }
  }
}
```

需要column字段顺序保持一致

需要保证SDB Writer插件中的column字段顺序和MySQL Reader插件中配置的column字段顺序保持一致。否则，数据会错乱。

启动MySQL到时序引擎迁移任务

```
$ cd ${DATAX_HOME}/..
$ ls
  datax/  datax.tar.gz  mysql2tsdb.json
$ python datax/bin/datax.py mysql2tsdb.json
```

检查任务是否成功

下面是任务结束后的汇总信息，看到如下内容，即可表示迁移任务已成功完成：

```
任务启动时刻                    : 2019-05-21 18:25:16
任务结束时刻                    : 2019-05-21 18:25:27
任务总计耗时                    :                 11s
任务平均流量                    :               3B/s
记录写入速度                    :              0rec/s
读出记录总数                    :                   3
读写失败总数                    :                   0
```

FAQ

Q：是否支持调整迁移进程的 JVM 内存大小？

A：支持的。以从MySQL到时序引擎的数据迁移任务为例，启动命令如下：

```
python datax/bin/datax.py mysql2tsdb.json -j "-Xms4096m -Xmx4096m"
```

Q：如何给时序引擎设置网络白名单？

A：参考设置网络白名单。

Q：将迁移任务运行在ECS上的用户，如何配置 VPC，以及常见问题有哪些？

A：参考ECS 安全组配置案例和VPC 常见问题。

14.7.3. 从 Prometheus 到 Lindorm时序引擎的数据迁移

本文介绍如何使用阿里巴巴的开源工具Datax实现从Prometheus到TSDB的数据迁移。

背景

DataX相关使用介绍请参阅DataX的README。迁移操作涉及到的两个插件（Prometheus Reader 和 TSDB Writer）。

DataX

DataX 是阿里巴巴集团内被广泛使用的离线数据同步工具 / 平台，实现包括 MySQL、Oracle、SqlServer、PostgreSQL、HDFS、Hive、ADS、HBase、TableStore（OTS）、MaxCompute（ODPS）、DRDS 等各种异构数据源之间高效的数据同步功能。

Prometheus Reader

Prometheus Reader 是 DataX 的一个插件，实现了从 Prometheus 读取数据。

TSDB Writer

TSDB Writer 是 DataX 的一个插件，实现了将数据点写入到Lindorm时序引擎中。

Quick Start

步骤一：环境准备

- Linux
- JDK（1.8 以上，推荐 1.8）
- Python（推荐 Python 2.6.x）
- Prometheus（目前只支持兼容 2.9.x，其他版本暂不保证兼容）
- Lindorm时序引擎（目前只支持兼容 2.4.x 及以上版本，其他版本暂不保证兼容）

步骤二：下载 DataX 及其插件
点击 链接
步骤三：利用 DataX 自带的迁移任务，检查迁移流程能否走通
这里，我们先以最简单的 Stream Reader 到 Stream Writer 为例。因为这两个插件不依赖任何的外部环境，特别适合用来测试流程是否能走通。简单介绍下这两个插件，其中 Stream Reader 会随机地产生字符串，而 Stream Writer 则会将接受到的字符串输出到控制台，以此模拟一个最简单的数据迁移过程。

- **工具部署**
  将下载后的安装包，解压至某个目录（作为 DATAX_HOME），即可运行迁移任务：

```
$ cd ${DATAX_HOME}
$ python bin/datax.py job/job.json
```

- **检查任务是否成功**
  下面是任务结束后的汇总信息，看到如下内容，即可表示迁移任务已成功完成：

```
任务启动时刻                    : 2019-04-26 11:18:07
任务结束时刻                    : 2019-04-26 11:18:17
任务总计耗时                    :                 10s
任务平均流量                    :          253.91KB/s
记录写入速度                    :          10000rec/s
读出记录总数                    :              100000
读写失败总数                    :                   0
```

  录制的命令行：数据迁移 Quick Start

步骤四：配置和启动OpenTSDB到时序引擎的迁移任务
通过上述 Stream Reader 到 Stream Writer 的迁移任务，我们就可以确保整个 DataX 流程是没有问题的。下面就可以真正开始 Prometheus Reader到TSDB Writer迁移任务了。

- **配置迁移任务**
  配置一个从 Prometheus 数据库同步抽取数据到 TSDB 的任务，命名为 `prometheus2tsdb.json` ，完整的配置信息如下（针对各个配置参数的详细说明，请看下一章节”参数说明”）：

```
{
  "job": {
    "content": [
      {
        "reader": {
          "name": "prometheusreader",
          "parameter": {
            "endpoint": "http://localhost:9090",
            "column": [
              "up"
            ],
            "beginDateTime": "2019-05-20T16:00:00Z",
            "endDateTime": "2019-05-20T16:00:10Z"
          }
        },
        "writer": {
          "name": "tsdbhttpwriter",
          "parameter": {
            "endpoint": "http://localhost:8242"
          }
        }
      }
    ],
    "setting": {
      "speed": {
        "channel": 1
      }
    }
  }
}
```

- **启动OpenTSDB 2时序引擎迁移任务**

```
$ cd ${DATAX_HOME}/..
$ ls
  datax/  datax.tar.gz  prometheus2tsdb.json
$ python datax/bin/datax.py prometheus2tsdb.json
```

- **检查任务是否成功**
  下面是任务结束后的汇总信息，看到如下内容，即可表示迁移任务已成功完成：

```
任务启动时刻                    : 2019-05-20 20:22:39
任务结束时刻                    : 2019-05-20 20:22:50
任务总计耗时                    :                 10s
任务平均流量                    :           122.07KB/s
记录写入速度                    :            1000rec/s
读出记录总数                    :                10000
读写失败总数                    :                    0
```

录制的命令行：从 Prometheus到TSDB数据迁移

参数说明

接下来，我们来看下各个配置项的含义：

Prometheus Reader 相关

名称	类型	是否必需	描述	默认值	举例
endpoint	String	是	Prometheus 的 HTTP 连接地址	无	`http://127.0.0.1:9090`

名称	类型	是否必需	描述	默认值	举例
column	Array	是	数据迁移任务需要迁移的 Metric 列表	`[]`	`["m"]`
beginDateTime	String	是	和 endDateTime 配合使用，用于指定哪个时间段内的数据点，需要被迁移	无	`2019-05-13 15:00:00`
endDateTime	String	是	和 beginDateTime 配合使用，用于指定哪个时间段内的数据点，需要被迁移	无	`2019-05-13 17:00:00`

TSDB Writer 相关

名称	类型	是否必需	描述	默认值	举例
endpoint	String	是	时序引擎的HTTP连接地址	无	`http://127.0.0.1:8242`
batchSize	Integer	否	每次批量数据的条数，需要保证大于0	100	100
maxRetryTime	Integer	否	失败后重试的次数，需要保证大于1	3	3
ignoreWriteError	Boolean	否	多次重试后，如果该参数设置为true，则忽略错误，继续写入；否则，则会终止写入任务	false	false

注意事项

确保与时序引擎的网络是连通的

因为 TSDB Writer 写入数据的方式是调用 HTTP 接口（ `/api/put` ）来完成的，所以需要确保迁移任务的进程能正常访问到时序引擎暴露出来的 HTTP 接口的。否则，会报错 Connect Exception 异常。

确保与Prometheus的网络是连通的

因为 Prometheus Reader 读取数据是通过调用 HTTP 接口（ `/api/v1/query_range` ）来完成的，所以需要确保迁移任务的进程能正常访问到 Prometheus 暴露出来的 HTTP 接口的。否则，会报错 Connect Exception 异常。

FAQ

Q：是否支持调整迁移进程的 JVM 内存大小？

A：以”从 Prometheus 到时序引擎的数据迁移任务”为例，启动命令如下：

```
python datax/bin/datax.py prometheus2tsdb.json -j "-Xms4096m -Xmx4096m"
```

Q：如何给时序引擎设置网络白名单？

A：参考《Lindorm时序引擎版 > 快速入门 > 设置网络白名单》。

Q：将迁移任务运行在 ECS 上的用户，如何配置 VPC，以及常见问题有哪些？

A：参考《ECS 安全组配置案例》和《VPC 常见问题》。

14.8. 搜索引擎数据迁移

前提条件

1. 已开通LTS。
2. 已把网络打通。
3. 已添加添加HBase数据源。
4. 开通DTS订阅服务， 添加DTS数据源。

14.8.1. 增量导入

本章节介绍搜索引擎如何进行增量导入。

1. 请参见RDS增量导入将增量数据导入到宽表引擎中。
2. 导入宽表引擎后，后台会通过BDS服务自动将宽表引擎的增量数据同步到搜索引擎中。

> 注意 同步到搜索引擎的数据仅限于已经配置好的schema映射。

14.8.2. 全量导入

本章节介绍搜索引擎如何进行全量导入。

1. 请参见RDS全量导入将全量数据导入到宽表引擎中。
2. 将数据导入到宽表引擎后，请参见全量数据索引构建，将全量数据导入到搜索引擎中。

14.9. 文件引擎数据迁移

14.9.1. 从自建HDFS迁移数据

本章介绍如何将在开源HDFS的数据平滑地迁移到Lindorm的文件引擎中。

背景介绍

在某些场景下面，我们需要从自建的Hadoop中存储的数据迁移到Lindorm的文件引擎当中。

适用范围

阿里云ECS自建Hadoop集群中的数据迁移到文件引擎。

准备工作

- 开通文件引擎，详情请参见开通指南 。
- 修改Hadoop 配置信息，详情请参见使用开源HDFS客户端访问。
- 检查自建的Hadoop集群和Lindorm文件引擎的连通性 。
  在自建的Hadoop 集群上执行以下命令测试集群的连通性。

  ```
  hadoop fs -ls hdfs://${实例Id}/
  ```

  其中${实例Id}请根据您的实际情况进行修改。如果可以正常查看到文件引擎的文件列表，则表示集群连通成功。
- 准备迁移工具
  您可以通过Hadoop社区标准的Distcp工具实现全量或增量的HDFS数据迁移，详细的Distcp工具使用说明请参见Hadoop Distcp 工具官方说明文档。

自建Hadoop集群数据迁移

阿里云ECS自建Hadoop集群和文件引擎处于相同VPC网络环境时，可以直接通过VPC网络迁移数据到文件引擎。迁移命令如下所示：

```
hadoop distcp  -m 1000 -bandwidth 30 hdfs://oldcluster:8020/user/hive/warehouse  hdfs://${实例Id}/user/hive/warehouse
```

其中oldcluster为原自建Hadoop集群namenode的IP或者域名，${实例Id}为文件引擎的实例Id，请根据您的实际情况进行修改。

常见问题

- 整体迁移速度受Hadoop集群与文件存储HDFS之间的带宽、集群规模影响。如果迁移数据量大，建议先尝试迁移几个目录评估下整体时间。如果只能在指定时间段内迁移数据，可以将目录切分为几个小目录，依次迁移。
- 一般全量数据同步时，需要一个短暂的业务停写过程，用来启用双写双算或直接将业务切换到新集群上。

14.9.2. 从OSS迁移数据

本章节介绍如何将对象存储OSS上的数据迁移到文件引擎。

准备工作

1. 开通文件引擎，详情请参见 开通指南 。
2. 搭建Hadoop集群。建议您使用的Hadoop版本不低于2.7.3，本文档中使用的Hadoop版本为Apache Hadoop 2.7.3，修改Hadoop 配置信息，详情参见使用开源HDFS客户端访问。
3. 在Hadoop集群所有节点上安装JDK，本操作要求JDK版本不低于1.8。
4. 在Hadoop集群安装OSS客户端JindoFS SDK。JindoFS SDK详细介绍请参见JindoFS SDK。
   - 下载 jindofs-sdk.jar。

     ```
     cp ./jindofs-sdk-*.jar  ${HADOOP_HOME}/share/hadoop/hdfs/lib/
     ```

   - 为Hadoop集群所有节点创建JindoFS SDK配置文件。
     - 添加如下环境变量到 `/etc/profile` 文件。

       ```
       export B2SDK_CONF_DIR=/etc/jindofs-sdk-conf
       ```

     - 创建OSS存储工具配置文件 `/etc/jindofs-sdk-conf/bigboot.cfg` 。

       ```
       [bigboot]
       logger.dir=/tmp/bigboot-log[bigboot-client]
       client.oss.retry=5
       client.oss.upload.threads=4
       client.oss.upload.queue.size=5
       client.oss.upload.max.parallelism=16
       client.oss.timeout.millisecond=30000
       client.oss.connection.timeout.millisecond=4000
       ```

     - 加载环境变量使之生效。

       ```
       source /etc/profile
       ```

     - 验证是否可以在Hadoop 集群上使用 oss。

       ```
       ${HADOOP_HOME}/bin/hadoop fs -ls oss://<accessKeyId>:<accessKeySecret>@<bucket-name>.<endpoint>/
       ```

将对象存储OSS数据迁移到文件引擎

1. 检查并且确定需要迁移的数据大小。

   ```
   ${HADOOP_HOME}/bin/hadoop du -h oss://<accessKeyId>:<accessKeySecret>@<bucket-name>.<endpoint>/test_data
   ```

2. 启动Hadoop MapReduce任务（DistCp）将测试数据迁移至文件引擎。

   ```
   ${HADOOP_HOME}/bin/hadoop distcp  \
   oss://<accessKeyId>:<accessKeySecret>@<bucket-name>.<endpoint>/test_data.txt \
   hdfs://${实例Id}/
   ```

   其中${实例Id}请根据您的实际情况进行修改。
   参数说明如下表所示：

参数	说明
accessKeyId	

参数	说明
accessKeySecret	对象存储OSS API的密钥。获取方式请参见创建AccessKey。
bucket-name.endpoint	对象存储OSS的访问域名，包括存储空间（Bucket）名称和对应的地域域名（Endpoint）地址。

3. 任务执行完成后，查看迁移结果。
   如果回显包含如下类似信息，说明迁移成功。

```
20/09/29 12:23:59 INFO mapreduce.Job:  map 100% reduce 0%
20/09/29 12:23:59 INFO mapreduce.Job: Job job_1601195105349_0015 completed successfully
20/09/29 12:23:59 INFO mapreduce.Job: Counters: 38
 File System Counters
  FILE: Number of bytes read=0
  FILE: Number of bytes written=122343
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=470
  HDFS: Number of bytes written=47047709
  HDFS: Number of read operations=15
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=4
  OSS: Number of bytes read=0
  OSS: Number of bytes written=0
  OSS: Number of read operations=0
  OSS: Number of large read operations=0
  OSS: Number of write operations=0
 Job Counters
  Launched map tasks=1
  Other local map tasks=1
  Total time spent by all maps in occupied slots (ms)=5194
  Total time spent by all reduces in occupied slots (ms)=0
  Total time spent by all map tasks (ms)=5194
  Total vcore-milliseconds taken by all map tasks=5194
  Total megabyte-milliseconds taken by all map tasks=5318656
 Map-Reduce Framework
  Map input records=1
  Map output records=0
  Input split bytes=132
  Spilled Records=0
  Failed Shuffles=0
  Merged Map outputs=0
  GC time elapsed (ms)=64
  CPU time spent (ms)=2210
  Physical memory (bytes) snapshot=222294016
  Virtual memory (bytes) snapshot=2672074752
  Total committed heap usage (bytes)=110100480
 File Input Format Counters
  Bytes Read=338
 File Output Format Counters
  Bytes Written=0
 org.apache.hadoop.tools.mapred.CopyMapper$Counter
  BYTESCOPIED=47047709
  BYTESEXPECTED=47047709
  COPY=1
20/09/29 12:23:59 INFO common.AbstractJindoFileSystem: Read total statistics: oss read average -1 us, ca
che read average -1 us, read oss percent 0%
```

4. 验证迁移结果。
   查看迁移到文件引擎的测试数据大小。

```
${HADOOP_HOME}/bin/hadoop fs -du -s -h hdfs://${实例Id}/
```

15.SDK参考

15.1. OpenAPI-SDK下载

OpenAPI的SDK用于管理Lindorm实例（例如创建实例、扩容实例、删除实例）和设置白名单。本文介绍Lindorm OpenAPI支持的SDK。

下表列举了各语言SDK的下载地址和开发指南，更多SDK信息，请参见阿里云开放平台。

阿里云SDK	Lindorm版SDK	开发指南文档
Alibaba Cloud SDK for Java	Alibaba Cloud Lindorm SDK for Java	Alibaba Cloud SDK for Java 快速开始
Alibaba Cloud SDK for Python	Alibaba Cloud Lindorm SDK for Python	Alibaba Cloud SDK for Python 快速开始
Alibaba Cloud SDK for Go	Alibaba Cloud Lindorm SDK for Go	Alibaba Cloud SDK for Go 快速开始
Alibaba Cloud SDK for PHP	Alibaba Cloud Lindorm SDK for PHP	Alibaba Cloud SDK for PHP 快速开始

通过Maven依赖下载SDK安装包时，请下载最新版本的SDK安装包。以下示例为Java通过Maven依赖下载SDK的方式。

```
<dependency>
    <groupId>com.aliyun</groupId>
    <artifactId>aliyun-java-sdk-hitsdb</artifactId>
    <version>3.0.4</version>
</dependency>
<dependency>
    <groupId>com.aliyun</groupId>
    <artifactId>aliyun-java-sdk-core</artifactId>
    <version>4.5.21</version>
</dependency>
```

15.2. Java SDK for Lindorm OpenAPI使用说明

阿里云开发者工具套件（Alibaba Cloud SDK for Java）让您不用复杂编程可对Lindorm实例进行管理。本文介绍Alibaba Cloud SDK for Java工具的安装和使用操作，并提供示例供您参考。

前提条件

- 已创建AccessKey。具体操作，请参见获取AccessKey。
- 已下载Lindorm版Java SDK安装包。具体操作，请参见OpenAPI-SDK下载。

操作步骤

1. 安装Lindorm版Java SDK安装包。具体操作，请参见安装Alibaba Cloud SDK for Java。
2. 设置地域和AccessKey信息。参数说明如下表。

```
IClientProfile profile = DefaultProfile.getProfile("<RegionId>","<accessKeyId>","<accessSecret>");
```

参数名称	参数描述
RegionId	地域ID。
accessKeyId	testAccessKey ID。
accessSecret	testAccessKey Secret。

示例如下：

```
IClientProfile profile = DefaultProfile.getProfile("cn-shanghai","LTAI4FnvzUftE****","PxPzgelK1XCkWBXW****");
```

3. 设置Endpoint信息。参数说明如下表。

> **说明** Endpoint是阿里云服务的API服务端地址。针对不同的地域，单个服务可能有不同的Endpoint。阿里云SDK内置了Endpoint寻址模块，当您调用SDK对一个服务发起请求时，SDK会自动根据您设置的地域ID（Region ID）和产品ID来找到Endpoint。各地域的Endpoint信息请参见服务地址。

```
DefaultProfile.addEndpoint("<RegionId>", "hitsdb", "<domain>");
```

参数名称	参数描述
RegionId	地域ID。
hitsdb	产品编码，固定值hitsdb。
domain	接入地址，即域名信息。

示例如下：

```
DefaultProfile.addEndpoint("cn-shanghai", "hitsdb", "hitsdb.cn-shanghai.aliyuncs.com");
```

4. 初始化客户端。

```
IAcsClient client = new DefaultAcsClient(profile);
```

5. 构造请求对象。

```
GetLindormInstanceRequest request = new GetLindormInstanceRequest();
request.setInstanceId("ld-****");
```

6. 请求调用。

```
GetLindormInstanceResponse response;
try {
      response = client.getAcsResponse(request);
      System.out.println(new Gson().toJson(response));
} catch (ClientException e) {
      e.printStackTrace();
}
```

请求示例参考

```
DefaultProfile profile = DefaultProfile.getProfile("cn-shanghai",
        "ak****", "akSecret****");
DefaultProfile.addEndpoint(
        "cn-shanghai",
        "hitsdb",
        "hitsdb.cn-shanghai.aliyuncs.com"
);
IAcsClient client = new DefaultAcsClient(profile);
GetLindormInstanceRequest request = new
    GetLindormInstanceRequest();
request.setInstanceId("ld-****");
GetLindormInstanceResponse response;
try {
    response = client.getAcsResponse(request);
    System.out.println(new Gson().toJson(response));
} catch (ClientException e) {
    e.printStackTrace();
}
```

响应示例参考

```
{
  "InstanceStorage": "480",
  "EngineList": [
    {
      "MemorySize": "8",
      "CpuCount": "4",
      "Version": "2.2.3",
      "IsLastVersion": "false",
      "Engine": "lindorm",
      "CoreCount": "2"
    }
  ],
  "ZoneId": "cn-shanghai-f",
  "RequestId": "1986EB6C-B50B-4F41-B365-38A1095DA5C2",
  "InstanceId": "ld-****",
  "ColdStorage": "800",
  "AutoRenew": "false",
  "CreateTime": "2021-01-12 20:40:20",
  "DiskCategory": "cloud_efficiency",
  "DeletionProtection": "false",
  "PayType": "POSTPAY",
  "VswitchId": "vsw-****",
  "VpcId": "vpc-****",
  "ServiceType": "lindorm",
  "NetworkType": "vpc",
  "InstanceStatus": "ACTIVATION",
  "InstanceAlias": "lindorm_test",
  "EngineType": "4",
  "RegionId": "cn-shanghai",
  "ExpireTime": "2020-12-17 00:00:00",
  "AliUid": "100000000000****"
}
```


15.3. 宽表引擎

15.3.1. HBase API

15.3.1.1. 升级HBase Java SDK

通过HBase Java SDK访问云原生多模数据库Lindorm宽表引擎，请参考以下说明升级HBase客户端版本。

前提条件

已安装Java环境，Alibaba Cloud SDK for Java要求使用JDK1.8或更高版本。

使用须知

- HBase 1.x版本客户端对应1.x版本的HBase集群。
- HBase 2.x版本客户端对应2.x版本的HBase集群。
- 新用户推荐使用2.x版本客户端。

> 注意　如果目前已依赖社区版HBase客户端，那么无法升级到上述推荐的HBase客户端版本，可以通过以下方式升级HBase客户端版本。
>
> - 通过Maven方式，使用与社区版本相适配的云数据库HBase客户端，请参见历史版本适配（Maven方式）。
> - 通过导入.jar包方式，使用与社区版本相适配的云数据库HBase客户端，请参见通过导入JAR文件升级HBase客户端。

升级HBase Java SDK

- 如果客户端是HBase 1.x版本，您可以在pom.xml文件中添加Maven依赖来升级HBase Java SDK。

```
<dependency>
  <groupId>com.aliyun.hbase</groupId>
  <artifactId>alihbase-client</artifactId>
  <version>1.8.7</version>
</dependency>
```

- 如果客户端是HBase 2.x版本，您可以在pom.xml文件中添加Maven依赖来升级HBase Java SDK。

```
<dependency>
  <groupId>com.aliyun.hbase</groupId>
  <artifactId>alihbase-client</artifactId>
  <version>2.8.7</version>
</dependency>
```

15.3.1.2. 通过HBase非Java SDK访问Lindorm宽表引擎

云原生多模数据库Lindorm宽表引擎支持通过HBase非Java（例如C++、Python和Go等）API进行访问，本文介绍具体的SDK安装和访问操作。

背景信息

云原生多模数据库Lindorm宽表引擎通过Thrift支持非Java语言（例如C++、Python和Go等）访问。

云原生多模数据库Lindorm宽表引擎使用的HBase Thrift2接口定义，所以需要下载HBase Thrift2定义文件来生成对应语言的接口。与HBase Thrift1定义相比，Lindorm中的HBase Thrift2接口定义更加清晰，您可以获得和Java语言类似的API调用体验。目前HBase Thrift2定义文件比HBase Thrift1定义文件的功能更全而且更容易使用。

准备工作

- 已下载Thrift安装包，下载链接请单击Thrift安装包。
- 已下载HBase Thrift2定义文件，下载链接请单击HBase Thrift2定义文件。
- 已获取Lindorm宽表引擎的**使用HBase 非Java API访问**的连接地址，具体操作请参见查看连接地址。



访问Lindorm宽表引擎（以Python语言为例）

Thrift安装包的使用请参见Apache Thrift官方文档，通过Thrift访问云原生多模数据库Lindorm宽表引擎的步骤如下：

1. 使用HBase Thrift2定义文件来生成对应语言的接口文件。 命令语句如下：

   ```
   thrift --gen <language> Hbase.thrift
   ```

   > **说明** `language` 为目标语言，例如：python、php、cpp、py等。

   语句示例：

```
thrift --gen python Hbase.thrift
```

2. 构造客户端访问云原生多模数据库Lindorm宽表引擎。
   Lindorm中Thrift服务器端的transport层使用的是HTTP，因此在构造客户端时，需要thrift中的ThttpClient（各个语言都有相应实现）。并且在ACL开启的情况下，需要在ThttpClient上加上两个header来向服务器传输用户名和密码进行认证（如果关闭ACL则不需要）。Thrift在每个语言实现的ThttpClient都有加定制header的函数。以Python语言为例，使用以下语句构造客户端并访问Lindorm宽表引擎。

```
# -*- coding: utf-8  -*-
# 以下两个模块通过执行pip install thrift 语句生成
from thrift.protocol import TBinaryProtocol
from thrift.transport import THttpClient
# 以下两个模块通过执行thrift --gen py hbase.thrift 语句生成
from hbase import THBaseService
from hbase.ttypes import TColumnValue, TColumn, TTableName, TTableDescriptor, TColumnFamilyDescriptor,
TNamespaceDescriptor, TGet, TPut, TScan
# 配置Lindorm宽表引擎的连接地址
url = "http://ld-bp17j28j2y7pm****-proxy-lindorm.lindorm.rds.aliyuncs.com:9190"
transport = THttpClient.THttpClient(url)
headers = {}
# 设置用户名
headers["ACCESSKEYID"]="root";
# 设置用户名对应的密码
headers["ACCESSSIGNATURE"]="root"
transport.setCustomHeaders(headers)
protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)
client = THBaseService.Client(protocol)
transport.open()
# 具体操作后，执行关闭连接操作
transport.close()
```

相关代码示例

其他非Java语言的完整代码示例请直接下载Github相应语言的代码示例，下载链接如下所示。

- Python
- Go
- C++
- node.js
- PHP

15.3.1.3. 通过Maven方式升级HBase客户端

本文介绍通过Maven方式将社区版HBase客户端升级至最新版本的云数据库HBase客户端。

使用场景

适用于目前已依赖社区版HBase客户端，并且无法升级至最新稳定版的云数据库HBase客户端。

应用直接依赖社区版HBase客户端

```
<dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-client</artifactId>
      <version>x.x.x</version>
</dependency>
```

替换成

```
<dependency>
    <groupId>com.alibaba.hbase</groupId>
    <artifactId>alihbase-client</artifactId>
    <version>x.x.x</version>
</dependency>
<!--${connector-version} 需要替换成hbase-client对应的版本，版本对应关系见最下面表格的依赖-->
<dependency>
    <groupId>com.alibaba.hbase</groupId>
    <artifactId>alihbase-connector</artifactId>
    <version>${connector-version}</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

应用间接依赖社区版HBase客户端

适用于应用依赖的第三方包依赖了HBase。

```
<!--从第三方包依赖如下-->
<dependency>
      <groupId>xxx.xxx.xxx</groupId>
      <artifactId>yyy</artifactId>
      <version>z.z.z</version>
</dependency>
```

替换成

```
<!--从第三方包中去掉社区版hbase-client的依赖-->
<dependency>
    <groupId>xxx.xxx.xxx</groupId>
    <artifactId>yyy</artifactId>
    <version>z.z.z</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!--增加alihbase-client的依赖-->
<dependency>
    <groupId>com.alibaba.hbase</groupId>
    <artifactId>alihbase-client</artifactId>
    <version>x.x.x</version>
</dependency>
<!--${connector-version} 需要替换成hbase-client对应的版本，版本对应关系见最下面表格的依赖-->
<dependency>
    <groupId>com.alibaba.hbase</groupId>
    <artifactId>alihbase-connector</artifactId>
    <version>${connector-version}</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

hbase-client同alihbase-connector版本对应关系

hbase-client版本	connector版本
1.0.0	1.1.0.1
1.0.1	1.1.0.1
1.0.1.1	1.1.0.1
1.0.2	1.1.0.1
1.0.3	1.1.0.1
1.1.0	1.1.0.1
1.1.0.1	1.1.0.1
1.1.1	1.1.0.1
1.1.2	1.1.0.1
1.1.3	1.1.0.1
1.1.4	1.1.0.1
1.1.5	1.1.0
1.1.6	1.1.0
1.1.7	1.1.0
1.1.8	1.1.0
1.1.9	1.1.0
1.1.10	1.1.0
1.1.11	1.1.0
1.1.12	1.1.0
1.1.13	1.1.0
1.2.0	1.1.0.1
1.2.1	1.1.0
1.2.2	1.1.0
1.2.3	1.1.0
1.2.4	1.1.0
1.2.5	1.1.0
1.2.6	1.1.0
1.2.6.1	1.1.0
1.2.7	1.1.0
1.2.8	1.1.0

hbase-client版本	connector版本
1.2.9	1.1.0
1.2.10	1.1.0
1.2.11	1.1.0
1.2.12	1.1.0
1.3.0	1.1.0
1.3.1	1.1.0
1.3.2	1.1.0
1.3.2.1	1.1.0
1.3.3	1.1.0
1.3.4	1.1.0
1.3.5	1.1.0
1.3.6	1.1.0
1.4.0	1.1.0
1.4.1	1.1.0
1.4.2	1.1.0
1.4.3	1.1.0
1.4.4	1.1.0
1.4.5	1.1.0
1.4.6	1.1.0
1.4.7	1.1.0
1.4.8	1.1.0
1.4.9	1.1.0
1.4.10	1.1.0
1.4.11	1.1.0
1.4.12	1.1.0
1.4.13	1.1.0
1.5.0	1.1.0
1.6.0	1.1.0
2.0.0	2.1.0
2.0.1	2.1.0
2.0.2	2.1.0.1

hbase-client版本	connector版本
2.0.3	2.1.0.1
2.0.4	2.1.0.1
2.0.5	2.1.0.1
2.0.6	2.1.0.1
2.1.0	2.1.0
2.1.1	2.1.0.1
2.1.2	2.1.0.1
2.1.3	2.1.0.1
2.1.4	2.1.0.1
2.1.5	2.1.0.1
2.1.6	2.1.0.1
2.1.7	2.1.0.1
2.1.8	2.1.0.1
2.1.9	2.1.0.1
2.2.0	2.1.0.1
2.2.1	2.1.0.1
2.2.2	2.1.0.1
2.2.3	2.1.0.1

15.3.1.4. 通过导入JAR文件升级HBase客户端

本文介绍通过导入JAR文件将社区版HBase客户端升级至最新版本的云数据库HBase客户端。

使用场景

适用于目前已依赖社区版HBase客户端，并且无法升级至最新稳定版的云数据库HBase客户端。

下载JAR文件

在开源服务的*lib*目录下找到名称为*hbase-client-x.x.x.jar*的JAR文件，将*hbase-client-x.x.x.jar*JAR文件替换为对应版本的alihbase-client的JAR文件，例如替换前JAR文件为*hbase-client-1.0.0.jar*，则替换后为*alihbase-client-1.0.0.jar*。

您可以通过以下地址下载云数据库HBase客户端的JAR文件。请修改替换后的版本号，例如将 `alihbase-client-x.x.x.jar` 修改为 `alihbase-client-1.0.0.jar` 。

```
https://hbaseuepublic.oss-cn-beijing.aliyuncs.com/alihbase-client-x.x.x.jar
```

下载插件

使用以下语句下载访问云数据库HBase增强版插件。

```
#1.x版本插件
https://hbaseuepublic.oss-cn-beijing.aliyuncs.com/alihbase-connector-1.1.0.jar
#2.x版本插件
https://hbaseuepublic.oss-cn-beijing.aliyuncs.com/alihbase-connector-2.1.0.jar
```

15.3.2. Cassandra CQL Driver

15.3.2.1. Cassandra CQL Java Driver 安装和升级

通过Cassandra CQL Java Driver访问Lindorm宽表，请参考以下说明升级客户端版本

Mavenue依赖

可以用Datastax的java driver访问Lindorm的CQL，推荐不低于datastax java driver 3.7 版本，maven依赖如下：

```
<dependency>
    <groupId>com.datastax.cassandra</groupId>
    <artifactId>cassandra-driver-core</artifactId>
    <version>3.7.2</version>
</dependency>
```

这个依赖会引入一些公共库，为了避免依赖冲突导致不必要的麻烦，请先在一个新工程中测试。

15.4. 时序引擎

15.4.1. 通过JDBC Driver连接并访问时序引擎

本文介绍通过Lindorm JDBC Driver连接并访问时序引擎的具体操作，以及JDBC Driver访问时序引擎时支持的API接口和方法。

背景信息

Lindorm时序引擎的JDBC Driver遵循JDBC 4.1规范定义的接口。由于时序引擎自身是一个时序数据库，所以JDBC Driver中提供的实际上是JDBC 1.0定义的基础功能集中的一个子集。

前提条件

- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 已将客户端IP地址添加至Lindorm白名单，具体操作请参见设置白名单。
- 已获取云原生多模数据库Lindorm时序引擎的连接地址，具体操作请参见查看连接地址。

操作步骤

1. 通过以下两种方式安装Lindorm JDBC Driver依赖。
   - 手动安装。
     在本地自行下载JAR包集成JDBC Driver，下载链接为：Lindorm-all-client。选择需要安装的版本，以2.0.14为例，下载lindorm-all-client-2.0.14.jar包。
   - 通过Maven下载。
     如果在Maven项目中集成JDBC Driver，创建Project并在pom.xml中添加以下依赖配置，具体内容如下：

     ```
     <dependency>
         <groupId>com.aliyun.lindorm</groupId>
         <artifactId>lindorm-all-client</artifactId>
         <version>2.0.14</version>
     </dependency>
     ```

     > **说明**
     > lindorm-all-client的版本号根据需求填写。
2. 访问Lindorm时序引擎。完整的代码示例请参考访问示例。

i. 建立Lindorm时序引擎和数据之间的连接。

```
String url = "jdbc:lindorm:tsdb:url=http://<host:port>";
Connection conn = DriverManager.getConnection(url);
```

> 说明
>
> - `host:port`：Lindorm时序引擎的连接地址和端口，例如：`ld-bp17j28j2y7pm****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242`。
> - `jdbc:lindorm:tsdb:url=http://<host:port>`：表示JDBC的连接地址，具体语法和连接参数说明请参见JDBC的连接地址说明。

ii. 创建Lindorm时序数据库和表。

```
Statement stmt = conn.createStatement();

stmt.execute("CREATE DATABASE test");
stmt.execute("USE test");
stmt.execute("CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time TIMESTAMP,temperatu
re DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");
```

iii. 在表sensor中插入数据。

```
//PreparedStatement 批量写入数据（推荐使用）
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO sensor (device_id, region, time, temperature, humidity) VALUES (?, ?, ?, ?, ?)");
int batchSize = 100;
long ts = System.currentTimeMillis();

for (int i = 0; i < batchSize; i++) {

     //按INSERT语句中的列参数顺序依次绑定参数
     pstmt.setString(1, "F07A1260");
     pstmt.setString(2, "north-cn");
     pstmt.setLong(3, ts + i * 10000);
     pstmt.setDouble(4, 12.1);
     pstmt.setLong(5, 45);

     pstmt.addBatch();
}

pstmt.executeBatch();
pstmt.clearBatch();

//批量写入数据
Statement stmt = conn.createStatement();
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1260', 'north-cn',1619076780000,12.1,45)");
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1260', 'north-cn',1619076790000,13.2,47)");
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1260', 'north-cn',1619076800000,10.6,46)");
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1261', 'south-cn',1619076780000,18.1,44)");
stmt.addBatch("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1261', 'south-cn',1619076790000,19.7,44)");
stmt.executeBatch();
stmt.clearBatch();

//单条写入数据（不推荐使用）
Statement stmt = conn.createStatement();
stmt.execute("INSERT INTO sensor(device_id, region, time, temperature, humidity) values('F07A1260','north-cn',1619076780000,12.1,45)");
```

iv. 查询表sensor中的数据。

```
ResultSet rs = stmt.executeQuery("select device_id, region,time,temperature,humidity from sensor where time >= 1619076780000 and time <= 1619076800000");

while (rs.next()) {
    String device_id = rs.getString("device_id");
    String region = rs.getString("region");
    Long time = rs.getLong("time");
    Double temperature = rs.getDouble("temperature");
    Long humidity = rs.getLong("humidity");
    System.out.printf("%s %s %d %f %d\n", device_id, region, time, temperature, humidity);
}
```

支持的API接口和方法

JDBC Driver访问时序引擎时支持的API接口和对应的方法如下表所示。

> **说明**
> 下表未列出的API接口部分兼容上层框架，会返回默认值。如果您有其他问题，请提交工单。

接口	支持的方法
Connection	• `close()` • `createStatement(int, int)` • `createStatement(int, int, int)` • `getSchema()` • `prepareStatement(String, int, int)` • `prepareStatement(String, int, int, int)` • `setSchema(String)` **说明** 由于Lindorm JDBC Driver的Connection接口实现不是线程安全的，所以建议不要在多个线程中共用同一个Connection对象。
Statement	• `addBatch(String)` • `close()` • `execute(String)` • `executeQuery(String)` • `getMoreResult` • `isClosed()` • `setFetchSize(int)`
ResultSet	• `addBatch()` • `clearBatch()` • `close()` • `execute()` • `executeBatch()` • `executeQuery()` • `getParameterNumber()` • `setNull(int)` • `setXXX (int, T)` **说明** Lindorm时序引擎支持的数据类型请参见数据类型。

接口	支持的方法
PreparedStatement	• `close()` • `getStatement()` • `next()` • `wasNull()` • `getXXX(int)` 说明 Lindorm时序引擎支持的数据类型请参见数据类型。

JDBC的连接地址说明

JDBC的连接地址是应用程序基于Connection接口建立数据库连接时的重要依据，通常用于指定数据库连接的协议，实际是通过JDBC Driver实现。

JDBC的连接地址语法如下所示：

```
jdbc:lindorm:tsdb:url=http://${host:port}[;${connection_parameter}=${connection_value}]...
```

说明

`host:port`：Lindorm时序引擎的连接地址和端口，例如：`ld-bp17j28j2y7pm****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242`。

JDBC的连接地址中支持的连接参数和说明如下表：

参数	说明
user	Lindorm实例的用户名。用于指定连接时序引擎时进行用户认证的用户名。 说明 当引擎的用户鉴权功能未启用时，用户名可以不指定。
password	用户名对应的密码。用于指定连接时序引擎时进行用户认证的密码。如果您忘记了密码，可以通过Lindorm宽表引擎的集群管理系统修改密码，具体操作请参见修改用户密码。 说明 当引擎的用户鉴权功能未启用时，密码可以不指定。
database	连接访问的Database。如果该参数未指定，默认访问`default`数据库。
lindorm.tsdb.driver.connect.timeout	连接时序引擎服务端的超时时间，单位为ms（毫秒）。默认值60秒（即60000毫秒）。
lindorm.tsdb.driver.socket.timeout	Socket读写间隔超时时间，单位为ms（毫秒）。默认值为-1，表示无超时。

> **注意**
> 服务端为了提高资源利用率，在连接Lindorm客户端空闲90s后，服务端会主动断开连接。如果还需要使用该连接，会出现 `com.aliyun.lindorm.client.shaded.org.apache.calcite.avatica.http.ConnectionDisconnectedException` 报错，重新建立连接即可解决。

使用Druid连接池访问时序引擎

使用Java开发访问时序引擎的应用程序时，通常建议通过JDBC Driver连接Lindorm时序引擎，但是JDBC访问逻辑的基础是需要创建Connection对象。如果每次连接时序引擎都创建一个新的Connection对象会带来很大的开销。为了降低开销，解决方法是使用连接池来执行连接的创建和管理。
目前可以选择的连接池有很多种，以下介绍开源连接池Alibaba Druid的使用步骤。

1. 在Maven项目中添加Druid连接池的依赖。打开客户端，创建Project并在pom.xml中配置Maven依赖，具体内容如下：

```
<dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid</artifactId>
      <version>1.2.6</version>
    </dependency>
```

2. 在Maven项目中添加Lindorm JDBC Driver依赖，具体请参见操作步骤中的步骤1。
3. 执行以下代码创建DataSource对象，为DataSource设置Lindorm时序引擎的JDBC连接地址。

```
dataSource = new DruidDataSource();
        try {
            dataSource.setDriver(DriverManager.getDriver("jdbc:lindorm:tsdb:url=http://<host:port>"));
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
```

> **说明**
> `host:port`：Lindorm时序引擎的连接地址和端口，例如：`ld-bp17j28j2y7pm****-proxy-tsdb.lindorm.rds.aliyuncs.com:8242`。

4. 基于DataSource创建JDBC Connection并访问Lindorm时序引擎。完整的代码如下：

```
import com.alibaba.druid.pool.DruidDataSource;
import java.sql.*;

public class App {
    private DruidDataSource dataSource;

    public void setUp() {
        dataSource = new DruidDataSource();
        try {
            dataSource.setDriver(DriverManager.getDriver("jdbc:lindorm:tsdb:url=http://ld-bp17j28j2y7pm*
***-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242"));
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        dataSource.setUrl("jdbc:lindorm:tsdb:url=http://ld-bp17j28j2y7pm****-proxy-tsdb-pub.lindorm.rds.
aliyuncs.com:8242");
        dataSource.setValidationQuery("select 1");
        dataSource.setMaxActive(1);
        dataSource.setUsername("username");
        dataSource.setPassword("****");
        dataSource.setKeepAlive(true);
        dataSource.setMinEvictableIdleTimeMillis(80);
        dataSource.setMaxWait(10000);
        dataSource.init();

    }

    public void close() {
        dataSource.close();
    }

    public void testQuery() {
        try (Connection conn = this.dataSource.getConnection()) {
            try (Statement stmt = conn.createStatement()) {
                ResultSet rs = stmt.executeQuery("select * from sensor");
                long count = 0;
                while (rs.next()) {
                    count++;
                }
                System.out.println("count=" + count + " ");
            } catch (RuntimeException e) {
                e.printStackTrace();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        } catch (RuntimeException e) {
            System.out.println("connection failed.");
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main( String[] args ) {
        App app = new App();
        app.setUp();
        app.testQuery();
        app.close();
    }
}
```

15.4.2. 通过Native SDK连接并访问时序引擎

本文档主要介绍Lindorm 时序引擎Java Native SDK的安装以及快速开始。

前提条件

- 云原生多模数据库Lindorm实例已开通时序引擎。
- 已安装Java环境，要求安装JDK 1.8及以上版本。
- 时序引擎版本为3.4.7及以上版本，查看和升级时序引擎版本的方法请参见时序引擎版本说明。
- 已将客户端IP地址添加至Lindorm白名单，具体操作请参见设置白名单。
- 已获取云原生多模数据库Lindorm时序引擎的连接地址，获取方法请参见获取连接串。

准备工作

通过Java Native SDK连接Lindorm时序引擎前，需要安装Java Native SDK。以1.0.0版本为例，您可以通过以下三种方式安装Java Native SDK：

- （推荐）在Maven项目中使用Lindorm TSDB Java SDK。在*pom.xml*文件的`dependencies`中添加以下依赖项。

```
<dependency>
  <groupId>com.aliyun.lindorm</groupId>
  <artifactId>lindorm-tsdb-client</artifactId>
  <version>1.0.0</version>
</dependency>
```

> **说明** Lindorm TSDB Java SDK提供了一个基于Maven的示例工程，您可以直接下载示例工程并在本地编译和运行，也可以以示例工程为基础开发您的项目工程。

- 在Eclipse项目中导入JAR包。
  i. 下载Java SDK开发包。
  ii. 解压下载的Java SDK开发包。
  iii. 将解压后JAR包添加至Eclipse项目中。
    a. 在Eclipse中打开您的项目，右键单击该项目，选择**Properties**。
    b. 在弹出的对话框中，单击**Java Build Path** > **Libraries** > **Add JARs**，选择解压后的*lindorm-tsdb-client-1.0.0.jar*和*lib*文件中的JAR包。
    c. 单击**Apply and Close**。
- 在IntelliJ IDEA项目中导入JAR包。
  i. 下载Java SDK开发包。
  ii. 解压下载的Java SDK开发包。
  iii. 将解压后JAR包添加至IntelliJ IDEA项目中。
    a. 在IntelliJ IDEA中打开您的项目，在菜单栏单击**File** > **Project Structure**。
    b. 在**Project Structure**对话框的左边选择**Project Structure** > **Modules**。
    c. 单击右边  ，选择**JARs or directories**。
    d. 在弹出的对话框中，选择解压后的*lindorm-tsdb-client-1.0.0.jar*和*lib*文件中的JAR包，并单击**OK**。
    e. 单击**Apply**。
    f. 单击**OK**。

> **说明**
> - Lindorm时序引擎的Java Native SDK各版本可以通过Maven中央仓库获取，更多信息请参见Maven Repository。
> - Lindorm时序引擎的Java Native SDK各版本说明请参见版本说明。

操作步骤

1. 创建数据库实例。新建LindormTSDBClient时，需要指定Lindorm时序引擎的连接地址，获取方法请参见获取连接串。

```
String url = "http://ld-bp17j28j2y7pm****-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242";
// LindormTSDBClient线程安全，可以重复使用，无需频繁创建和销毁
ClientOptions options = ClientOptions.newBuilder(url).build();
LindormTSDBClient lindormTSDBClient = LindormTSDBFactory.connect(options);
```

2. 创建数据库demo和时序表sensor。关于创建数据库和时序表的SQL语句说明，请参见CREATE DATABASE和CREATE TABLE。

```
lindormTSDBClient.execute("CREATE DATABASE demo");
lindormTSDBClient.execute("demo","CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time BIG
INT,temperature DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");
```

3. 在表中写入数据。

> **说明** 默认情况下，为了提高写入数据的性能，LindormTSDBClient通过异步攒批的方式进行数据写入。如果需要通过同步的方式进行数据写入，可以调用 `write()` 方法返回的 `CompletableFuture<WriteResult>` 的 `join()` 方法。

```
int numRecords = 10;
List<Record> records = new ArrayList<>(numRecords);
long currentTime = System.currentTimeMillis();
for (int i = 0; i < numRecords; i++) {
    Record record = Record
        .table("sensor")
        .time(currentTime + i * 1000)
        .tag("device_id", "F07A1260")
        .tag("region", "north-cn")
        .addField("temperature", 12.1 + i)
        .addField("humidity", 45.0 + i)
        .build();
    records.add(record);
}
CompletableFuture<WriteResult> future = lindormTSDBClient.write("demo", records);
// 处理异步写入结果
future.whenComplete((r, ex) -> {
    // 处理写入失败
    if (ex != null) {
        System.out.println("Failed to write.");
        if (ex instanceof LindormTSDBException) {
            LindormTSDBException e = (LindormTSDBException) ex;
            System.out.println("Caught an LindormTSDBException, which means your request made it to Lind
orm TSDB, "
                               + "but was rejected with an error response for some reason.");
            System.out.println("Error Code: " + e.getCode());
            System.out.println("SQL State:  " + e.getSqlstate());
            System.out.println("Error Message: " + e.getMessage());
        }  else {
            ex.printStackTrace();
        }
    } else  {
        System.out.println("Write successfully.");
    }
});
// 这里作为示例，简单同步处理写入结果
System.out.println(future.join());
```

4. 查询时序表的数据。关于查询操作的SQL语句说明，请参见基本查询。

```
String sql = "select * from sensor limit 10";
ResultSet resultSet = lindormTSDBClient.query("demo", sql);
try {
    // 处理查询结果
    QueryResult result = null;
    // 查询结果使用分批的方式返回，默认每批1000行
    // 当resultSet的next()方法返回为null，表示已经读取完所有的查询结果
    while ((result = resultSet.next()) != null) {
        List<String> columns = result.getColumns();
        System.out.println("columns: " + columns);
        List<String> metadata = result.getMetadata();
        System.out.println("metadata: " + metadata);
        List<List<Object>> rows = result.getRows();
        for (int i = 0, size = rows.size(); i < size; i++) {
            List<Object> row = rows.get(i);
            System.out.println("row #" + i + " : " + row);
        }
    }
} finally {
    // 查询结束后，需确保调用ResultSet的close方法，以释放IO资源
    resultSet.close();
}
```

完整的代码示例

```
import com.aliyun.lindorm.tsdb.client.ClientOptions;
import com.aliyun.lindorm.tsdb.client.LindormTSDBClient;
import com.aliyun.lindorm.tsdb.client.LindormTSDBFactory;
import com.aliyun.lindorm.tsdb.client.exception.LindormTSDBException;
import com.aliyun.lindorm.tsdb.client.model.QueryResult;
import com.aliyun.lindorm.tsdb.client.model.Record;
import com.aliyun.lindorm.tsdb.client.model.ResultSet;
import com.aliyun.lindorm.tsdb.client.model.WriteResult;
import com.aliyun.lindorm.tsdb.client.utils.ExceptionUtils;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
public class QuickStart {
    public static void main(String[] args) {
        // 1.创建客户端实例
        String url = "http://ld-xxxx-proxy-tsdb-pub.lindorm.rds.aliyuncs.com:8242";
        // LindormTSDBClient线程安全，可以重复使用，无需频繁创建和销毁
        ClientOptions options = ClientOptions.newBuilder(url).build();
        LindormTSDBClient lindormTSDBClient = LindormTSDBFactory.connect(options);
        // 2.创建数据库demo和表sensor
        lindormTSDBClient.execute("CREATE DATABASE demo");
        lindormTSDBClient.execute("demo","CREATE TABLE sensor (device_id VARCHAR TAG,region VARCHAR TAG,time
BIGINT,temperature DOUBLE,humidity DOUBLE,PRIMARY KEY(device_id))");
        // 3.写入数据
        int numRecords = 10;
        List<Record> records = new ArrayList<>(numRecords);
        long currentTime = System.currentTimeMillis();
        for (int i = 0; i < numRecords; i++) {
            Record record = Record
                    .table("sensor")
                    .time(currentTime + i * 1000)
                    .tag("device_id", "F07A1260")
                    .tag("region", "north-cn")
                    .addField("temperature", 12.1 + i)
                    .addField("humidity", 45.0 + i)
                    .build();
            records.add(record);
        }
        CompletableFuture<WriteResult> future = lindormTSDBClient.write("demo", records);
```

```
        // 处理异步写入结果
        future.whenComplete((r, ex) -> {
            // 处理写入失败
            if (ex != null) {
                System.out.println("Failed to write.");
                Throwable throwable = ExceptionUtils.getRootCause(ex);
                if (throwable instanceof LindormTSDBException) {
                    LindormTSDBException e = (LindormTSDBException) throwable;
                    System.out.println("Caught an LindormTSDBException, which means your request made it to
Lindorm TSDB, "
                            + "but was rejected with an error response for some reason.");
                    System.out.println("Error Code: " + e.getCode());
                    System.out.println("SQL State:  " + e.getSqlstate());
                    System.out.println("Error Message: " + e.getMessage());
                }  else {
                    throwable.printStackTrace();
                }
            } else  {
                System.out.println("Write successfully.");
            }
        });
        // 这里作为示例，简单同步等待
        System.out.println(future.join());
        // 4.查询数据
        String sql = "select * from sensor limit 10";
        ResultSet resultSet = lindormTSDBClient.query("demo", sql);
        try {
            // 处理查询结果
            QueryResult result = null;
            // 查询结果使用分批的方式返回，默认每批1000行
            // 当resultSet的next()方法返回为null，表示已经读取完所有的查询结果
            while ((result = resultSet.next()) != null) {
                List<String> columns = result.getColumns();
                System.out.println("columns: " + columns);
                List<String> metadata = result.getMetadata();
                System.out.println("metadata: " + metadata);
                List<List<Object>> rows = result.getRows();
                for (int i = 0, size = rows.size(); i < size; i++) {
                    List<Object> row = rows.get(i);
                    System.out.println("row #" + i + " : " + row);
                }
            }
        } finally {
            // 查询结束后，需确保调用ResultSet的close方法，以释放IO资源
            resultSet.close();
        }
        lindormTSDBClient.shutdown();
    }
}
```

15.5. 搜索引擎

15.5.1. 通过开源Solr API访问搜索引擎

本文介绍通过开源Solr API（Java）访问Lindorm搜索引擎的步骤和示例说明。

前提条件

- 安装Java环境，使用JDK1.6或更高版本。
- 已将本机IP添加至白名单，具体操作，请参见设置白名单。

步骤一：下载solr-solrj

您可以通过添加Maven依赖或者直接下载solr-solrj来获取开发工具包。

- 添加Maven依赖获取。如果您使用Maven仓库做项目，请在*pom.xml*文件中添加solr-solrj依赖，具体代码如下。

```
<dependency>
   <groupId>org.apache.solr</groupId>
   <artifactId>solr-solrj</artifactId>
   <version>7.3.1</version>
</dependency>
```

- 直接下载solr-solrj。

步骤二：创建实例并初始化

- 如果您使用公网访问实例建议采用HttpSolrClient模式，代码如下所示。

```
    String httpUrl = "http://***:8983/solr/";
    HttpSolrClient solrClient = new HttpSolrClient.Builder(httpUrl).build();
```

- 如果您使用内网访问实例建议采用CloudSolrClient模式，代码如下所示。

```
    String zkHost = "zk1:2181,zk2:2181,zk3:2181/solr"
    CloudSolrClient solrClient = new CloudSolrClient.Builder(Collections.singletonList(zkHost),
                                 Optional.empty()).build();
```

> **说明** CloudSolrClient是线程安全的，应用多线程可以共享一个对象。

步骤三：访问搜索引擎示例说明

1. 通过以下代码写入数据。

```
    List docs = new ArrayList<>();
    SolrInputDocument doc = new SolrInputDocument();
    doc.addField("id", 1);
    doc.addField("name_s", "bob");
    doc.addField("age_i", 18);
    docs.add(doc);
    solrClient.add(docs);
```

2. 通过以下代码查询数据。

```
    SolrQuery solrQuery = new SolrQuery("name_s:bob");
    QueryResponse response = solrClient.query("your_index", solrQuery);
    SolrDocumentList documentList = response.getResults();
    for(SolrDocument doc : documentList){
        String id = (String)doc.getFieldValue("id");
        //do something
    }
    solrClient.close();
```

> **说明** 关于常见的查询数据代码示例，请参见代码示例。

16.行业案例

16.1. 车联网

16.1.1. 上海新能源汽车车辆基础数据

方案亮点：PB级新能源车辆基础数据、高性能采集入库、高效数据分析、低成本运营

客户感言

2019年开始上海市新能源汽车大数据平台从自建Hadoop集群迁移至阿里云Lindorm+DLA Spark产品，有效解决了我们平台存储和计算的横向动态扩容瓶颈，同时借助其产品中间件LTS实现了我们平台数据的冷热分离，有效降低了数据存储成本，依托于阿里云强大丰富的技术生态解决了我们的诸多技术壁垒，使得我们的技术团队专注于业务开发。

客户简介

上海市新能源汽车公共数据采集与监测研究中心（以下简称“数据中心”）是2014年底由上海市社会团体管理局批准注册成立，由上海市经济和信息化委员会负责业务指导，是全国首个、上海市唯一的新能源汽车市级监管平台，其主要职责是对上海市推广的全部新能源汽车进行数据采集和分析应用，以支撑政府政策制订和安全监管。截至2021年1月31日，数据中心累计接入新能源汽车41.8万辆，涉及车企95家，品牌107个，车型777款，数据存储量突破1个PB，数据规模继续处于全球城市前列。成立以来，数据中心在多源数据融合应用方面做了许多探索，陆续建成了 “上海市新能源汽车大数据平台”、“上海市动力电池溯源管理平台”、“上海市加氢站与氢燃料电池汽车公共数据平台”、“GEF6上海能源管理中心平台”，有效发挥了数据在车辆安全监管、动力电池全生命周期监管、燃料电池汽车补贴发放等方面工作的支撑作用。

业务挑战

- 国家政策支持，保有车辆数量快速增长。
- 电动车作为一个新生事物，还处于不断发展中，数据采集点有不断变化的需求。
- 为满足分析及上层业务诉求，采集频率也有不断变化的需求，采集频率的提高往往意味着吞吐量、数据量翻倍甚至数量级提升。
- 国家对于电动车数据的保存年限做了规范性要求。
- 海量采集数据有实时归档到离线数仓案并分析的需求。
- 数据分析结果有服务化的需求，需要回流到在线存储。

解决方案

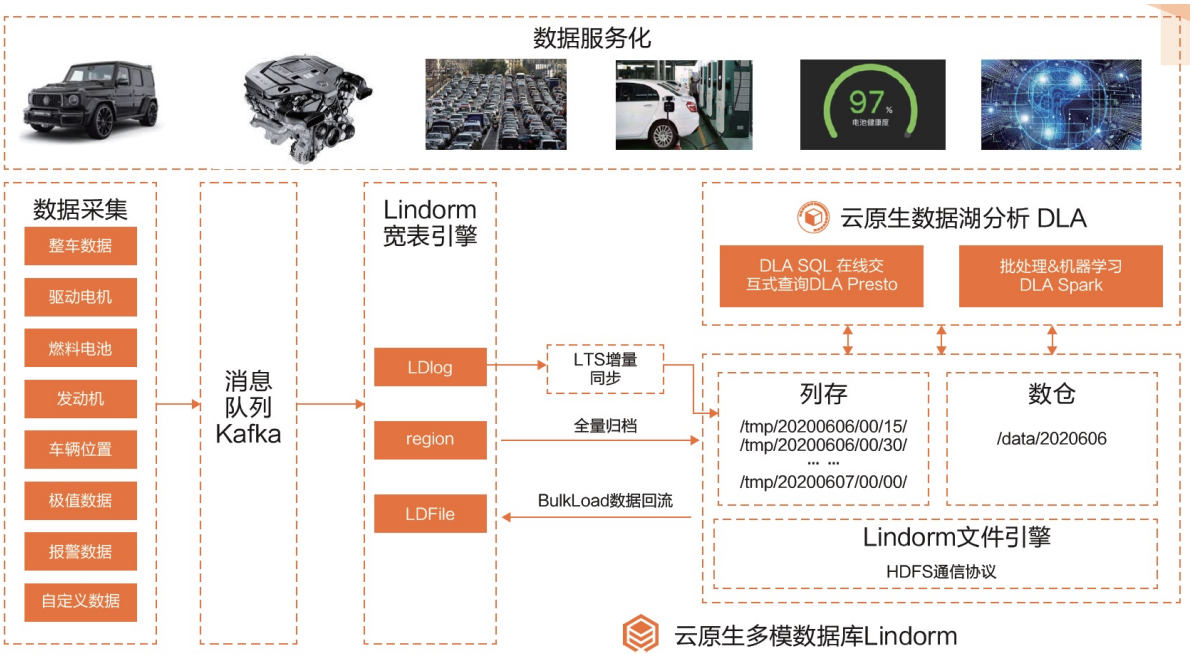
- Lindorm历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠，构建起坚实的离在线存储底座，使得客户可以聚焦于业务侧的发展。
- Lindorm宽表引擎批量组提交优化，大幅度提升集群吞吐量、降低请求响应时间，性能提升3倍以上。



某应用上线该特性后，batch 1000行的性能提升23倍

- 开启Lindorm宽表引擎压缩优化特性，显著减少存储空间开销。

- 基于Lindorm HDFS大数据存储方案，满足海量数据ETL和分析需求。



- 云原生数据湖分析（简称DLA）Spark分析引擎满足在线交互式查询、流处理、批处理、机器学习等业务诉求。

客户价值

- Lindorm宽表引擎批量写入、高效压缩，线性扩展等特性使得数据采集入库性能更高、成本更低，很好的支撑了业务的快速发展，对于采集点&采集频率变更带来的流量突增也可以从容应对。
- 通过APP->Lindorm->LTS实时归档->parquet列存(Lindorm HDFS)->DLA Spark分析->bulkload->Lindorm这样一条链路形成了数据存储、实时归档、分析、数据回流、分析后数据查询的全链路数据闭环，满足业务发展的诉求。

16.1.2. 某网约车公司车辆轨迹数据

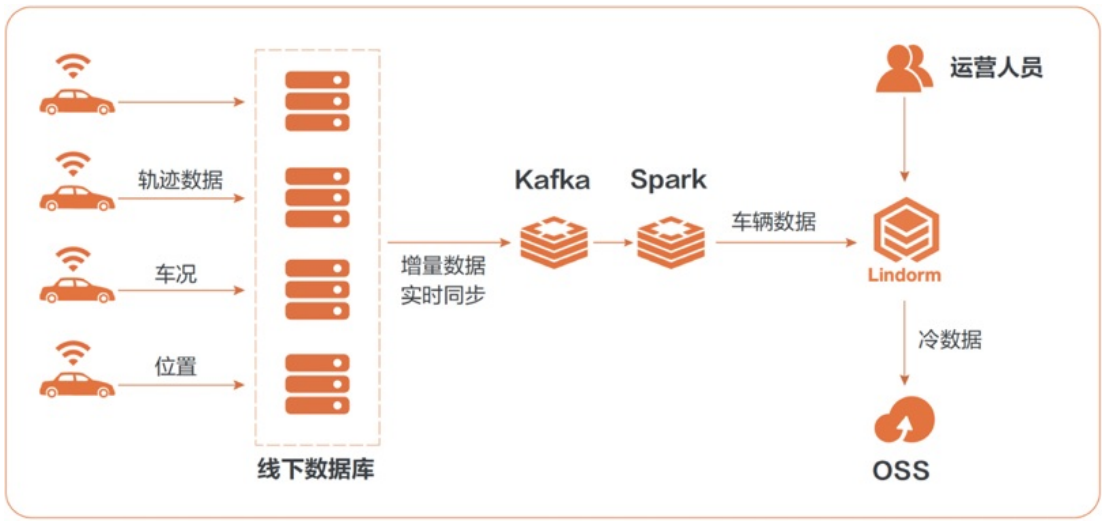
随着车辆数量的快速增长，每辆车每分钟上传轨迹、位置等信息到MySQL导致MySQL库数据量过大，将全量数据写到云原生多模数据库Lindorm中可以解决数据量过多和成本问题。

业务挑战

- 目前线下有7万+辆车，车辆数量在快速增长，每辆车每分钟上传轨迹、位置等信息到MySQL导致MySQL库数据量过大，直接影响了在线查询业务。
- 车辆数据需要存储3年，日增数据量100 GB，3年总数据量超100 TB，数据存储成本高昂。

解决方案

- 在原有的架构上引入了阿里云数据库，并将全量数据写到云原生多模数据库Lindorm中，同时增量数据通过Kafka、Spark等实时同步到云原生多模数据库Lindorm，这样解决了客户线下数据库数据过大的问题。
- 客户需要存储最近3年的数据但是最近一个月的数据访问很频繁。云原生多模数据库Lindorm支持OSS做为冷存储，这样业务上把一个月之前的数据（冷数据）全部通过Lindorm自动存储到OSS里面，最近一个月内的数据存储到高效云盘，这样在零代码开发、零运维、客户无感知情况下进行了冷热分层存储，大大节约了客户的存储成本。
- 运营人员可以通过云原生多模数据库Lindorm提供的SQL组件进行一些日常的运维工作，例如查找某车的位置。



客户价值

- 通过将车联网的数据存储到云原生多模数据库Lindorm，存储成本问题得到解决。
- 近几年线下的车辆还不断增加，但是由于云原生多模数据库Lindorm是一种分布式的数据库，所以车辆的数据增加不需要再变动已有的架构。

16.1.3. 某新能源汽车造车新势力

方案亮点：车联网、高并发、高吞吐、存储计算分离、极致弹性、成本下降60%

客户感言

阿里云云数据库Lindorm相比于其他存储引擎和HBase开源版本，提供了更高效的压缩率和同规格下的更大吞吐量，能稳定的支撑高并发、海量数据存储的业务需求。同时提供了成熟的冷热分离方案，大大降低了存储和运维成本，对于不断增大的业务量，也能进行平稳的横向扩容。我们也将基于云数据库Lindorm构建实时存储方案，为售后和用户提供更高效实时的数据读写能力。

业务背景

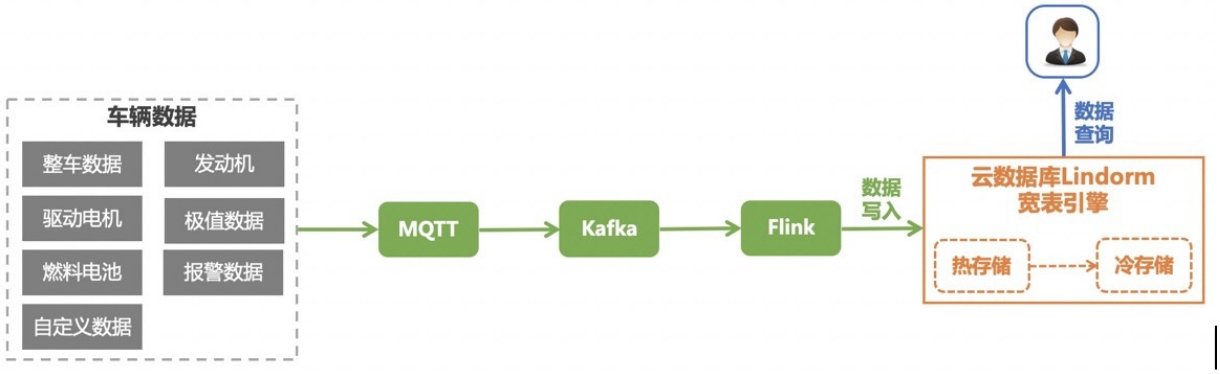
在碳中和、碳达峰这样的大背景下，国家大力支持新能源汽车的推广使用，但同时监管部门也提出了相应的要求，需要实时存储车辆状态数据，进行故障告警及数据的实时查看。为了能及时响应车辆告警信息，排查车辆故障，需要实时能下载、存储车辆信号数据并进行技术分析。

业务挑战

- 当前新能源汽车得到国家政策支持，处于高速发展时期，车辆交付数量快速增长。同时，数据采集的需求不断增长，近半年来集群写入流量已经翻倍，因此需要存储层具备优异的弹性和线性扩展能力，以适应车辆数量，采集点，以及采集频率的变化。
- 业务写入请求高并发，当前峰值写请求近4w/s，对写入吞吐有较高的要求。
- 高并发的写入带来大量的IO，每秒写入数据量超过220MB，导致资源开销高，成本投入大，需要降低成本。
- 海量数据需要高效、低成本、实时存储，并能满足业务的实时查询和监控需求，要求写入、查询响应低时延、高可靠。

解决方案

- 采用云数据库Lindorm后的数据链路。



- 采用云数据库Lindorm作为底层存储，得益于其存储计算分离架构，从而具备了极致的弹性伸缩能力，从容应对业务的快速增长或由于指标采集变更带来的业务峰值变化。
- 使用ZSTD压缩算法，大幅度提升了压缩比，相比开源版本的HBase存储空间大幅度下降，可大幅降低存储成本。
- 使用Lindorm专业优化后的批量组提交特性，集群吞吐量明显提升，写入响应时间明显下降，相比开源HBase提升3倍以上。
- 通过采用Lindorm提供的集群内单表冷热分离特性，不修改代码的情况下即可实现冷数据的自动归档，从而降低存储成本。
- Lindorm历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠，构建起坚实的离在线存储底座，使得客户可以聚焦于业务侧的发展。

客户价值

- 云数据库Lindorm历经阿里众多核心业务10年以上双十一业务峰值考验。并由拥有国内相关技术领域的顶尖技术团队提供全托管的运维支持，降低客户运维成本的同时有力的保障了客户业务的持续稳定、可靠。
- 云数据库Lindorm原生的存储计算分离架构、极致的弹性伸缩能力、简洁友好的交互体验可以轻松实现集群的线性扩展。
- 云数据库Lindorm提供了丰富的监控指标，更好的帮助客户做出扩缩容的决策。
- 通过合理的使用Lindorm的冷热分离、压缩算法，实现运行成本下降60%。

16.2. 航运物流

16.2.1. 申通快递迁移Oracle到Lindorm

方案亮点：去Oracle、云Lindorm宽表引擎 +LTS+搜索引擎一体化架构

客户感言

物流订单、巴枪等业务为非强事务要求的业务，最初选择Oracle是出于性能、稳定性的考虑，但基于菜鸟使用Lindorm，我们对Lindorm宽表引擎+搜索引擎经过充分测试后发现Lindorm的线性扩展能力更适合快递业务，0代码0迁移即可轻松应对快速业务增长及双十一大促，并且成本相比Oracle大幅下降。经过近2年的运行，Lindorm的稳定性也完全符合我们的高可用要求，给Lindorm点赞。

客户简介

申通快递品牌初创于1993年，公司致力于民族品牌的建设和发展，不断完善终端网络、中转运输网络和信息网络三网一体的立体运行体系，立足传统快递业务，全面进入电子商务领域，以专业的服务和严格的质量管理推动中国快递行业的发展。经过二十余年的发展，申通快递在全国范围内形成了完善、流畅的自营快递网络。截至2021年1月，公司拥有独立网点及分公司超4500家，服务网点及门店25000余个，从业人员超过30万人。

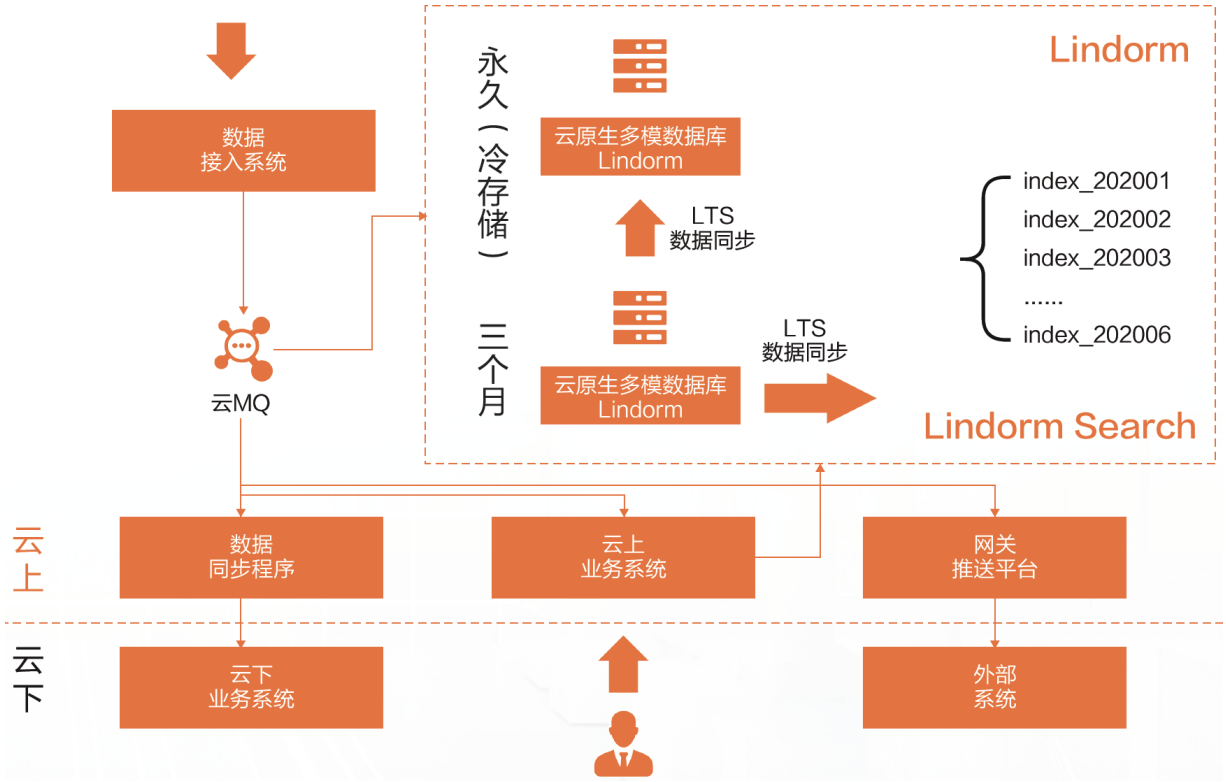
业务挑战

- 目前客户的巴枪、订单、分单等业务每天都产生大量的数据，巴枪业务数据量数亿每天，订单&分单数据量数千万每天，总数据量超百TB。
- 客户业务有面向不同场景的查询需求，既有根据订单号的点查，也有其他多种不同维度的范围甚至模糊查询的需求。
- 每年双十一大促客户业务都会面临大幅度上涨。

解决方案

- Lindorm历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠。
- Lindorm为云原生多模数据库，可同时支持宽表引擎和搜索引擎，并且可通过Lindorm Tunnel Service（LTS）实时将写入宽表引擎的数据同步至搜索引擎，并可保证搜索引擎和宽表引擎数据的一致性。

- Lindorm具备动态升降配、节点扩缩容的能力，轻松应对客户双十一大促的极致弹性需求。
- Lindorm采用存储计算分离架构，可以按需扩容存储或计算节点，扩容和缩容操作对应用透明，业务无需任何代码改动，数据和业务请求自动均衡，零运维。
- 系统架构如下



客户价值

- Lindorm多存储介质支持、高压缩比、完全线性扩展能力，成功帮助客户完成去Oracle，实现可扩展、低成本、高性能上云。
- Lindorm宽表引擎配合搜索引擎完美的满足了业务对于订单、运单、分单的快速点查及多维度范围和模糊查找，实现了宽表引擎支撑10万包裹秒的高速查询，性能较原始方案提升5倍；搜索引擎支持上万网点的随机多维查询，并支持结果集的实时导出下载功能。
- 通过LTS实现宽表引擎向搜索引擎的实时、高效，并且保证双引擎数据一致性，无需业务系统双写并保证数据一致性。
- Lindorm云原生分布式及存储计算分离架构，具备极致弹性伸缩能力，助力客户轻松应对双十一大促业务峰值。

16.2.2. 亿海蓝-航运大数据的可视化案例

方案亮点：千亿级船舶轨迹时空分析、多元时空数据整合、开发周期短

客户感言

“阿里Ganos产品很好地提升了我们对异构时空数据的处理能力。Lindorm高性能、低延迟、高吞吐、稳定性让我们节省了不少运维投入，更多精力投入到业务实现上。” ——亿海蓝CTO

客户简介

亿海蓝是国内领先的航运大数据公司，做为国内最大的AIS数据服务运营商，发展目标是通过大数据技术推动全球航运物流与互联网的融合，加速航运产业转型升级。亿海蓝目前已为全球上百万行业用户提供数据服务，客户涵盖港口、船公司、物流、金融、保险和设备制造商等多类群体。其中，航运数据可视化分析系统的核心功能，是对用户关心的航运数据进行分析和报告输出。输出形式主要是热力图和统计指标。示例使用场景如下：

- 大宗航线繁忙度分析
- 铁矿石LNG等货种航线分布
- 港口业务强度分析
- 内河横向截面的多航道通行量研究
- 区域内船舶周期性活动强度研究

- 对政策面影响区域业务活动强度的研究

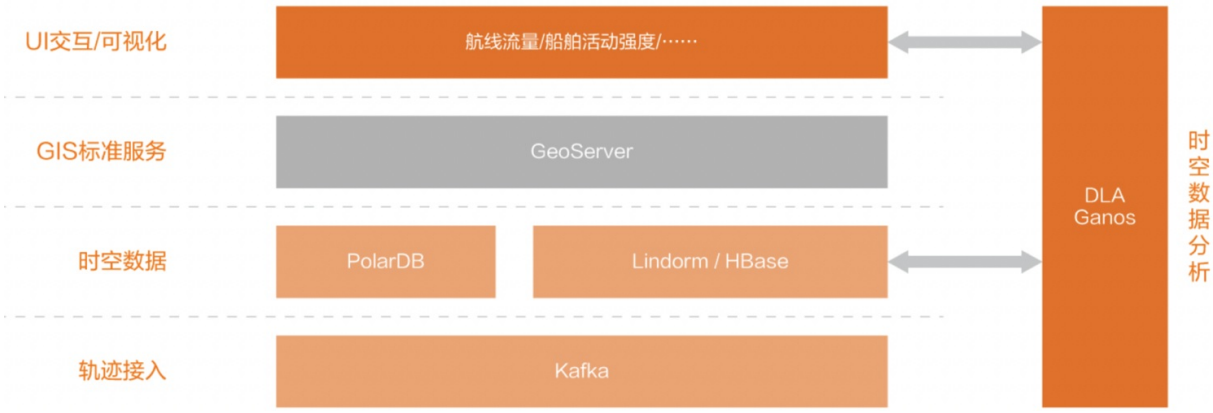
业务挑战

- 数据访问要求高效性。一旦数据查询性能低，可视化体验就会受到大幅影响。
- 数据量大，数据提取周期长。亿海蓝有17年AIS（船舶自动识别系统）数据积累，每日AIS数据增量约10亿。用户需要对历史年份数据进行分析，时间跨度大。
- 业务实现技术复杂度高。业务涉及数据种类多，亿海蓝对这些业务数据使用了多种存储引擎。

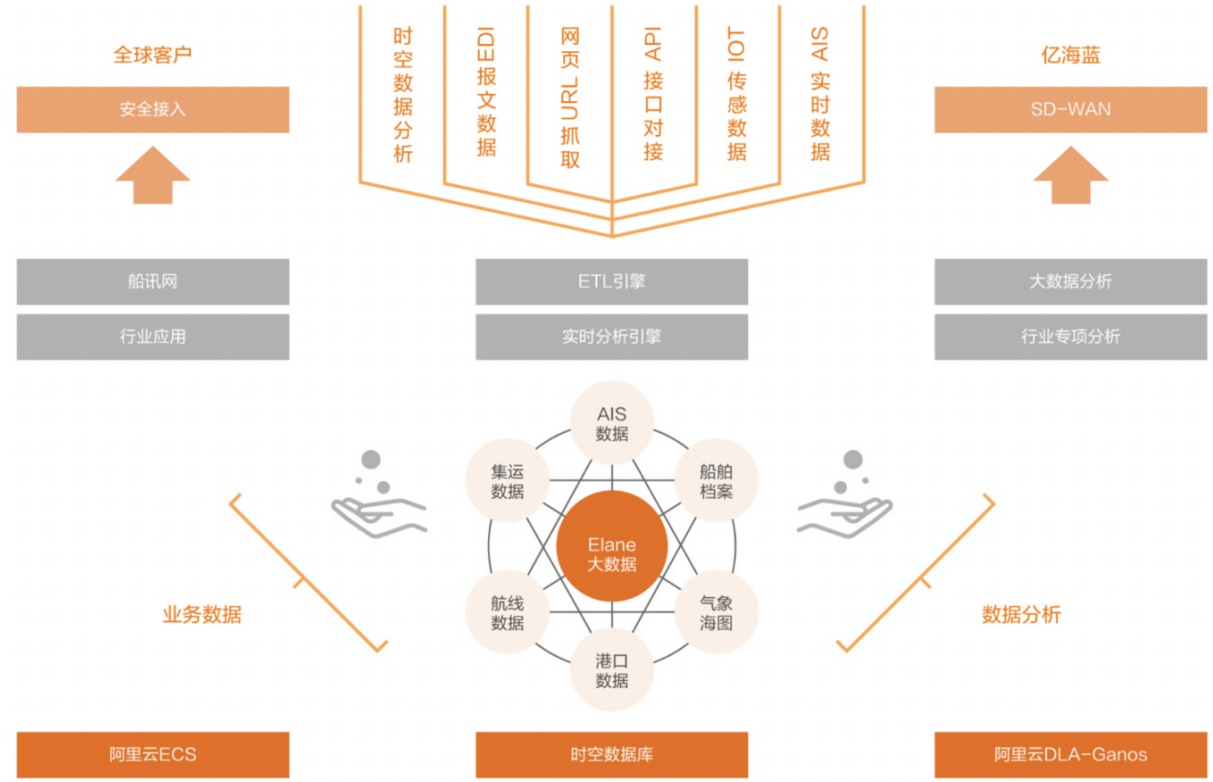
解决方案

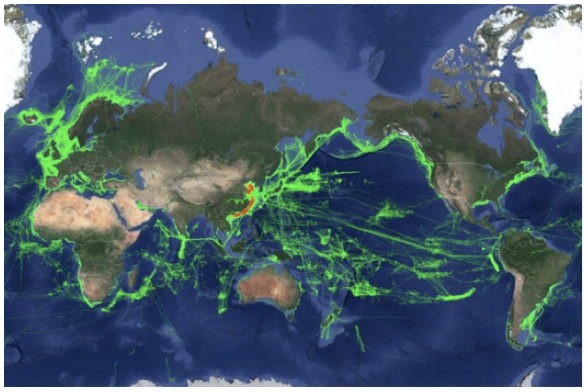
- DLA Ganos 引擎对不同时空存储的存取进行了统一封装，降低编码难度。
- 原始数据使用DLA Ganos进行高效预处理，降低数据提取和结果数据再落地时间。
- 数据抽取结果存储到Ganos on Lindorm中，Lindorm的低延迟、高性能、高吞吐查询能力保障业务能流畅显示。

整体系统架构方案如下图所示：

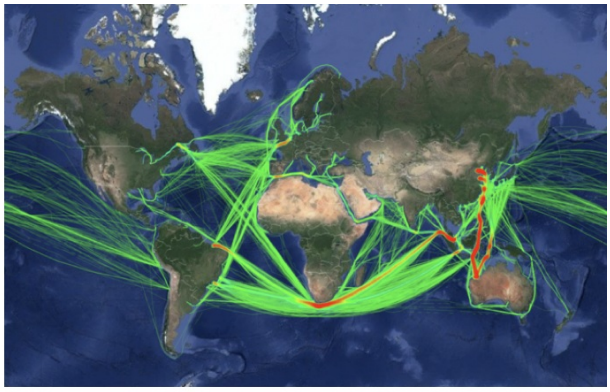


航运数据可视化分析系统架构图如下：

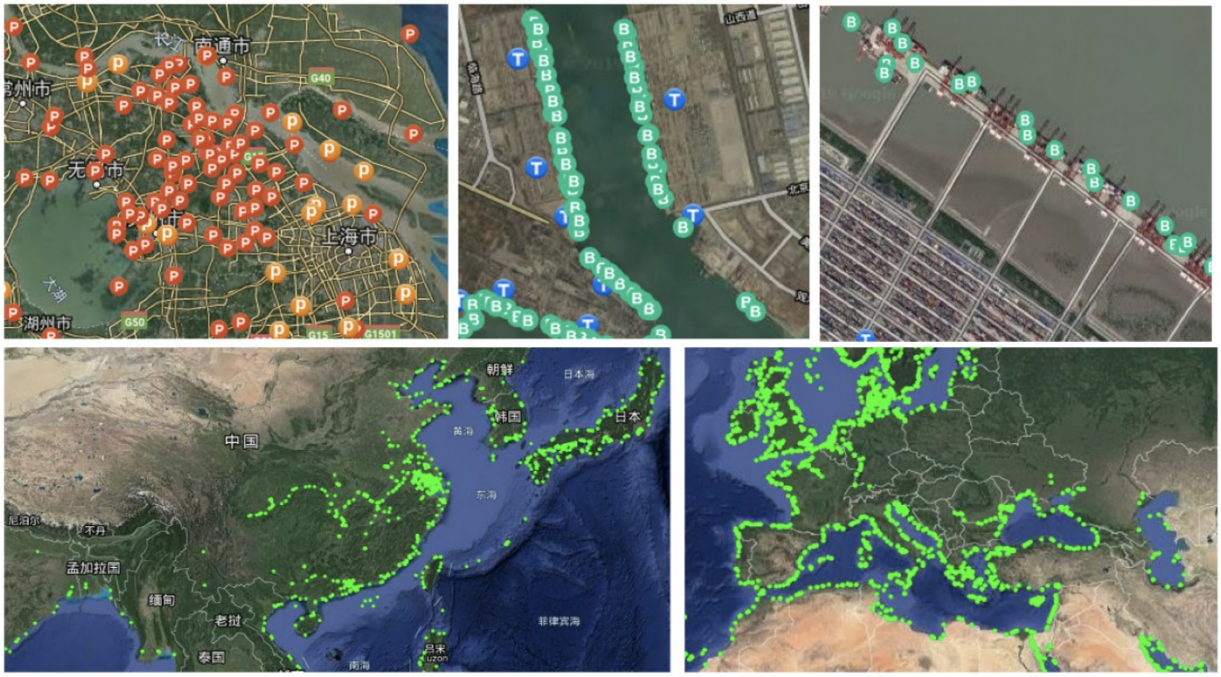




航运数据可视化-2019 全球渔船轨迹热力图



航运数据可视化-2019 全球铁矿石热力图



基础数据截图:港口泊位分布

业务价值

- 通过阿里云产品的支持，缩短了业务的技术实现周期，降低了运维投入。
- Ganos on Lindorm高性能查询指标满足了业务上低延迟流畅显示的要求。
- 不同于轨迹数据，航线数据业务性更强，需要更丰富的存储和查询方式。DLA Ganos 满足了整合分析多种时空数据的需求。
- 航海业务相对复杂，需要对数据做一些自定义的运算。DLA Ganos内置Spark集群，提供定制处理数据的计算引擎，节省了单独购买Spark的开支。
- 提供金字塔式的矢量快显数据存储，优化了热力图缩放效果。

16.3. 医疗健康

16.3.1. 某国内领先的健康体检公司迁移SQL Server到Lindorm

方案亮点：成本显著下降情况下满足C端客户毫秒级实时在线查询

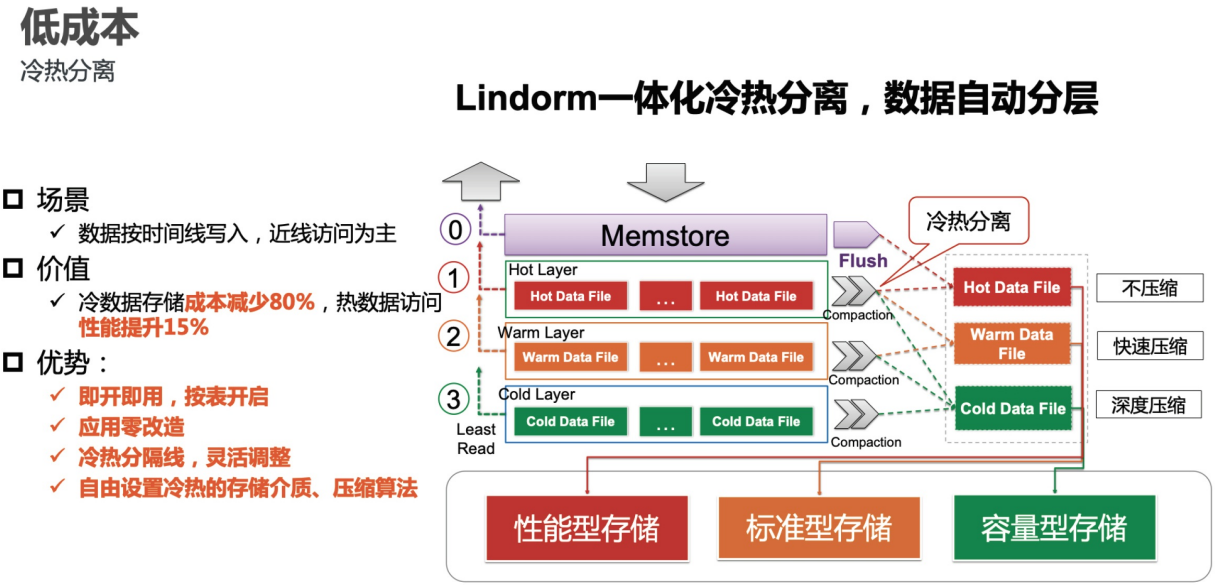
业务挑战

- 健康体检报告数据要求保留周期长达15年，数据量达数十TB，存储成本高昂
- 面向C端实时查询，系统可用性要求高

- 百亿数据量低延时实时查询，线性扩展能力及请求稳定性要求高

解决方案

- Lindorm历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠。
- 通过专利内存管理技术CCSMap和Bucket Cache，结合AlijDK深度定制ZGC算法，实现GC停顿小于5ms，更好的满足实时在线查询。
- 开启云原生多模数据库Lindorm压缩优化特性，显著减少存储空间开销。
- 借助云原生多模数据库Lindorm集群冷热分离功能，0应用改造实现单表内数据冷热分离，降低存储成本。



客户价值

- 业务无事务要求，实现从SQL Server迁移至Lindorm，满足C端客户百亿级别海量数据下，单行或多行高性能、实时查询。
- 在满足业务需求的情况，开启Lindorm压缩优化、集群内单表冷热分离能力使得整体成本显著下降。
- 利用Lindorm数据通道LTS(Lindorm Tunnel Service)，实现从MaxCompute高效导入数据，同时保证在线查询的稳定。

16.4. 广告行业

16.4.1. 某国际智能营销服务公司广告投放数据迁移至阿里云Lindorm

方案亮点：数据保留周期延长4倍、成本下降86%

客户感言

迁移到云数据库Lindorm后，数据压缩优化、冷热存储分离实现了成本的显著降低。而且云数据库Lindorm在更少节点的情况下实现了更高的性能、稳定性，更低的写入延迟。我们已经着手下一个业务的迁移上云！

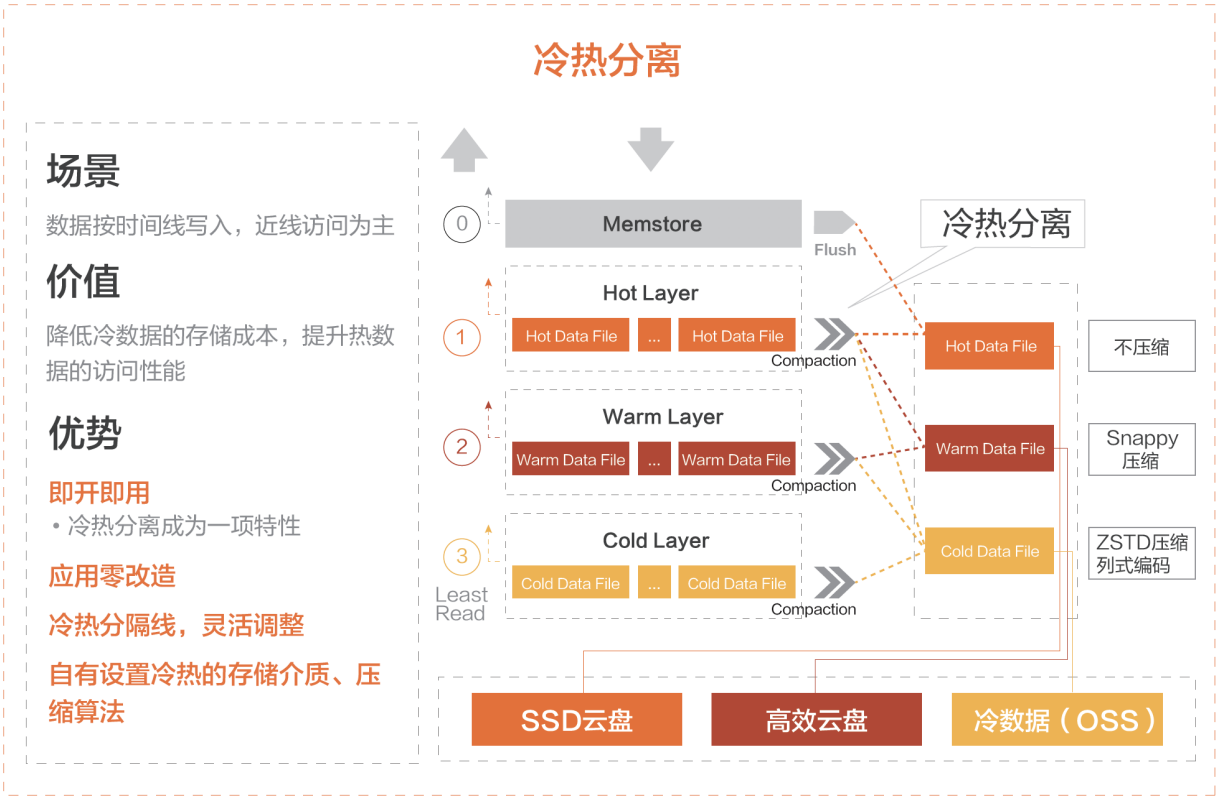
业务挑战

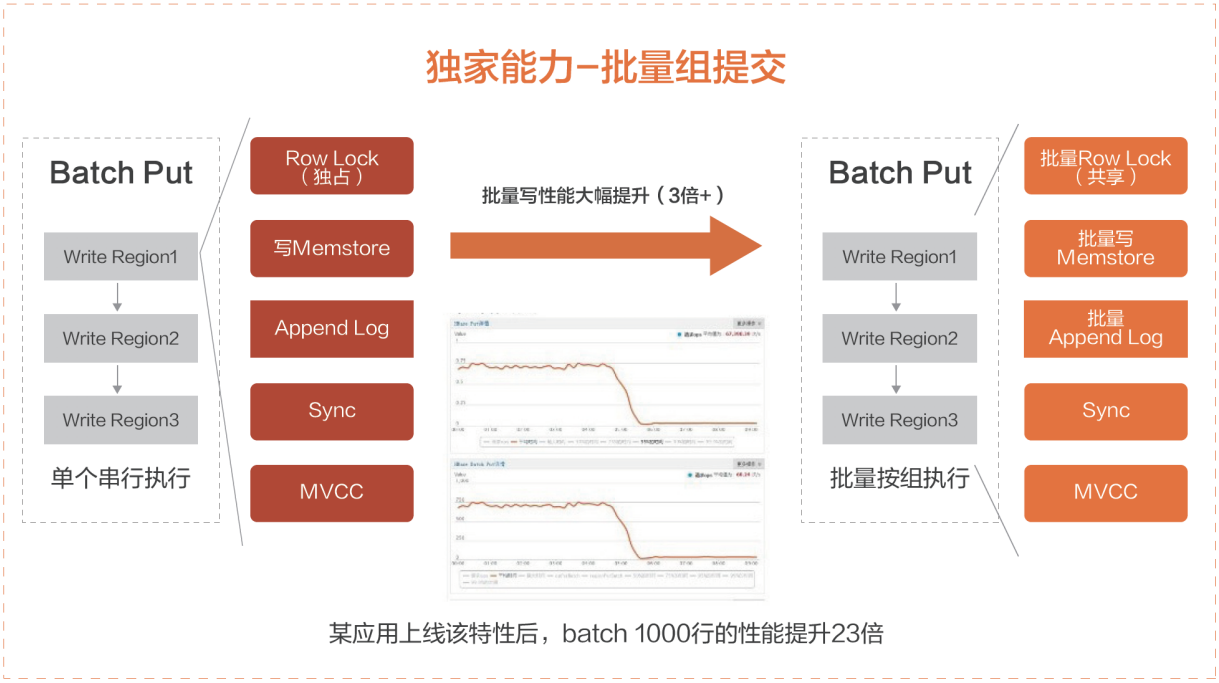
- 开源版HBase集群的故障恢复能力差，而该业务为广告投放核心链路，一旦发生集群故障将导致业务出现几乎波及全部用户的故障，严重影响营收。
- 集群读写请求量、数据增量大，日增数据量高达4TB。
- 集群计算资源、存储资源开销高昂，为了降低成本，数据仅能保留7天，导致部分业务需求无法满足。

解决方案

- 云数据库Lindorm 100%兼容开源HBase并深度扩展，历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠。
- 开启云数据库Lindorm压缩优化特性，通过压缩优化数据占用空间，降低存储开销。

- 借助云数据库Lindorm冷热分离功能，0应用改造实现单集群单表内数据冷热分离，降低数据存储成本，将数据保留时间延长至1个月。
- 开启云数据库Lindorm批量组提交优化，大幅度提升集群吞吐量、降低请求响应时间，实现3倍以上性能提升。
- 针对该广告业务场景提供有效的性能优化方案，避免不必要的increment操作，降低资源开销。





某应用上线该特性后，batch 1000行的性能提升23倍

客户价值

- 从开源版HBase迁移到云数据库Lindorm后，稳定性得到大幅提升，同时阿里云提供了全托管、免运维及SLA保障，专家团队的免费技术支持，使客户能聚焦业务侧发展。
- 云数据库Lindorm的专家服务提供了有效的、针对业务场景的优化方案，降低集群资源开销，提升性能

- 实现成本数量级下降，仅用不足七分之一的成本便获得了迁移前4倍的数据保留周期。

16.4.2. 某互动广告运营商实时竞价数据迁移至阿里云Lindorm

从IDC自建到云数据库HBase标准版，再到云原生多模数据库Lindorm，满足更少的资源投入同时降低99.9%请求返回的最大时间。

业务挑战

- 实时竞价（RTB）不同于传统的广告投放模式，RTB对广告曝光提交竞价请求的延迟通常小于100ms，超出意味着无法成功提交竞价请求，对业务而言意味着资损。
- 为达到精准投放，RTB需要高并发访问海量用户的画像数据。
- RTB需要实时采集并存储用户在不同广告的行为数据，并统计出不同维度不同指标的统计结果，实现投放效果的实时反馈并完成投放策略优化。

解决方案

- 从 IDC 自建 HBase 迁移云 HBase 标准版，实现更少的资源投入，P999 毛刺（99.9%请求返回的最大时间）初步达到客户要求。
- 从云 HBase 标准版迁移到同规格云数据库Lindorm，实现 P999 毛刺10倍下降。

核心技术

独家能力-10倍GC优化

GC 停顿时间

10ms 云HBase

100ms 社区HBase

三大核心技术

一.CCSMap:无锁高并发无对象Skiplist

核心设计

- cacheline友好
- object-free
- 内存换需申请
- 无锁高并发
- slab复用

多维能力

- 吞吐提升20%
- 内存效率提升40%
- YGC时间优化4倍

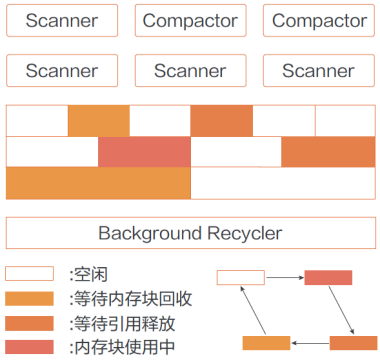
高度复用

- 兼容JDK Con-currentMap接口

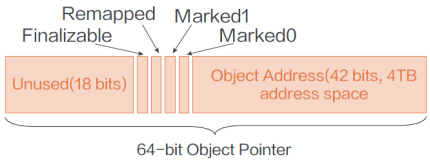
技术领先

- 业界首创，贡献至开源社区，HBase3.0核心特性

二.Shared BucketCache:内存块引用共享



三.AJDK ZGC:单个毫秒暂停的垃圾收集器



核心技术

独家能力-LLC 日志三副本

原HDFS Pipeline缺

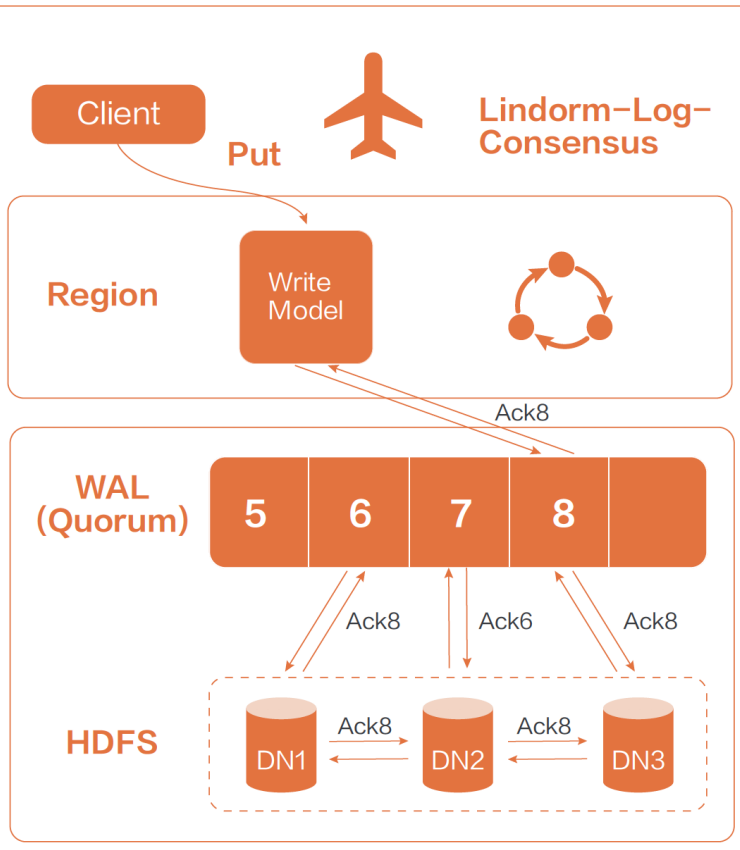
- 链路长，3个 Round Trip
- 网络、磁盘抖动，产生毛刺
- 坏盘，宕机故障恢复时间长

LLC核心设计

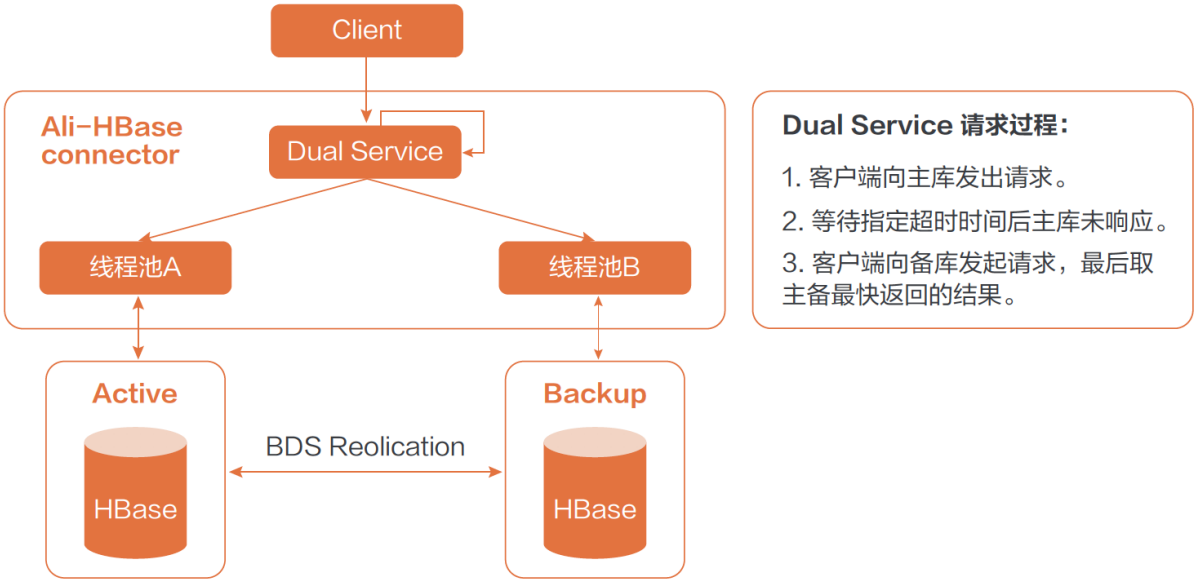
- 写3等2的 Quorum 机制，屏蔽坏、慢节点
- 慢节点摘除，节点落后较多时重选 Quorum Group
- 黑名单管理，避免再次选中坏、慢节点
- 故障恢复，通过 Recover Lease 确定文件长度

使用LLC替换Pipeline

- 平均写入 RT 减少 50%
- P99 延迟下降至 1/6
- 抖动、坏盘、宕机无感知



- 高可用方案：同城主备、同城双活（Dual Service）。Dual Service架构如下：



客户价值

- 从IDC自建到云HBase标准版，云HBase在降低用户资源投入的同时提供免运维及 SLA 保障，并有专家团队的免费技术支持，使客户能聚焦业务侧发展。
- 迁移到云Lindorm，使得业务请求RT更加稳定，实现P999毛刺10倍下降，有效提升客户营收。
- 同城主备，使得广告核心具备了故障情况下的一键切换能力。
- Dual Service 技术可有效利用备集群资源，进一步提升请求稳定性。

16.4.3. 某全球领先媒介投资管理公司迁移Bigtable到Lindorm

方案亮点：P99毛刺低于5ms、跨可用区容灾、 灵活的数据生命周期支持

客户感言

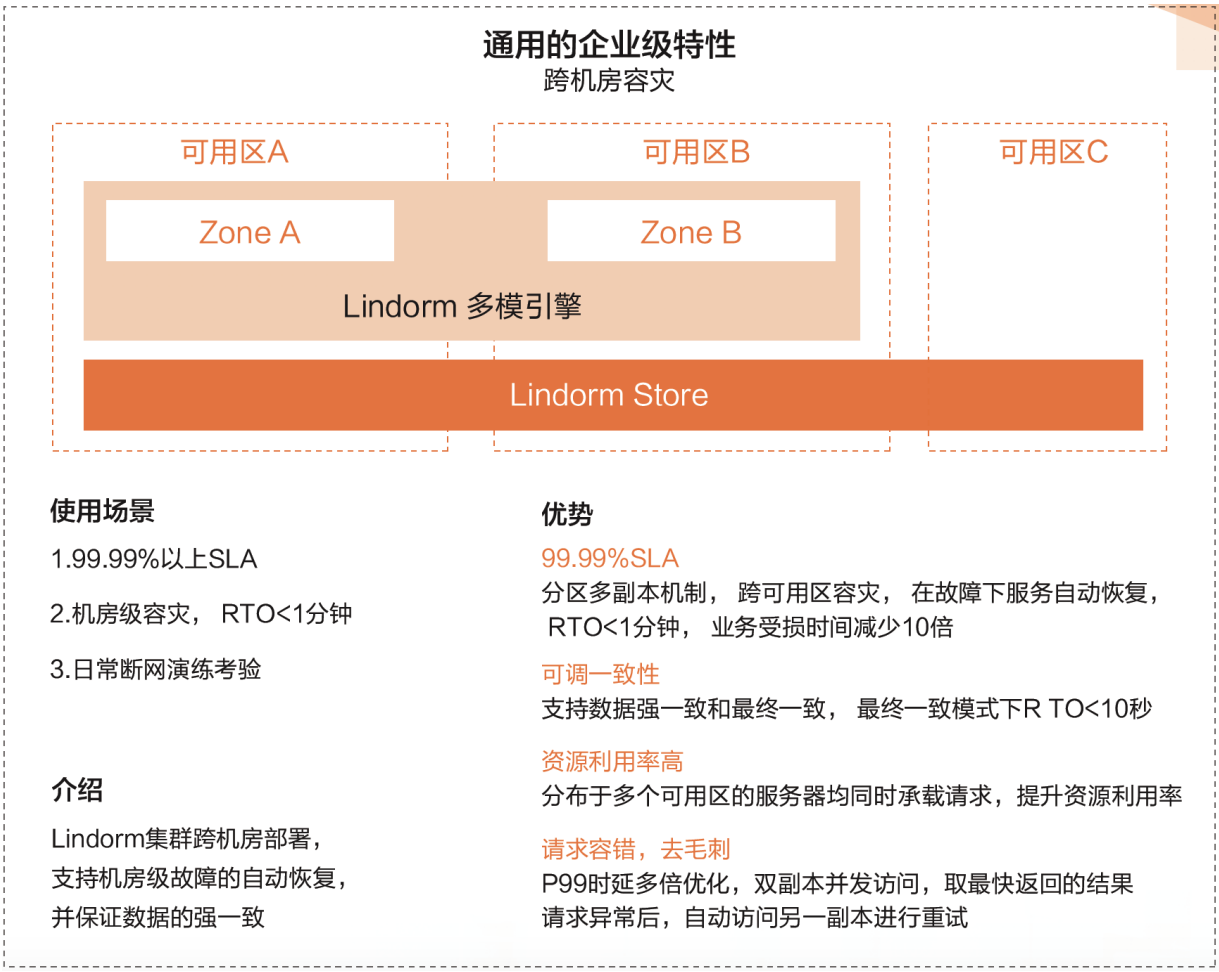
之所以选择lindorm基于两方面的考虑。从产品能力来讲，Lindorm相比Google BigTable表现更好，特别是请求的稳定性方面，请求毛刺率有数量级差异。从服务角度讲，Lindorm团队具备非常强的服务意识，虽然我们base在美国，但总是能第一时间得到支持，而且技术实力也非常强，问题总是能得到快速解决。

业务挑战

- 10TB以上数据量，单个KV最大达到MB级别。
- 需要支持表级、列簇级、行级、Cell级等多级数据的生命周期管理方案。
- 延迟响应敏感，P99响应时间要求低于10ms。
- 多可用容灾强一致，SLA要求高。

解决方案

- 云原生多模数据库Lindorm跨可用区高可用容灾能力，提供机房级别故障强一致容灾（RTO 60s, RPO = 0）以及最终一致容灾能力（RTO < 10s， RPO < 1s），保障用户业务持续在线。



- 通过专利内存管理技术CCSMap和Bucket Cache， 结合AlijDK深度定制ZGC算法，实现GC停顿小于5ms。吞吐提升20%，内存效率提升40%，相比开源方案毛刺优化20倍以上。
- 支持表级，列簇级，行级，Cell级等灵活的生命周期管理方案，便于根据场景定制数据的生命周期。

客户价值

- Couchbase迁移云原生多模数据库Lindorm，同样满足业务需求请求稳定性的情况下，整体使用成本显著下降。
- 利用云原生多模数据库Lindorm多机房容灾能力，提供全年99.99%的SLA保障。
- 利用云原生多模数据库Lindorm数据通道BDS，实现数据实时备份，增量异构道出的能力。

16.5. 金融行业

16.5.1. 收钱吧订单查询&搜索

方案亮点：百TB级数据，毫秒级实时查询、高维度随机组合，亚秒级订单搜索

客户感言

使用云数据库Lindorm集群和服务将近一年，在方案制定和实施过程中，阿里云数据库专家们为我们提供了大量珍贵的建议以及实战经验。遇到任何问题都能第一时间得到耐心细致的解答。系统正式上线以来，一直平稳运行，未有任何波动。云数据库Lindorm提供了海量的数据存储，百亿数据毫秒级的实时查询性能，高级专家的耐心解答和护航，每年节约近百万元成本。

客户简介

收钱吧，隶属于上海喔噻互联网科技有限公司，是中国移动支付服务商领军者，致力于用网络和数据的力量服务线下实体商家。收钱吧不仅为商家提供专业移动支付收款工具，同时也是为商家提供金融、广告、营销管理、供应链等多种服务的生意帮手。2014年12月，收钱吧正式上线，开创了中国移动支付市场“一站式收款”时代，并成功研发了“收钱吧扫码王”等全场景智能收款设备，产品获得多项国家专利。目前收钱吧服务超过330万商家，日服务3000万人次。

业务挑战

- 业务涉及对象种类繁多，需要支持15个查询维度随机组合查询。
- 数据量庞大，查询响应时间要求亚秒级。
- 每日新增数据量至少75GB，当前总数据量百TB级别，同时由于移动支付的特殊性，数据需要永久保存，存储成本高昂。

解决方案

- 云数据库Lindorm全文索引方案，通过LTS实现Lindorm宽表引擎与Lindorm搜索引擎之间的数据实时同步，使业务轻松应对高维度&随机组合查询需求。

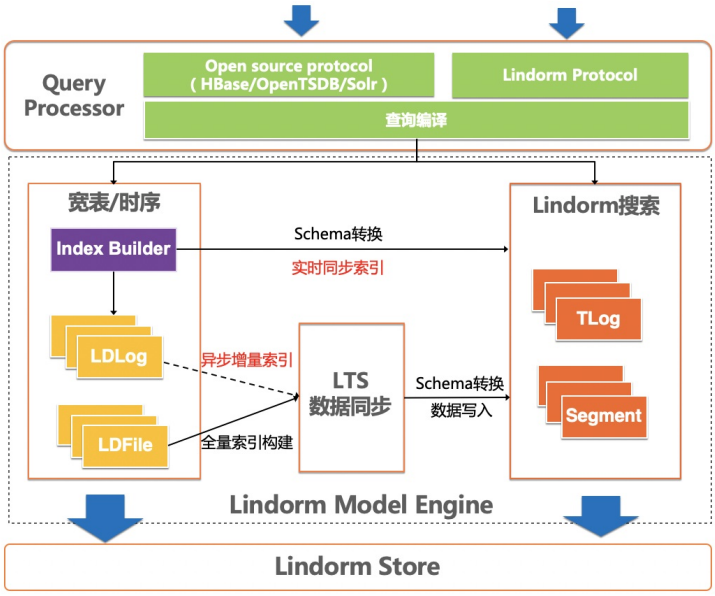
加速**宽表/时序引擎**查询，提供全维检索能力

统一API，简单易用

- 系统自动维护索引，应用开发不感知索引表
- 索引支持非冗余、冗余部分列、全冗余
- 查询时基于编译优化(RBO)自动路由到搜索引擎，并智能判断是否需要回查宽表/时序引擎

数据自动同步到搜索引擎

- **异步增量索引**，基于LTS提供可扩展的同步通道，数据同步可视化
- **实时同步索引**，引擎间数据强一致



- 开启云数据库Lindorm压缩优化特性，通过压缩优化数据占用空间，降低存储成本。
- 通过云数据库Lindorm集群内冷热分离功能，0应用改造实现冷热数据分离存储，降低不常用冷数据存储成本，提升常用热数据访问性能。

冷热分离

场景

数据按时间线写入，近线访问为主

价值

降低冷数据的存储成本，提升热数据的访问性能

优势

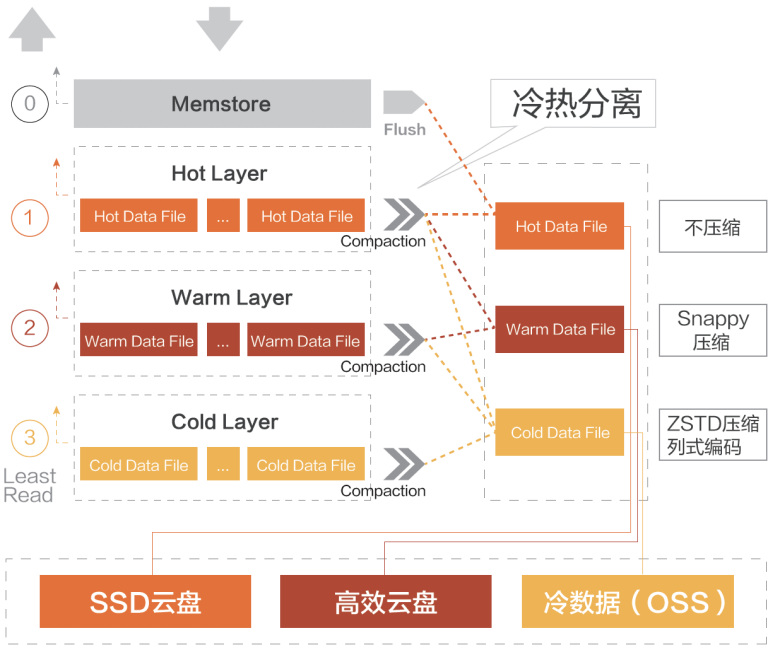
即开即用

• 冷热分离成为一项特性

应用零改造

冷热分隔线，灵活调整

自有设置冷热的存储介质、压缩算法



客户价值

- 云数据库Lindorm搜索引擎方案，使得业务高性能实现高维度&随机组合场景下的订单搜索。
- 云数据库Lindorm宽表引擎压缩优化及集群内冷热分离，使得业务低成本实现数据永久保留。
- 从自建到云数据库Lindorm，提供全托管、免运维及SLA保障，并有专家团队的免费技术支持，使客户能聚焦业务侧发展。

16.5.2. 云数据库Lindorm助力数禾科技业务系统实现真正的流批一体

技术亮点：高性能&低成本解决方案、基于LTS主备集群数据实时同步、Bulkload离线数据加载。

客户感言

从我们看中lindorm容量型存储低廉的价格决定采用阿里云Lindorm产品作为我们event hub流批一体特征平台的底层数据库服务，阿里云Lindorm专家团队和产品团队专程到公司为我们介绍Lindorm产品，特性以及应用场景；在开发过程中当我们碰到困难，阿里云Lindorm专家第一时间给出最佳实践和解决方案。在阿里云Lindorm团队的帮助下，我们相信event hub流批一体特征平台能成为业界领先平台产品。

客户简介

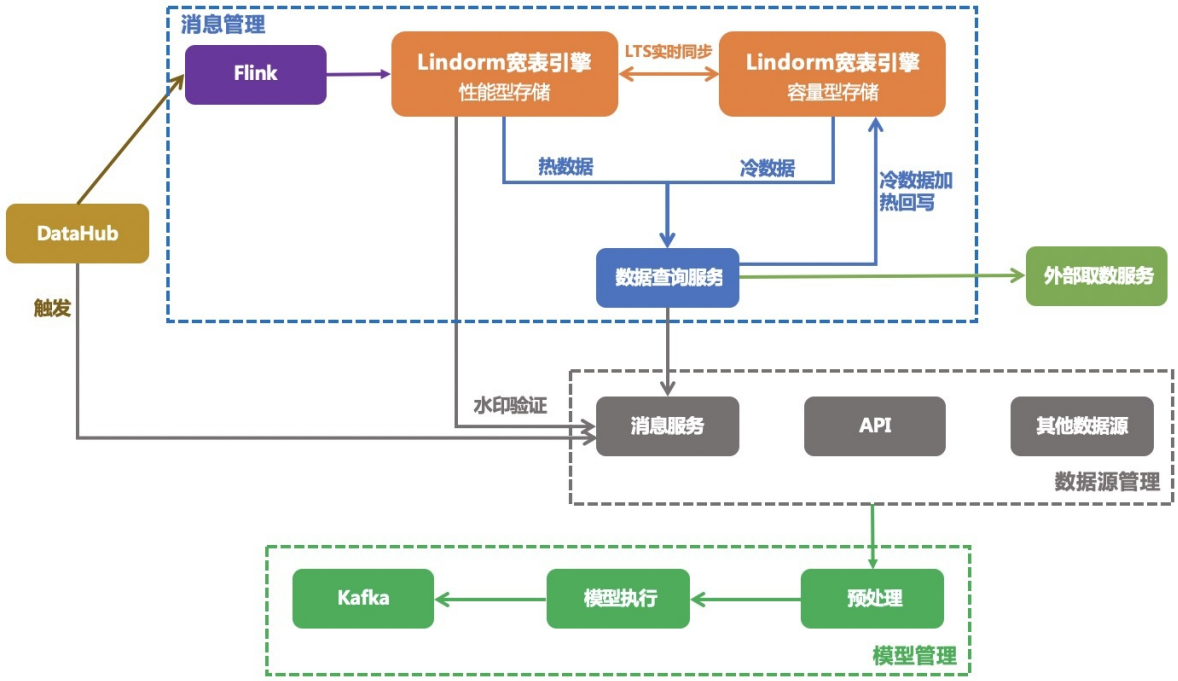
数禾科技以大数据和技术为驱动，为金融机构提供高效的智能零售金融解决方案，服务银行、信托、消费金融公司、保险、小贷公司等持牌金融机构，业务涵盖消费信贷、小微企业信贷、场景分期、财富管理等多个领域，提供营销获客、风险防控、运营管理等服务数禾科技通过自主开发的智能金融产品，连接金融机构与普罗大众，赋能金融机构数字化转型，迎接中国消费升级的大潮。

业务挑战

- 业务数据量大，对可用性高要求，同时成本控制高要求。
- 消息数据来源众多，数据结构变更频繁。
- 特征数据schema日增字段近千个。
- 有大量离线分析结果数据需要导入在线存储，并提供毫秒级、高并发在线访问。
- 自建开源HBase缺少运维平台，手动扩缩容成本高，而且容易导致运维故障。

解决方案

- 阿里云Lindorm历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠，构建起坚实的离在线存储底座，使得客户可以聚焦于业务侧的发展。
- 通过Lindorm LTS实现跨机房主备双集群的实时数据同步，实现内容有：
  - 主备集群存储异构，备集群采用高密度存储+数据保留长周期，从而有效控制成本。
  - 主集群采用高性能存储+数据保留短周期，实现数据高性能读写并且有效控制单个节点数据分片数量，可以有效控制单节点故障恢复时间，提升可用性。



- Lindorm为典型的NoSQL数据库，可以实现列的动态变更，从而很好的满足了数据结构变动频繁的业务。
- Lindorm采用LSM架构，可以通过Bulkload技术实现离线数据到在线数据库的高效加载并且可以将数据加载期间对在线业务请求的影响降到最低。
- 通过和客户沟通了解业务日增千个字段的原因，给出合理的优化建议。

客户价值

- 通过采用阿里云Lindorm，实现集群的运维自动化，大幅度降低了运维成本及由手动操作带来的不可预知的问题。
- 通过主备集群、LTS、异构存储、长短TTL等技术手段，有效的控制了总体成本的同时保证了生产集群高性能、高可用，同时具备了跨机房的数据容灾及数据保留长周期。
- 通过表结构的优化有效控制了字段数量的不断增长，一方面降低了存储开销，另一方面也提升了数据的访问效率。
- 通过Bulkload进行离线分析结果数据加载，消除了加载期间的写入请求尖刺，从而消除了对在线请求的影响，使得业务运行更加稳定。

16.5.3. 某互联网保险公司保险订单迁移MySQL到Lindorm

方案亮点：从MySQL上千张物理分库分表到Lindorm单表；Lindorm完全线性扩展能力从容应对业务量&数据量的快速增长；Lindorm原生存储计算分离架构从容应对大促极致弹性扩缩容需求

业务挑战

- 保单MySQL库业务增速快，需要不断分库分表，运维压力大。
- 业务数据量大，数十TB级别且持续高速增长，存储成本压力大。
- 数据有实时写入和批量导入，特别是批量导入对实时读写rt影响大。

解决方案

- Lindorm历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠
- Lindorm云原生分布式架构可以按需扩容存储和计算节点，扩容&缩容操作对应用透明，业务无需任何代码改动，数据和业务请求自动均衡，零运维
- 开启云原生多模数据库Lindorm压缩优化特性，显著减少存储空间开销
- 通过LTS（Lindorm Tunnel Service）实现离线数据的高效导入，并且对在线读、写业务的影响降到最低
- 借助云原生多模数据库Lindorm集群冷热分离功能，0应用改造实现单表内数据冷热分离，降低存储成本

企业级功能

冷热分离

■ 场景

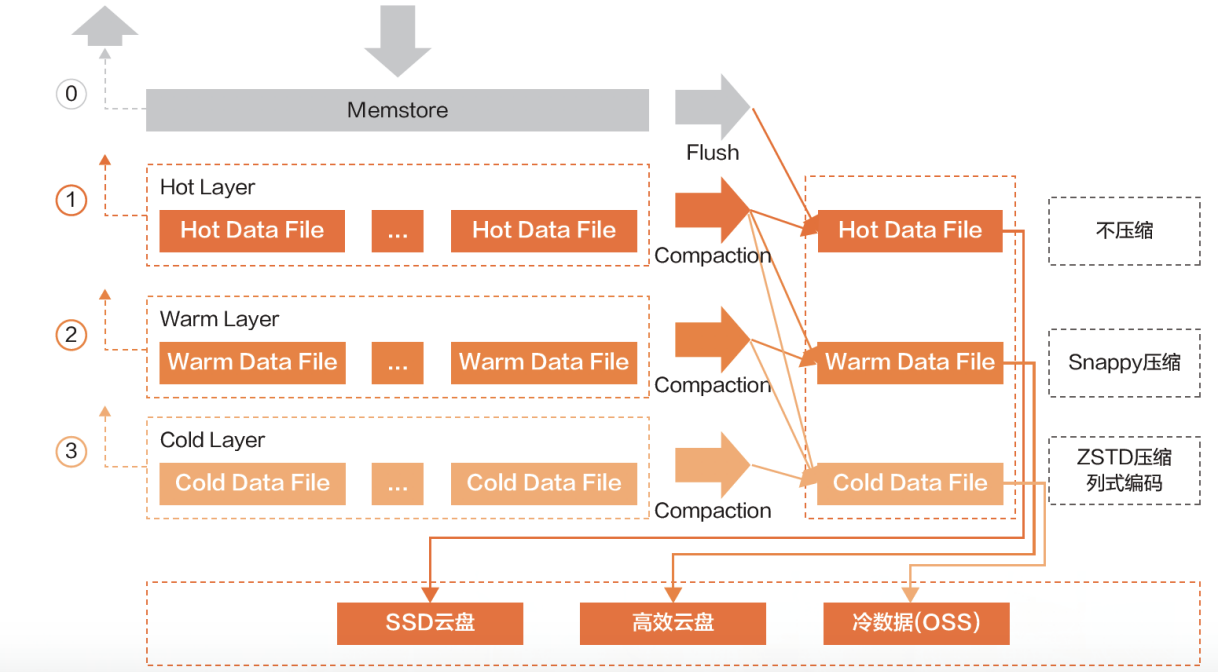
- 数据按时间线写入，近线访问为主

■ 价值

- 降低冷数据的存储成本，提升热数据的访问性能

■ 优势

- 即开即用：冷热分离成为一项特性
- 冷热分割线，灵活调整
- 应用零改造
- 自由设置冷热的存储介质，压缩算法



客户价值

- 从MySQL上千张物理分库分表到Lindorm单表，整个业务从1万+的物理分库分表减少到10+物理表，大幅度降低了运维成本
- Lindorm分布式架构具备完全线性扩展及存储计算分离的能力，从容应对业务请求峰值和数据量的快速增长
- Lindorm压缩存储、专业压缩算法优化、集群内单表冷热分离能力使得业务存储成本大幅度下降
- 利用Lindorm数据通道LTS，实现从MaxCompute高效导入数据，同时保证在线查询的稳定

16.5.4. 云数据库Lindorm助力蚂蚁链IoT数据降本

客户感言

综合对比了多种存储方案，Lindorm是最适合的选择。 Lindorm的自动冷热分离、数据压缩功能完美契合了我们需求，同时，工具链比较成熟，能够平滑完成数据迁移。在系统上线的过程中，Lindorm团队深厚的技术支持能力给了我们很深的印象。

客户简介

蚂蚁链是蚂蚁集团代表性的科技品牌，致力于打造数字经济时代的信任新基建。蚂蚁链坚持核心技术突破，融合包括区块链、AIoT、智能风控等技术，通过链接各个产业网络，扎实解决行业实际问题，推动区块链技术平民化。从2016到2020年，蚂蚁链连续四年区块链专利申请数和授权数全球第一，技术上已经能够支持10亿账户规模，同时能够支持每日10亿交易量，实现每秒10万笔跨链信息处理能力。蚂蚁链坚持开放生态，与合作伙伴共建共享区块链产业带来的价值互联红利。在实际应用上，蚂蚁链已携手生态合作伙伴，解决了50余个场景的信任难题，其中的蚂蚁链IoT可信平台致力于解决实物资产可信上链的问题，接入的设备数量和业务规模快速增长。

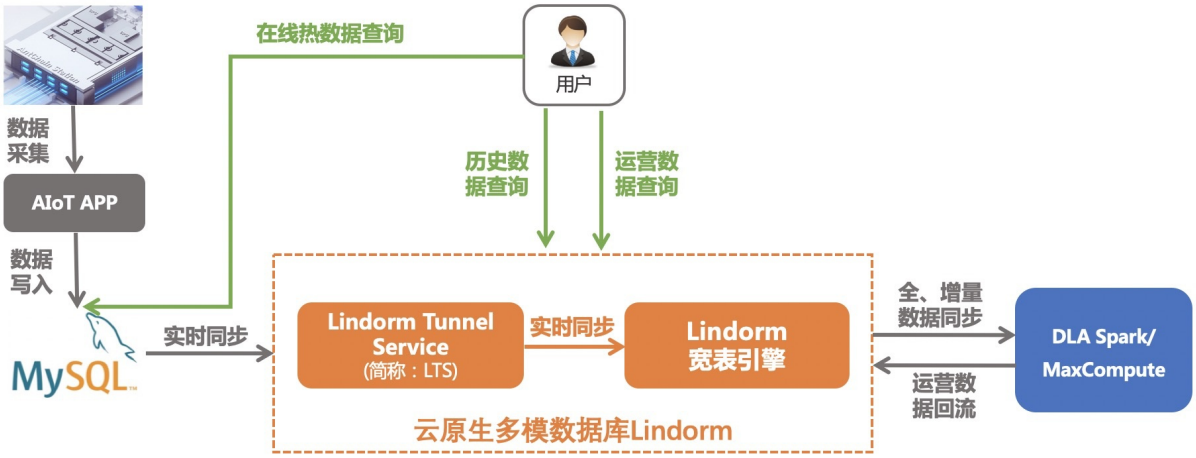
业务挑战

- 业务增速快，数据总量不断增大，MySQL出现性能瓶颈
- 日增数据量大，一年数据增量超6TB，存储成本压力大
- 数据冷热区分明显，和数据生成时间强相关

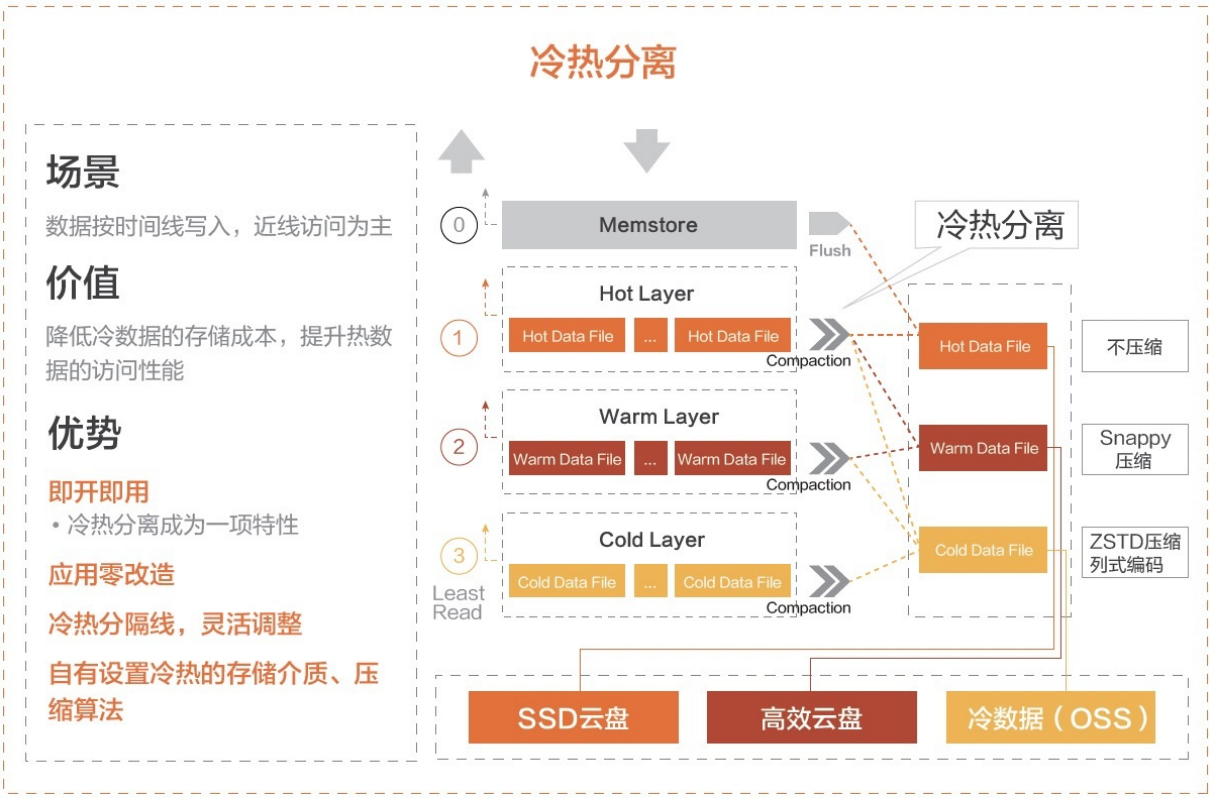
- 数据需要实时归档到离线进行分析

解决方案

- 阿里云Lindorm历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠，构建起坚实的离在线存储底座，使得客户可以聚焦于业务侧的发展。
- 在现有单独采用MySQL作为在线存储的架构下，引入Lindorm作为历史存储，热数据保留在MySQL内，并通过DTS+LTS将数据实时归档到Lindorm中压缩存储&永久保留
- 写入Lindorm的数据，可通过LTS实时将数据归档入湖至DLA Spark或MaxCompute，归档后在离线系统进行分析，分析产出的结果数据亦可通过LTS将数据Bulkload导入到Lindorm提供在线查询



- 借助Lindorm集群内单表冷热分层能力，0应用改造实现单表内数据冷热分离，可实现存储成本的进一步降低



客户价值

- 通过引入Lindorm作为历史数据存储后，MySQL内仅保留较短时间周期内的热数据，数据量大幅度下降，使得性能瓶颈问题得到缓解。

- 通过引入Lindorm作为历史数据存储，为未来做了技术上的储备。未来可以使用Lindorm作为在线存储，利用其良好的扩展能力来应对业务的不断增长。
- Lindorm提供了低成本永久保存数据的能力，并具备0代码改造实现集群内冷热分离，可进一步降低存储成本。
- 集群内单表冷热分离的能力为未来全面使用Lindorm替换MySQL奠定基础。全面使用Lindorm后将消除目前MySQL需要批量删除历史数据的需求，从而消除由此带来的请求尖刺，降低对业务的影响。
- 通过LTS将数据实时归档至MaxCompute，可消除凌晨批量扫描数据的需求，降低对在线业务的影响。LTS的引入也为未来分析结果导入在线存储提供查询服务做了技术上的准备。

16.6. 游戏行业

16.6.1. 某历史养成类游戏开发公司实时计算和数据仓库方案

方案亮点：实时计算峰值百万级TPS、10倍性能提升

业务挑战

某些恶意用户利用信用卡或者别的漏洞刷单（代充），系统需要实时查询多种日志（比如登入日志，充值日志等）从而识别并拦截游戏代充订单，才能避免损失代充订单的钱。

随着业务快速发展，用户行为日志快速增长，需要从海量的点击流日志和激活日志中挖掘数据的价值，比如广告转化率、激活率，每安装用户成本等等。

原来使用Greenplum做实时计算和统计分析。但是Greenplum存在以下缺陷，难以应对业务的快速发展：

1）Greenplum架构难以应对日益复杂的计算任务；

2）Greenplum的单表分区数目有限制，同时多级分区支持不够友好，不适用单表数据量比较大且需要永久保存的日志，如果单个分区表数据量比较大的时候查询性能无法满足业务性能需求；

3）Greenplum扩容时由于数据要重分布会比较慢；

4）Greenplum不适合处理非结构化的数据。

解决方案

客户选择云数据库Lindorm SQL(Phoenix)服务+ 数据湖Spark服务构建实时计算和数据仓库解决方案。

- Lindorm SQL提供在线查询能力、Spark提供流式处理、复杂分析等能力来满足业务需求；
- 梳理业务数据，对数据进行分层存储在Lindorm SQL+Spark中：

1）操作数据层：手游客户端、用户中心、广告监测、游戏服务器等产生的原始日志；
2）数据明细层：操作数据层使用Spark Streaming等进行数据去噪、去重、字段规范后写入Lindorm SQL服务，也可以直接使用JDBC写入数据；
3）数据汇总层：数据明细层的数据可以定期（按天或者按小时）ETL（比如关联维表、过滤、聚合）后写入到数据汇总层。数据汇总层的数据供后续的复杂分析；
4）应用数据层：直接对外提供数据查询服务（基于Lindorm SQL服务）支撑上层大数据风控、广告推荐、海量数据精细化运营；数据汇总层的数据经过处理后会把结果数据写入到Lindorm SQL中对外提供查询服务。



客户价值

- **毫秒级识别拦截代充订单：** Lindorm SQL高并发实时读写，TPS可达百万量级，并发十万量级，毫秒级简单查询；
- **Spark优秀的计算能力：** 同样作业Spark基于列式存储Parquet的分析在数据量大的情况下比Greenplum集群有10倍的性能提升。支撑了广告转化率、激活率，安装用户成本，数据驱动精细化运营等业务计算需求；
- **一站式解决方案：** Spark服务原生支持通过SQL读取Lindorm SQL数据能力，具备列裁剪、谓词下推、分区裁剪等优化；高效地把Lindorm SQL在线库的数据ETL后归档到Spark数据仓库；
- **聚焦业务：** 全托管的Spark服务保证了作业运行的稳定性，释放运维人力，同时数据工作台降低了Spark作业管理成本。

16.6.2. 某手游开发公司在离线一体分析方案

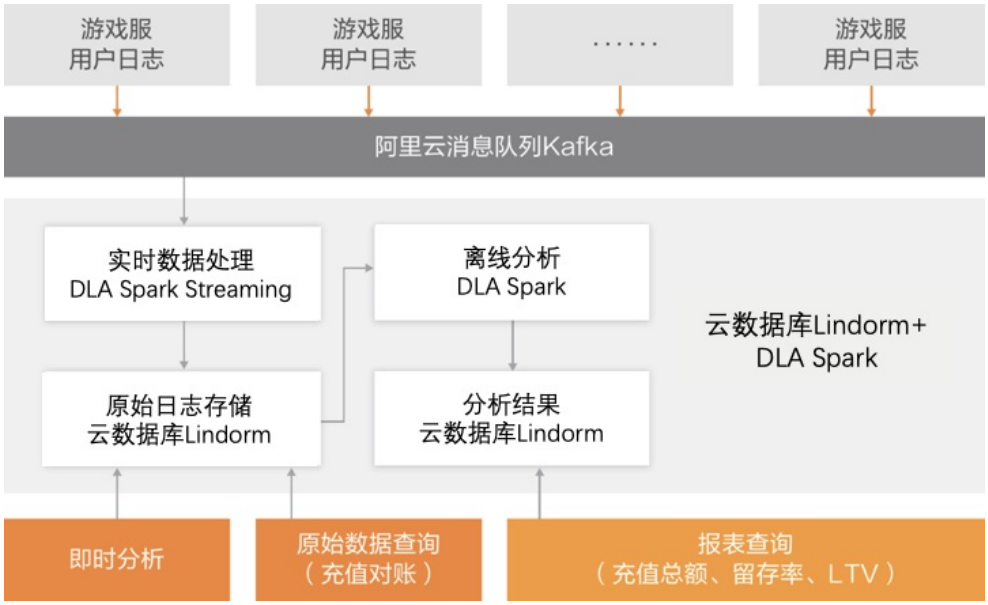
方案亮点：PB级数据量、同时支持在线和离线数据分析

业务挑战

- 通过对游戏服的用户行为日志进行收集、存储和分析，计算玩家留存率、LTV、ARPU、充值总金额等。
- 游戏运营变化快，需要灵活的Schema。
- 在线用户基数大，日志数据规模大，需要高吞吐低成本处理平台。
- GM需要对原始数据进行核对，另外有一些即时数据分析需求。需要同时支持在线型和离线型业务场景。

解决方案

- 云数据库Lindorm能适应业务的快速变化，支持灵活的Schema，动态列等。
- 云数据库Lindorm能支持海量数据实时写入，百万/千万并发，PB级存储，同时HBase支持冷热数据分离，通过直接读写OSS降低日志存储成本。
- DLA Spark和云数据库Lindorm无缝衔接，支持数据工作台、作业调度、SQL，支持构建离线数据仓库等；通过DLA Spark Streaming进行实时ETL，并将结果写入云数据库HBase版；通过DLA Spark进行离线分析解决客户的报表等需求。



客户价值

- 迁移到云数据库Lindorm后，不仅能够高效应对业务快速增长，而且满足了高并发写的需求，为辅助运营决策提供了支持。
- 通过和数据湖DLA Spark的组合，完美解决了同时支持实时ETL以及离线分析等需求，将各种需求形成闭环。

16.6.3. 某中生代游戏公司广告业务HBase搬站

方案亮点：迁移后性能提升至少30倍、平均响应时间从几百毫秒下降到20ms

客户感言

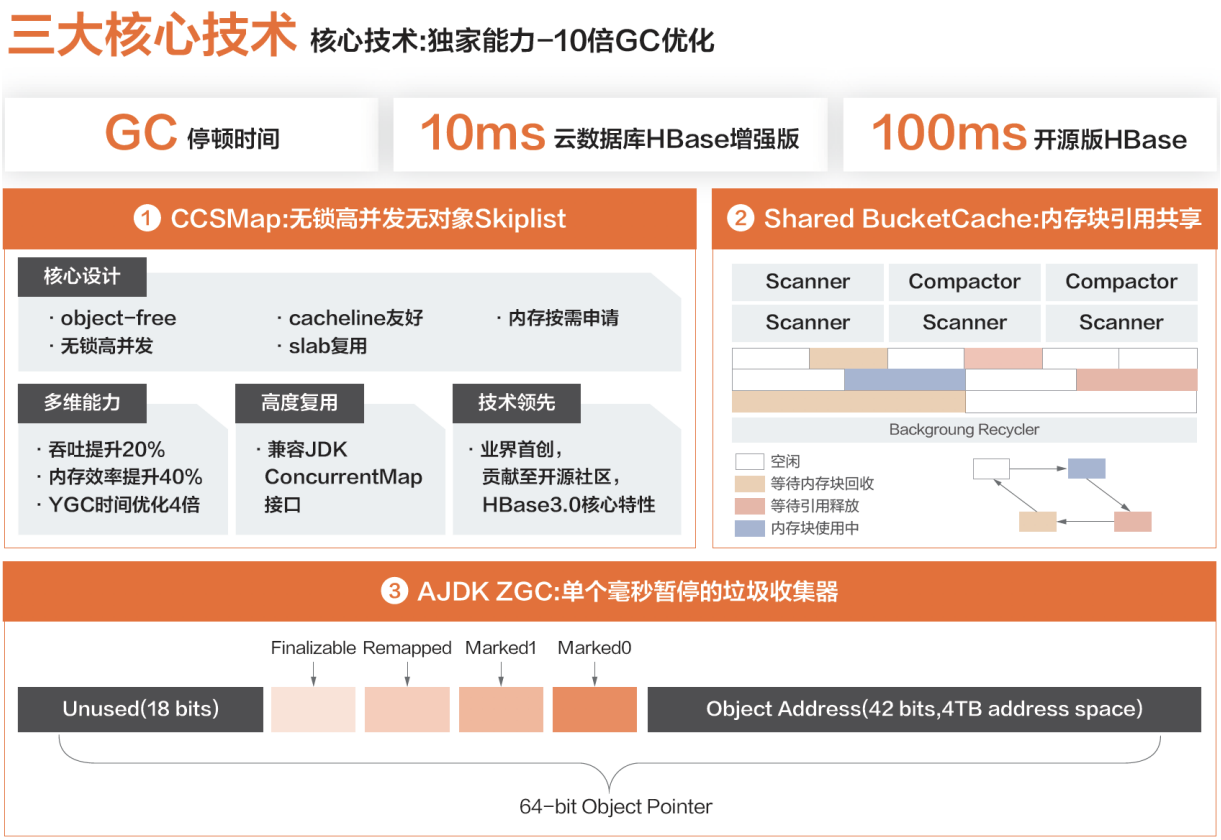
之前自建集群表现很不稳定，也很难取到一个固定表现值进行量化，自从切换到云数据库Lindorm后稳定性大幅提高，性能提升至少30倍，而且仍有提升空间。

业务挑战

- 自建集群为多个大数据产品混部，不同产品间相互影响，导致请求稳定性差，请求延迟高。
- 集群成本高昂，使用30余台高配32C128G ECS。
- 迁移前数据量已达到24TB，自建集群存储空间开销不断攀升。

解决方案

- 从自建混部集群迁移云数据库Lindorm独立集群，平均请求RT从500～800ms下降到20ms，大幅度提升广告采集系统的吞吐量。
- 通过和客户沟通给出业务侧优化建议，同时启用云数据库Lindorm压缩优化特性，最高可达13倍数据压缩比，最终实现存储空间开销下降20倍以上。
- 云数据库Lindorm GC优化，LLC三副本协议，实现P999毛刺数量级下降。
- 云数据库Lindorm高可用方案：同城主备架构，使业务具备故障情况下的快速切换能力。



客户价值

- 迁移到云数据库Lindorm后，性能显著提升，存储开销下降，从而使得业务的吞吐量大幅提高。
- 迁移后具备了同城主备能力，使得业务具备了跨可用区容灾的能力，极端情况下仍能保持业务的连续性。
- 从自建混部集群到云数据库Lindorm，阿里云提供全托管、免运维及SLA保障，并有专家团队的免费技术支持，使客户能聚焦业务侧发展。

16.6.4. 云数据库Lindorm助力趣加互娱低成本上云

技术亮点：集群内单表冷热数据分离、相比自建HBase成本下降60%、更高的可用性稳定性，连续两年稳定运行

客户感言

首先阿里云Lindorm很适合我们的业务应用场景，在接触阿里云Lindorm之前，一直使用云主机自建HBase，稳定性较差，在使用与维护过程中缺少专业的技术支持与积累，每次遇到问题总会影响业务。偶然机会与阿里云技术专家交流，发现有产品级服务可用。通过与阿里云专家们沟通，进行简单测试后迁移上线，上线与使用过程中一直有专家24*7支持。使用云数据库Lindorm集群两年来，一直平稳运行，也免去了运维消耗，真正做到了把精力放在业务上。不仅如此，通过采用阿里云Lindorm团队自研的数据冷热分离存储方案，实现成本比自建HBase集群节省近60%，两年内已经成功将4个项目运行在阿里云Lindorm集群上。

客户简介

FunPlus于2010年在硅谷创立，以“用最好的产品为全球玩家带来极致的娱乐享受”为使命，致力于用游戏及娱乐产品连接全球用户、连接合作伙伴、连接多元文化。十年来，FunPlus已成长为全球最顶级的移动游戏公司之一，创造出《阿瓦隆之王》、《火器文明》、《State of Survival》等风靡全球的移动游戏，连接了全球数亿用户。未来，FunPlus将继续追寻“成为全球互动娱乐前三极”的愿景，不断挑战和突破自我，创造充满爱和乐趣的梦想王国。2021年，趣加互娱正式成立。作为FunPlus为聚焦中国市场业务而进行的重要战略部署，趣加互娱将以“好奇心、冠军精神、追求极致”为企业文化，在游戏、电子竞技等领域不断探索和创新，为中国市场带来更多具有广泛影响力的头部产品，为用户带来极致的互动娱乐体验。2019年FunPlus趣加超越腾讯网易摘得“年度中国厂商出海收入第一”桂冠！

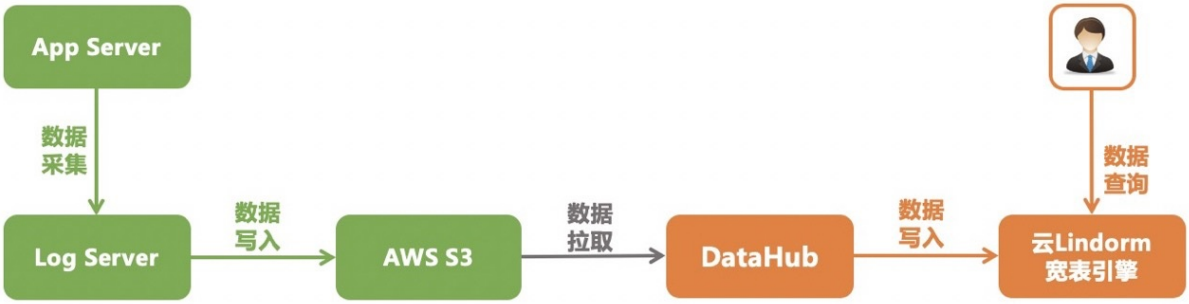
业务挑战

- 为满足问题排查和分析需求，需要将日增30TB的海量数据写入数据存储，提供低延迟在线查询能力
- 自建HBase运维成本高，稳定性&可用性问题突出，无法满足业务需求
- 业务写多读少，随时间冷热区分明显

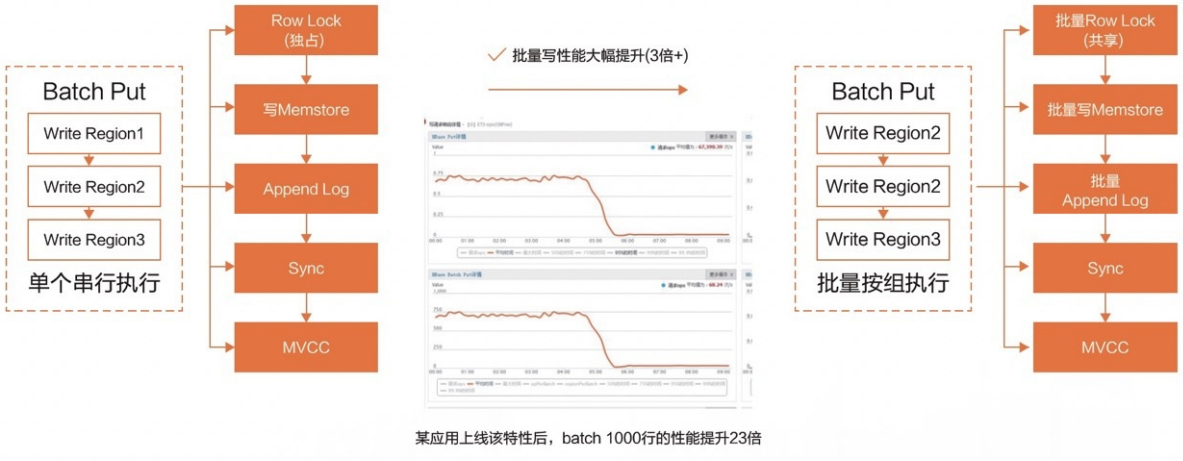
- 结合业务需求和成本诉求，数据需要保留至少60天，而且有进一步延长的可能，存储成本压力巨大

解决方案

- 数据写入、读取链路

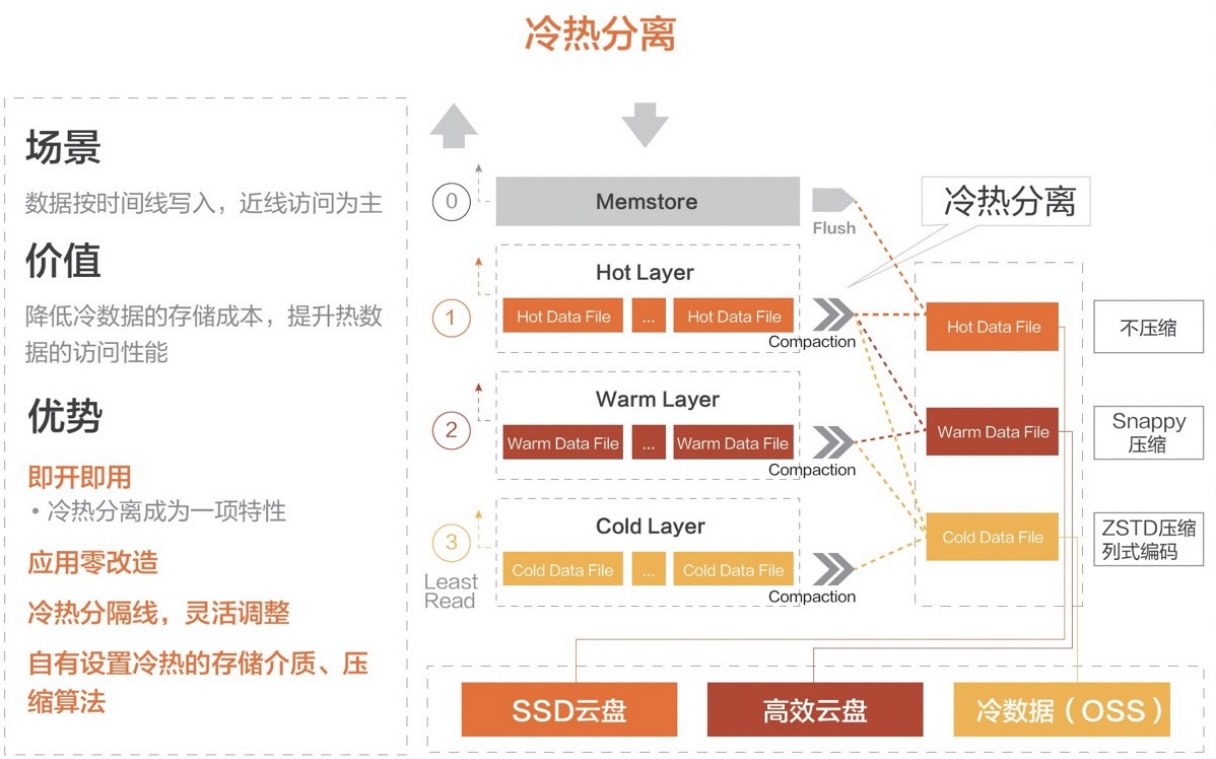


- Lindorm历经阿里众多核心服务的大规模验证，拥有国内相关技术领域的顶尖技术团队，保障了使用过程的持续稳定、可靠，构建起坚实的离在线存储底座，使得客户可以聚焦于业务侧的发展
- Lindorm宽表引擎批量组提交优化，大幅度提升集群吞吐量、降低请求响应时间，性能提升3倍以上



某应用上线该特性后，batch 1000行的性能提升23倍

- Lindorm宽表引擎支持ZSTD&字典压缩，提供超10倍数据压缩比
- Lindorm具备集群内单表冷热分层能力，0应用改造实现单表内数据冷热分离，可实现存储成本的进一步降低



客户价值

- 相比用户通过购买ECS自建开源HBase，云数据库Lindorm为全托管产品，使得用户能更加关注于业务侧的发展。
- 云数据库Lindorm历经十年以上双十一大促验证，相比自建开源HBase集群在稳定性、可用性方面更好的满足了业务要求。
- Lindorm宽表引擎批量写入优化可有效提升集群的吞吐能力，从而可以减少计算节点数量，达到降低成本的诉求。
- Lindorm高效压缩特性可提供超10倍的数据压缩比，结合Lindorm实现的集群内单表冷热分离功能，采用Lindorm后可大幅度降低业务的存储成本

16.7. 教育行业

16.7.1. 某在线教育领导品牌推荐业务迁移至阿里云Lindorm

方案亮点：吞吐能力3倍于自建、写入延迟降低到自建1/10、高度弹性轻松应对大促扩缩容

业务挑战

- 自建开源HBase集群性能不足，难以满足每秒数十万的事件实时写入和计算
- 自建开源HBase集群GC问题严重，导致请求稳定性差，故障频发，可用性问题突出
- 随着时间的积累数据量越来越大，存储成本问题突出
- 自建开源HBase缺少运维平台，手动扩缩容成本高，而且容易导致运维故障

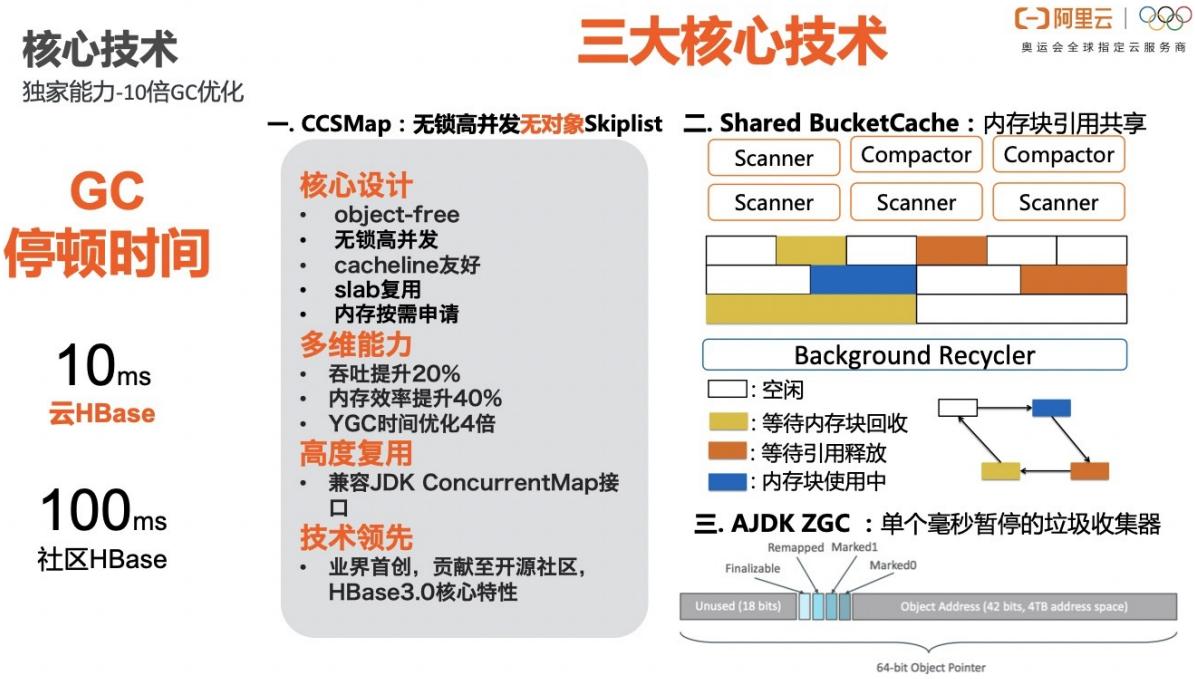
解决方案

- 高并发、高吞吐能力：1）Lindorm Group Commit写入优化，批量写性能优化3倍；2）Lindorm LLC日志三副本能力，通过Quorum机制写入延迟减少50%； 3）Lindorm具备完全线性扩展能力，天然支持单表千万级读写，无需分库分表

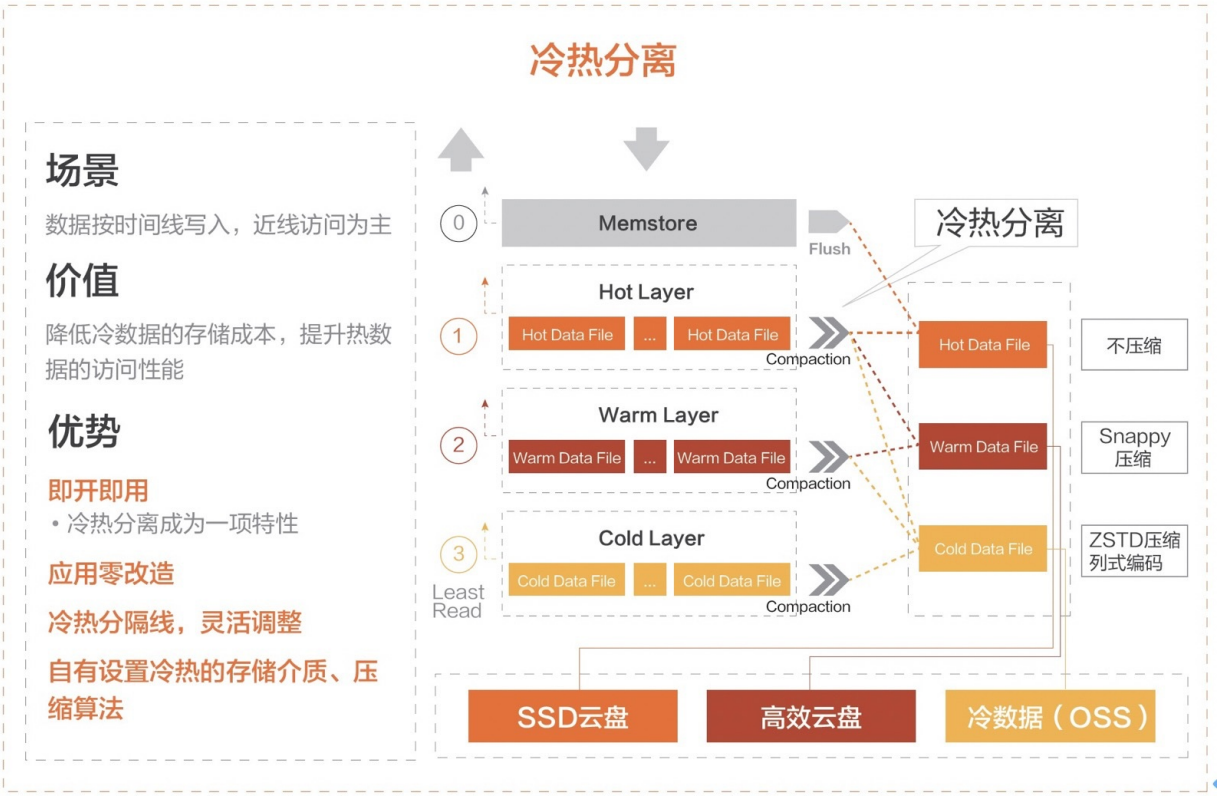


某应用上线该特性后，batch 1000行的性能提升23倍

- 通过GC优化，云Lindorm实现P999毛刺数量级下降，使得业务请求更加稳定



- 通过支持更优的压缩算法，可以实现存储开销减半，降低存储成本
- 借助云原生多模数据库Lindorm集群冷热分离功能，0应用改造实现单表内数据冷热分离，降低存储成本



- 云Lindorm采用存储计算分离架构，可以按需扩容存储或计算节点，扩容&缩容操作对应用透明，业务无需任何代码改动，数据和业务请求自动均衡，零运维

客户价值

- 迁移到云Lindorm后实现3倍于自建的吞吐能力，轻松支撑20W/秒的写入，写入延迟降低到1/10
- 迁移到云Lindorm后压缩能力两倍于自建开源HBase，存储成本降低50%以上，使用集群内单表冷热分层存储可进一步降低存储成本
- 云Lindorm基于AJDK ZGC（阿里JDK）实现GC优化，内核层面优化消除GC带来的请求不稳定问题，迁移后至今稳定未出现故障
- 通过云Lindorm管控实现一键扩缩容，轻松应对春晚大促。同时大幅度降低了运维成本及由手动操作带来的不可预知的问题。

16.8. 通用行业

16.8.1. Napatech案例

云原生多模数据库Lindorm助力Napatech大幅提升存储、索引性能。

生态合作伙伴介绍

Napatech公司（OSE：NAPA.OL）是一家提供FPGA解决方案的丹麦公司，成立于2003年。公司总部位于丹麦，销售和支持的区域包括美国，亚太地区，欧洲，中东和非洲。在网络和安全性应用领域率先使用基于FPGA的技术，通过FPGA可重构计算平台，向电信、金融行业客户及互联网公司提供网络管理和安全应用数据交付解决方案。

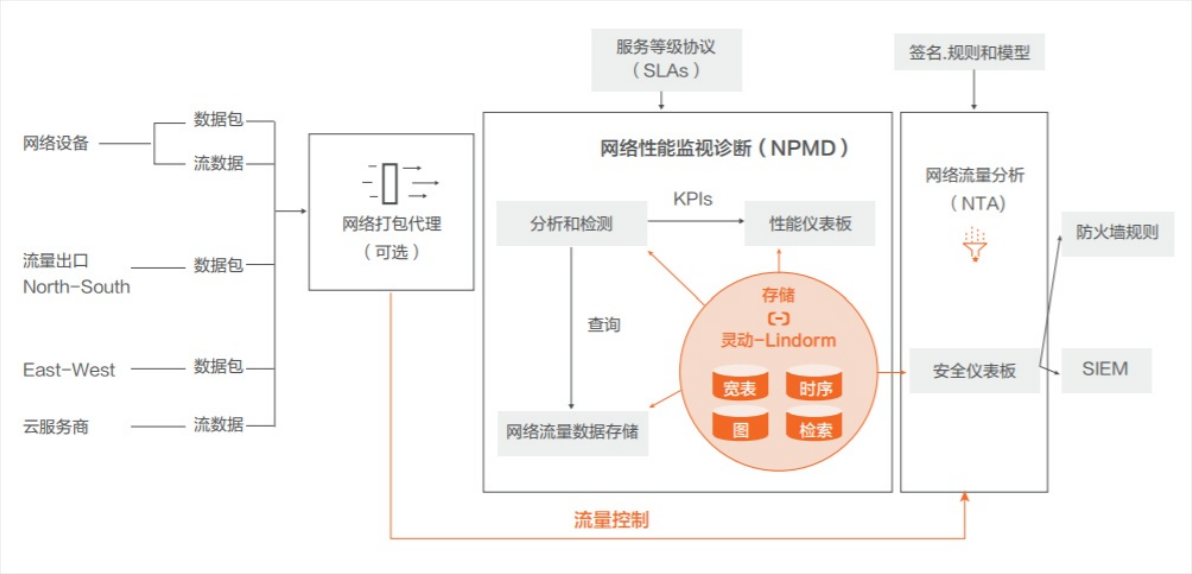
业务挑战

- 随着网络数据流量的爆增以及业务可用性要求的不断提升，Napatech公司的用户可以将数据包采集和分析的能力提升至200G，然而上层软件产生的流量数据包元数据也爆增，给后台数据库（比如开源的HBase/ElasticSearch）的存储和索引能力带来极大的挑战。
- 增加过多的节点会带来极大的成本开销及管理复杂问题。
- 流量数据包元数据的存储和索引技术的性能提升需求越来越迫切。

解决方案

- 阿里云原生多模数据库Lindorm是一款适用于任何规模、多种模型的云原生数据库服务，其基于存储计算分离、多模共享融合的云原生架构设计，具备弹性、低成本、稳定可靠、简单易用、开放、生态友好等优势。

- Lindorm支持海量数据的低成本存储、快速批量导入和实时访问，具备高效的增量及全量数据通道，可轻松与各类大数据平台集成，完成数据的大规模离线分析。
- 基于Lindorm的解决方案，用户在采用到100Gbps+流量产生的大量数据包元数据直接通过Lindorm存储，在数据的存储和索引性能方面得到极大的提升，同时也极大降低整个系统的运维成本。



客户价值

- 支持多个100G数据流量采集点的数据包元数据存储及快速索引。
- 帮助用户实现全量数据的回溯分析。
- 数据库架构简化、支持弹性扩容，运维难度及成本大幅减低。
- 集成第三方大数据系统，助力业务分析。

16.8.2. 图扑案例

厦门图扑软件联手阿里云Lindorm（灵动）数据库开启工业物联超融合存储模式，应用于工业、制造、楼宇建筑、航空与航天行业。

业务/技术亮点

- 百PB级海量多源异构监控数据一站存储。
- 每秒千万级吞吐量轻松应对监控指标高并发写入。
- 多模数据检索引擎协同工作，解锁更强悍数据可视化场景。

客户感言

阿里云原生多模数据库Lindorm针对工业物联网数据高并发写入，实时存取等特点，创新性地融合时序、索引、宽表等多模引擎能力，为存储、分析低价值密度、高通量、高实时性的工业物联网监控数据提供了高性价比的最优解决方案，大幅度降低了数据存储和存储系统运维成本。

阿里云Lindorm数据库驱动的IT运维监控系统



客户简介

图扑软件成立于 2013 年，总部位于厦门，在北京、上海、天津、大连、青岛等地设有分支机构。公司聚焦工业物联网监控运维可视化应用领域，为客户提供从咨询、设计、实施到售后的全方位可视化管理支持服务。图扑软件专注基于Web 的 2D 和 3D 图形界面组件技术，致力于让完全独立自主知识产权的 HT for Web 软件走向世界之巅。公司系列产品 HT for Web 已应用于电信、电力、交通、水利、石化、制造、医疗、工控等行业场景。

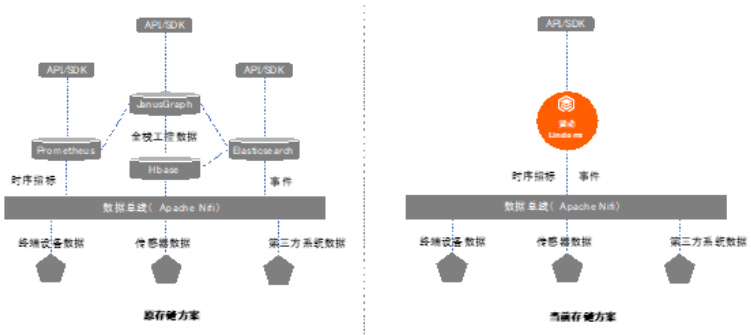
业务要求与挑战

5G、云计算、边缘计算等智能、互联技术的快速发展，推动了工业物联网IIoT场景下各种软件、硬件传感器数量激增，与之对映的采集数据量和数据类型快速增长，导致数据存储、检索难度更大。现有解决方案中，通常依赖自建单ElasticSearch检索引擎或OpenTSDB、Prometheus等时序引擎搭建数据存储，而采集数据类型多样化使得单模引擎存储方案技术复杂且运维成本高，市场需要新一代云原生且具备多模检索能力的存储系统。某著名IT咨询公司预测，工业物联网市场规模在2025年可能达到3.7万亿美元，但统计数据显示，现在只有不到30%的供应商从中盈利，大多数企业都陷入了技术陷阱之中。新技术在创造新机遇带动产业升级的同时也带来了新的技术挑战，更加复杂的系统架构和更高的性能、稳定性要求制约了工业物联网系统实施落地，企业需要专业技术公司来帮助解决数据采集、传输、存储、分析及可视化全链数据处理系统建设难点。

解决方案

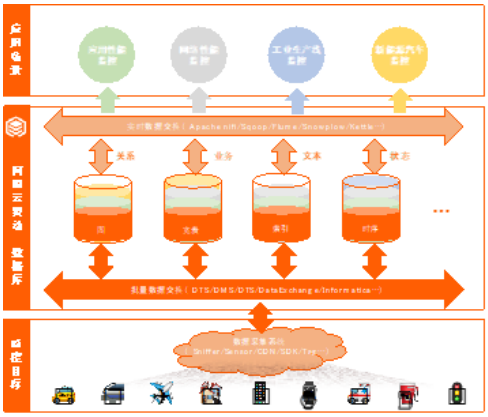
厦门图扑软件致力于解决工业物联网全链路数据处理最后一公里的数据可视化环节技术难点。面向工业物联网IIoT场景，图扑软件提供监控系统监控可视化解决方案，其产品可用于快速创建和部署，高度可定制化，并具有强大交互功能的拓扑图形及表盘图表等应用，非常适用于实时监控系统的界面呈现，广泛应用于电信网络拓扑和设备管理，以及电力、燃气等工业自动化(HMI/SCADA)领域。对于需要实时采集海量数据的生产线、风电厂和智能交通态势感知数据的场景，为了实现实时数据采集、存储、索引和聚合，原存储方案中采用了ElasticSearch、Prometheus、Hbase分别存储从现场传感器、第三方系统和用户终端设备采集的时序指标、日志、用户体验、网络流量等数据，随着数据量增加，可视化展示界面场景复杂化，数据存储和运维成本激增，检索难度快速上升，严重制约了交付效果。

数据存储系统改造前后方案对比



针对存储层面临的问题，厦门图扑软件基于阿里云Lindorm灵动云原生多模数据库改造存储层架构，以单库多模超融合模式存储全量采集的监控数据（技术方案对比如图2所示），极大地简化了存储层架构，进而降低了运维成本。利用Lindorm自研的数据压缩存储和存储优化能力，海量低价值密度的监控数据存储成本也有大幅度降低。为适应日趋复杂化多样化的终端设备、边缘设备、传感器以及第三方系统数据源采集、上报的异构数据，如图3所示，阿里云Lindorm在云端单实例融合了宽表、索引、时序等多种数据引擎能力，通过阿里云DTS/DMS或第三方开源数据交换/ETL软件（如Apache nifi、Sqoop等）打通多引擎数据交互通道，根据应用场景业务来适配数据。面向上层数据可视化、分析系统，Lindorm提供更为便捷的开发期SDK和REST API数据对接方案，同时兼容OpenTSDB、Prometheus、Hbase等原生接口，无缝对接主流生态，进一步简化了图扑软件集成部署成本。

阿里云Lindorm（灵动）多模数据融合存储架构



适用场景

- 实时场景监控大屏展现。
- 态势感知及风险监控。
- 设备监控数据探伤检测。
- 故障数据全量回溯分析。
- AI辅助异常检测等场景。

客户价值

- 超融合存储海量异构数据，大幅降低工业物联网场景数据存储、聚合成本。
- 高性能、高通量监控数据入库，提升可视化大屏数据时效性。
- 提供99.95% 高可靠性，保障业务持续稳定运行。
- 云端接入遍在可达，简化网络配置管理。
- 开箱即用免维护，进一步降低系统维护成本。
- 兼容OpenTSDB、Prometheus接口，无缝对接主流生态。

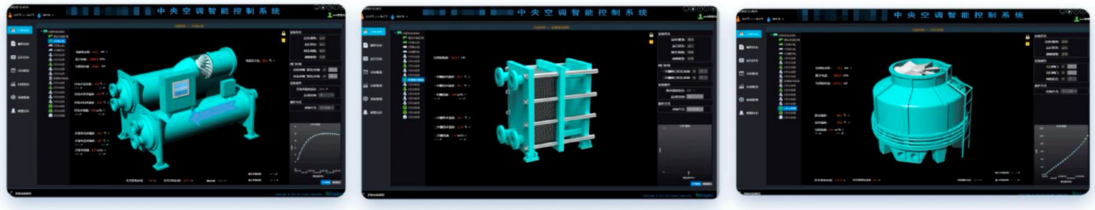
建设效果

目前图扑软件联合阿里云端支撑某行业领先泛在电力物联网及智能楼宇行业客户实现终端设备传感器云端数据采集、存储、检索，实现并发量100万TPS以上的传感器采集数据的并发写入，最高存储时序时间线40万能力，节约数据存储和系统维护成本达6成，实现效果如图4、图5所示。

泛在电力物联网监控系统建设效果



智能楼宇场景监控系统建设效果



16.8.3. 东软案例

阿里云Lindorm数据库让数字时代运维“灵动”起来，本文以东软集团有限公司为例，该集团归属于运营商、政务、汽车、医疗等行业。

业务/技术亮点

- 移动端到云端业务全链路海量运维大数据存储。
- 指标、日志等多源异构监控数据融合分析。
- 99.99%数据可用性的高可靠低成本存储。

客户感言

利用阿里云新一代云原生多模数据库Lindorm“灵动”引擎驱动的政府、企业数字信息系统智能运维解决方案，东软做到了实时、海量、异构监控数据一站式存储，实现指标、日志、代码链路和网络包等异构数据融合分析，高可靠数据保障和遍在可达实时监控数据存储云端服务。如下图所示，阿里Lindorm数据库在赋能政企数字运维团队更强大的运维大数据分析能力的同时，大幅度提升了政府、企业数字系统性能和在线服务的客户数字体验，降低了低价值密度监控数据存储管理成本。

阿里云Lindorm数据库驱动的IT运维监控系统



客户简介

东软创立于1991年，是中国第一家上市的软件公司，一直以来致力于以信息技术的创新，推动社会发展，创造美好生活。东软集团以软件技术为核心，业务领域覆盖智慧城市、医疗健康、智能汽车互联及软件产品与服务。目前，东软在全球拥有近20000名员工，在中国建立了覆盖60多个城市的研发、销售及服务网络，在美国、日本、欧洲等地设有子公司。此外，东软连续四次入选普华永道“全球软件百强企业”，还曾荣获最具全球竞争力中国公司20强、中国50强全球挑战者、亚洲最受赏识的知识型企业、亚太地区最佳雇主等奖项。

业务要求与挑战

线上互联网化的信息服务繁荣发展为政府、企业数字系统运维应用维稳和客户数字体验保障提出了更高要求。在互联网服务场景下，运维监控数据量激增，采集监控的数据类型更加多样（时序指标、日志、代码链路等），现有运维系统采用的单模引擎（如RRD数据库、openTSDB时序数据库、ElasticSearch检索类数据库）应对这些实时、高并发采集，且价值密度较低的监控数据存储和检索场景时，已经显得力不从心。

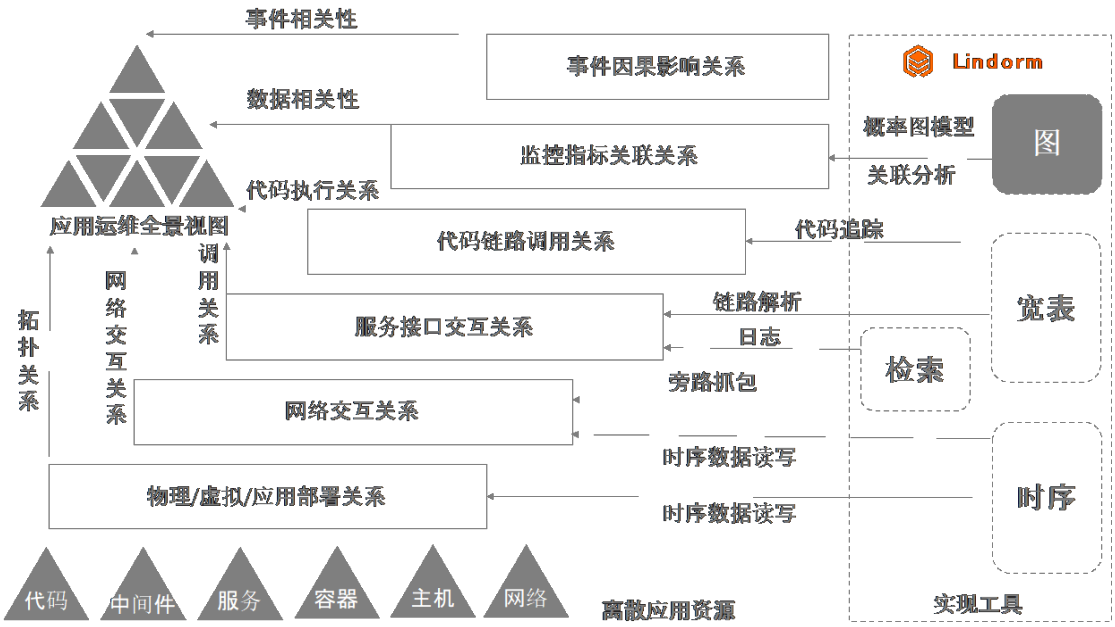
根据Forrest统计数据，目前有57%的企业客户IT运维部反馈至少每周会发生一次影响应用性能和可用性的问题；每天都发生问题的比例占到了28%。对于愈加依赖应用面向客户实现企业价值，提升工作效率的当今企业来说，这种问题越来越无法忍受。统计数据显示有超过一半的企业认为由于应用性能问题直接导致业务用户和IT部门生效率降低和时间浪费；有42%的企业认为应用性能问题直接影响了企业收入。总得来说，造成这种现象的主要问题原因，可总结为以下两个方面：

- 新需求推动IT系统监控数据量激增： 移动智能终端设备的普及使应用逐渐渗入到我们工作生活的方方面面，企业应用数量激增。企业面向客户、合作伙伴和内部员工建设IT系统及应用复杂度和数量会随着产品智能、互联化的深入持续增长，对映可用性保障运维监控采集数据采集种类，以及需要存储的数据量同步激增，现有监控系统基于RRD数据库或关系数据库搭建的单模引擎的存储、检索能力和需求脱节。
- 产品数字化导致应用结构愈加复杂，监控数据类型更加多样：技术方面，诸如混合云、数据分析、物联网、车联网、体域网等新技术的持续演进也使得应用结构愈加复杂，保障应用性能更加困难。据统计，超过一半（52%）的企业IT运维部门在监控管理工具上的投入是被动、针对特定问题且分散的。但由于存储能力有限，单模存储引擎功能单一，数据存储分散、碎片化，难以应对未来以应用为核心的IT监控运维新需求和技术演进。随着时间的推移，现有IT运维数据存储分析问题会恶化。激增的应用复杂度及监控数据采集量和在网运维系统有限的监控运维数据存储检索能力之间的矛盾在加剧。目前，东软在物联网、互联网等新场景下面临的IT系统运维主要问题与挑战有：多模型数据融合分析困难，面向海量数据采集终端同时写入数据的并发能力弱，数据量大且价值密度低导致存储成本高，基于开源软件自建数据存储集群稳定性低运维成本高等问题，东软急需新型运维大数据存储引擎支撑，来对运维系统存储引擎升级改造。

解决方案

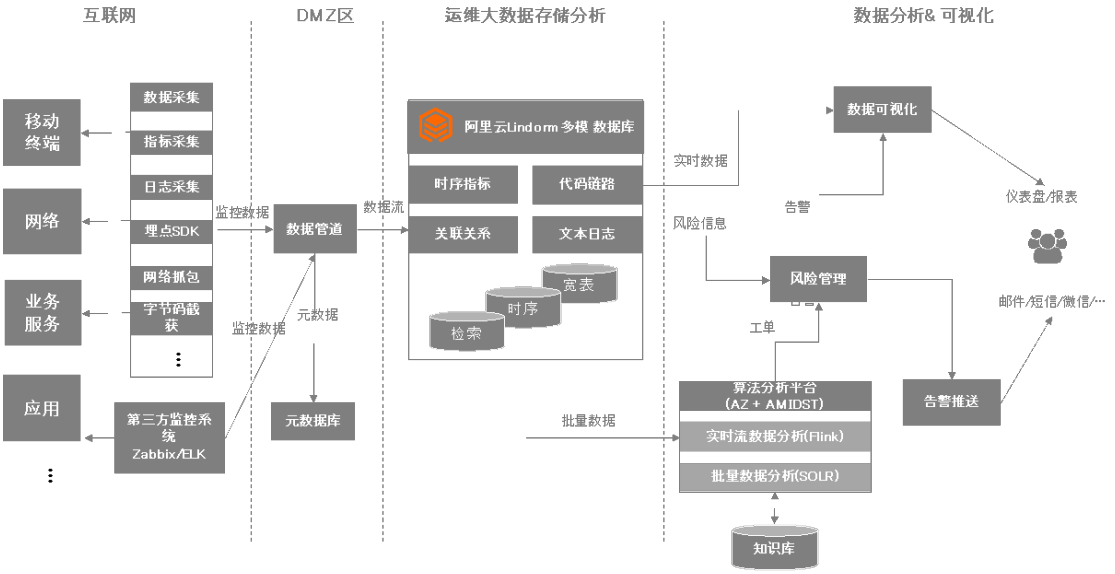
东软集团围绕运营商、政务云和汽车等行业新一代数字信息系统IT运维场景海量监控数据存储分析痛点，将从遍布全国多地域的手机、平板等终端连接应用系统业务服务的数字足迹和客户数字体验数据、云端&数据中心网络抓包分析数据，应用运行代码链路和日志数据全量采集存储于阿里云Lindorm灵动云原生多模数据库，如下图所示。

基于阿里云Lindorm的运维多模数据融合存储分析



以阿里云的云端Lindorm云原生多模数据库为核心，东软围绕运营商、汽车等目标场景监控运维特点，进一步扩展了应用智能运维产品RealSight APM和云管理产品SaCa Aclome能力，实现了高通量、高并发监控数据的实时、低成本存储。利用内置多模数据引擎处理能力，阿里云Lindorm完美适配东软客户IT运维监控场景数据存储分析需求。东软基于Lindorm打造，面向政企互联网化数字信息系统运维监控场景的新一代IT智能运维系统部署架构如下图所示，其中Lindorm作为核心运维数据存储、检索引擎全量接收来自移动终端、网络嗅探器、业务监控探针、日志采集器等多种数据来源实时采集的异构监控数据，以高并发、高通量数据写入方式将数据入库。在运维过程中，用户通过实时监控仪表盘、统计分析报表或风险定位、检测工具对接Lindorm多模引擎海量数据检索、分析能力，来满足不同应用场景的运维大数据信息提取需要。

东软新型IT智能运维系统部署方案



适用场景

- 实时状态大屏展现。
- 应用海量日志检索。
- 定点用户行为追踪。
- 故障数据全量回溯分析。
- AI辅助异常检测等场景。

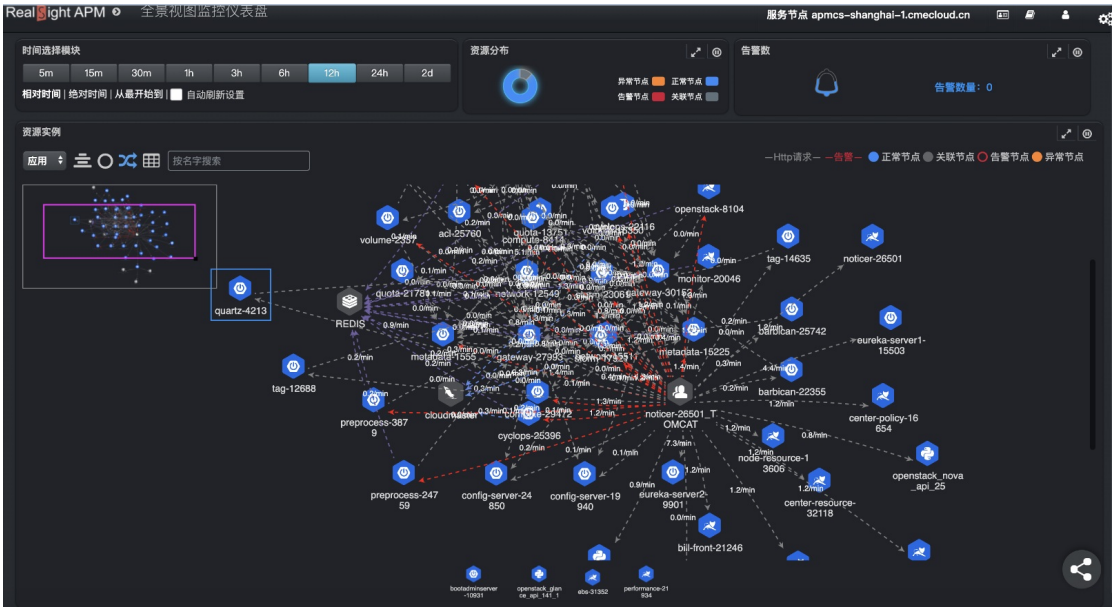
客户价值

- 一体化指标、日志、代码链路、网络包等结构化、半结构化、无结构数据存储、检索、分析能力，降低开发部署复杂度。
- 云端低成本海量运维数据存储，TCO大幅降低。
- 高性能、高通量监控数据入库，轻松搞定大日活量应用系统监控。
- 实时日志、指标等多模异构数据监控和回溯分析，简化低价值密度数据的信息提取。
- 99.99%数据可用性，降低数据丢失风险。
- 云端接入遍在可达，简化网络配置管理。
- 开箱即用免维护，进一步降低系统维护成本。
- 提供实时高精度全量监控数据存储和分析能力，为监控目标系统保驾护航，间接提升客户数字体验。

建设效果

目前系统已经在阿里云端支撑某行业领先车企存储、检索从全球客户终端采集的用户数字足迹和客户体验数据，应对日活量上万客户访问监控数据采集，日均实时监控指标数据采集1.2亿元组，节约数据存储和系统维护成本达4成，建设效果如下图所示。

运维大数据可视化界面效果图1



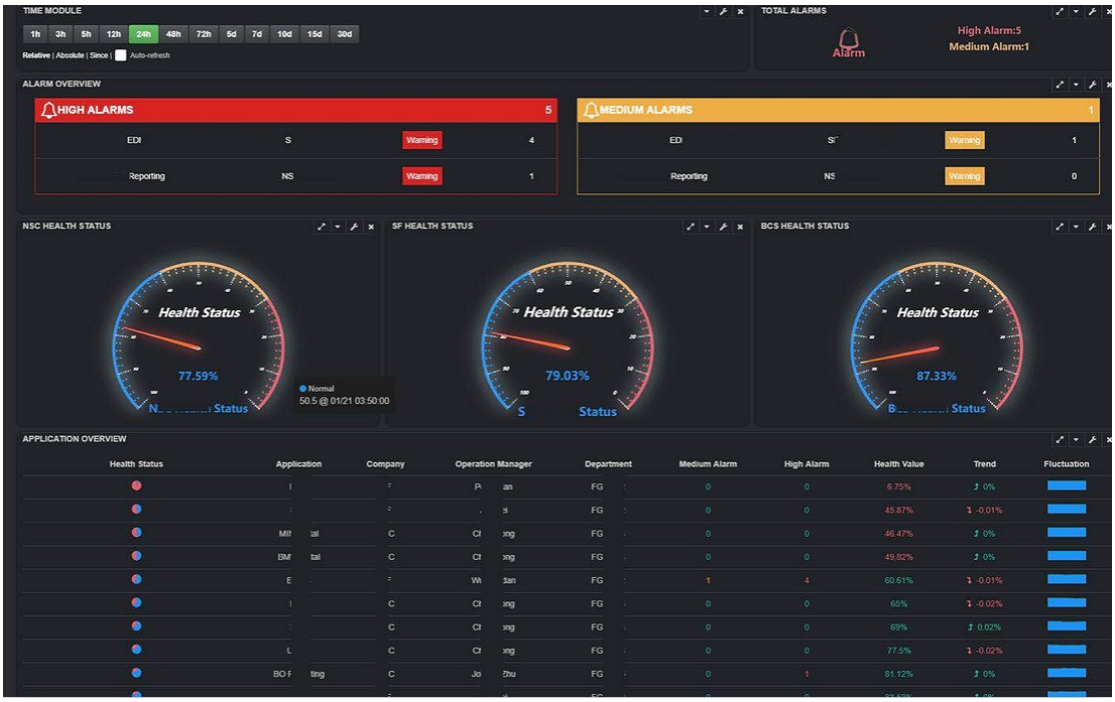
运维大数据可视化界面效果图2



运维大数据可视化界面效果图3



运维大数据可视化界面效果图4

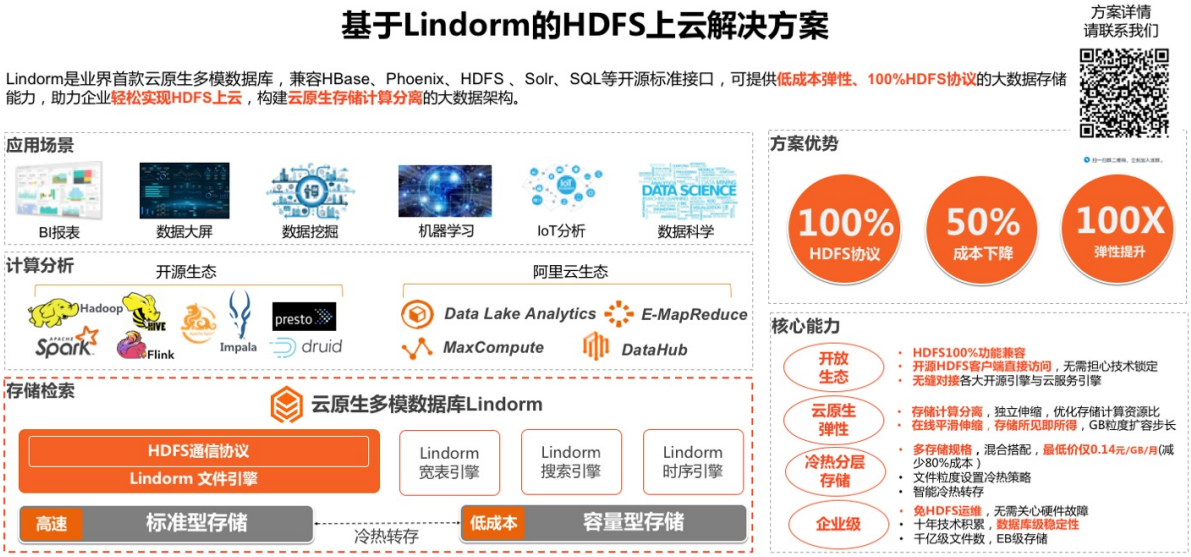


17.解决方案

17.1. HDFS云原生上云方案

Lindorm文件引擎是面向海量非结构化数据的分布式文件存储服务，具有弹性、低成本的优势，提供100%的HDFS协议和使用体验，支持企业大数据中的HDFS无缝平滑上云，构建云原生时代的存储计算分离的大数据架构。

方案总览



方案优势

- 简单：Lindorm文件引擎支持100%的HDFS协议，应用零改造。
- 低成本：相比自建，TCO最低可下降50%以上。
- 弹性：存储计算资源解耦，独立伸缩，并大幅提升弹性能力，轻松实现云原生架构。
- 稳定：免运维，无需关心硬件故障，由十年技术积累的专家团队保障服务稳定性。
- 更多内容，请参见Lindorm vs HDFS。

实施过程

1. 开通Lindorm文件引擎。
2. 将自建HDFS集群的数据迁移至Lindorm。
3. 将上层计算应用（例如Hadoop/Hive/Spark）依赖的HDFS服务器地址更换为新购买的Lindorm。
4. 下线或收缩自建HDFS集群，例如将集群中的本地盘节点，通过集群伸缩，更换为计算型的ECS节点，升级成存储计算分离的架构。

方案咨询

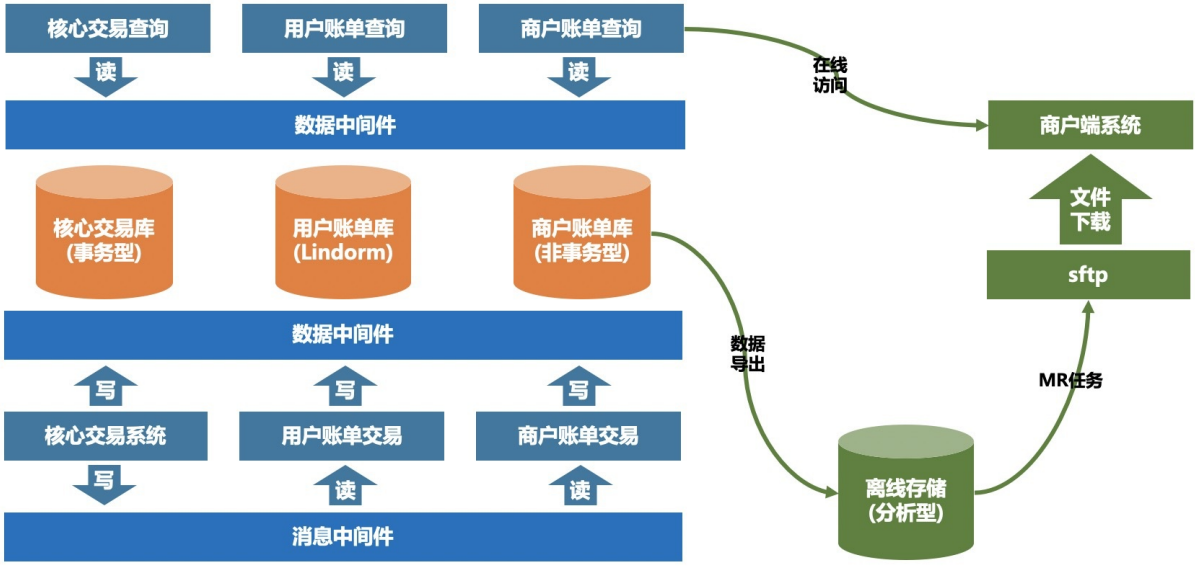
如果您遇到任何问题，欢迎联系专家服务。

17.2. 互联网账单解决方案

随着互联网业务场景的不断丰富，与之对应的交易系统相互垂直，相互独立，账单系统应运而生。基于这样的背景互联网账单系统，从诞生的第一天起附带了一些天然的属性，包括： 只读性、高增长、低成本、高可用、多维度查询。

方案总览

Lindorm作为一款适用于任何规模、多种模型的云原生数据库服务，可以很好的满足账单系统的需求。账单系统架构如下图所示：



方案优势

1. 低成本：集群内单表冷热分离能力、独有的压缩优化能力
2. 极致弹性：原生存储计算分离能力，轻量化扩缩容能力
3. 热点问题：高效的热点响应能力、有效避免存储水位不均衡
4. 高可用：支持强一致、最终一致等不同模式的高可用方案
5. 多维度查询能力：原生全局二级索引、Lindorm Search满足多维度查询需求

方案详情&专家服务

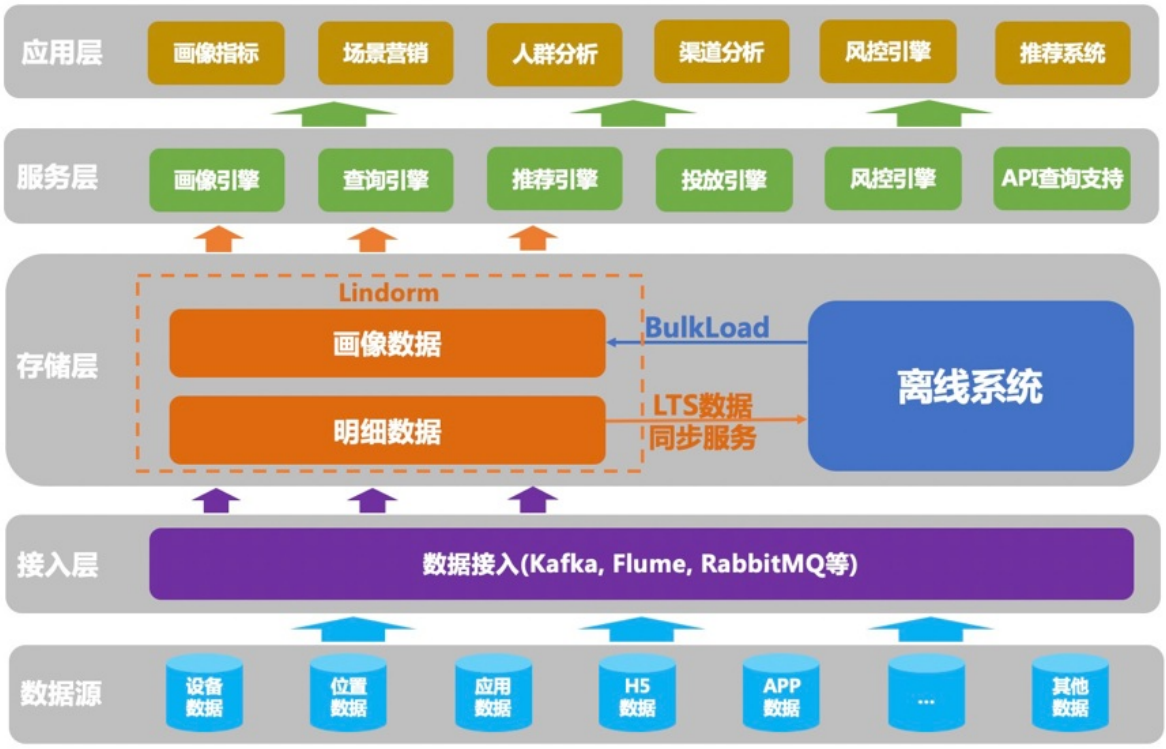
解决方案详情参见：基于Lindorm的互联网账单解决方案
任何问题，欢迎联系专家服务

17.3. 大数据用户画像解决方案

进入互联网，特别是移动互联网时代，用户随时随地都会在网上留下“言”和“行”，如何让这些言行数据“开口说话”就变得越来越重要。用户画像应运而生，而且已经广泛的应用到精准营销、推荐系统、广告投放、风控、智能客服等等领域。用户画像数据具有如下特征：数据量大、高并发读写、明细数据需要归档、大数据量回流、有动态列需求、查询种类多而且复杂。

方案总览

作为面向大数据场景的半结构化、结构化存储系统，Lindorm可以很好的满足用户画像：没有强事务要求，大数据量、高并发读写场景这样的业务特征。其架构如下图所示：



方案优势

1. 低成本：集群内单表冷热分离能力、独有的压缩优化能力、多样化存储类型支持
2. 高性能吞吐：不同场景下相比社区版本HBase2.0有数倍提升
3. 实时增量归档：通过LTS(Lindorm Tunnel Service)可以实时归档数据到离线存储
4. Bulkload技术：LSM技术架构使其具备高效、低影响的数据加载能力
5. 动态列：很好的满足用户画像结构变化频繁的业务特点
6. 多维度&复杂查询：原生全局二级索引、Lindorm Search满足多维度查询需求

方案详情&专家服务

解决方案详情参见：基于Lindorm的大数据用户画像解决方案

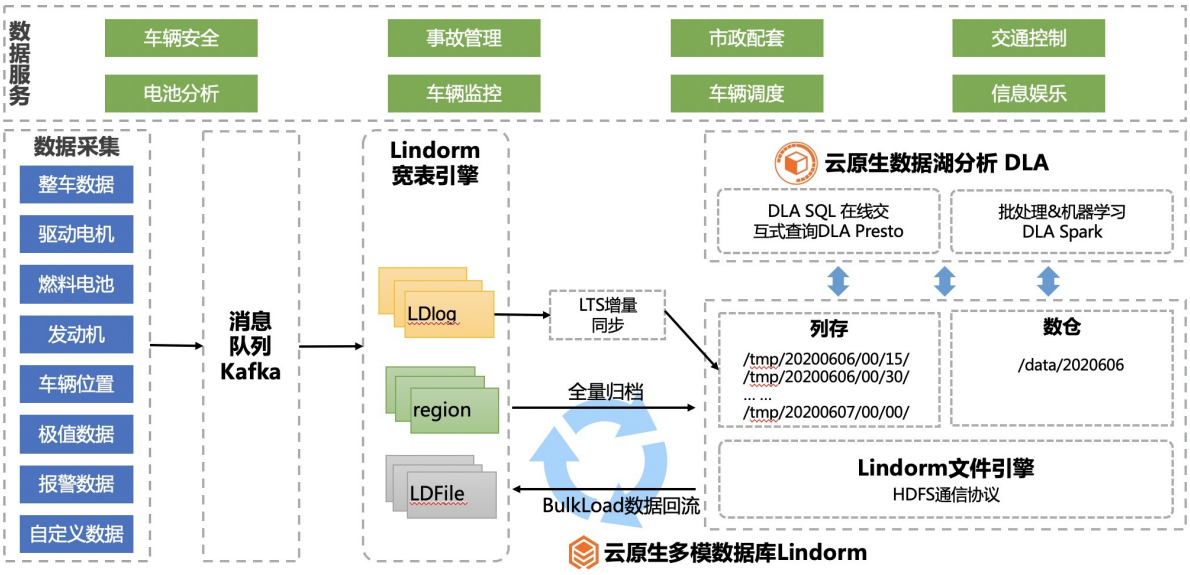
任何问题，欢迎联系专家服务

17.4. 车联网数据存储处理方案

本文从车联网的定义出发，结合行业趋势和国家规范，给出车联网通常需要采集的数据以及可以提供的服务。根据对车联网的业务特征的分析给出了为什么Lindorm作为一个数据存储为什么是车联网业务的合适选择。

方案总览

Lindorm是一款适用于任何规模、多种模型的云原生数据库服务，支持海量数据的低成本存储处理和弹性按需付费，提供宽表、时序、搜索、文件等多种数据模型，兼容HBase、Cassandra、Phoenix、OpenTSDB、Solr、SQL等多种开源标准接口，可以很好的满足车联网业务的需求。车联网架构如下图所示：



方案优势

1. 动态schema的能力可以很好的满足车联网不断变更的采集点需求
2. 超强的弹性&水平扩展能力可以很好地满足车联网业务采集频率变更带来的数倍甚至数量级的流量突增
3. 高压缩比、冷热分离，支持HDD/OSS等低速高密存储可以低成本满足车联网业务超长甚至永久的数据保留周期
4. Lindorm宽表引擎、Lindorm文件引擎、Lindorm LTS数据同步服务、DLA数据湖分析具备了一体化的数据存储、实时归档、离线数据高压缩比存储、高效数据分析的能力
5. Lindorm LSM架构使得bulkload数据回流成为可能，高效并且对在线业务影响很低

方案详情&专家服务

解决方案详情参见：基于Lindorm的车联网数据存储处理方案
任何问题，欢迎联系专家服务

17.5. 低成本RDS历史库

云原生多模数据库Lindorm和关系型数据库同时使用时可以满足低存储成本，方便运维，弹性伸缩等需求，本文介绍其方案架构和优势。

背景

在移动互联网高度发达的今天，每天都会有大量的业务数据产生，随着时间的积累和业务的发展数据量成指数级增长，同时历史数据的访问频率随着时间的推移却越来越低。这些数据全部存储在关系型数据库中会带来了一系列的问题。

挑战：

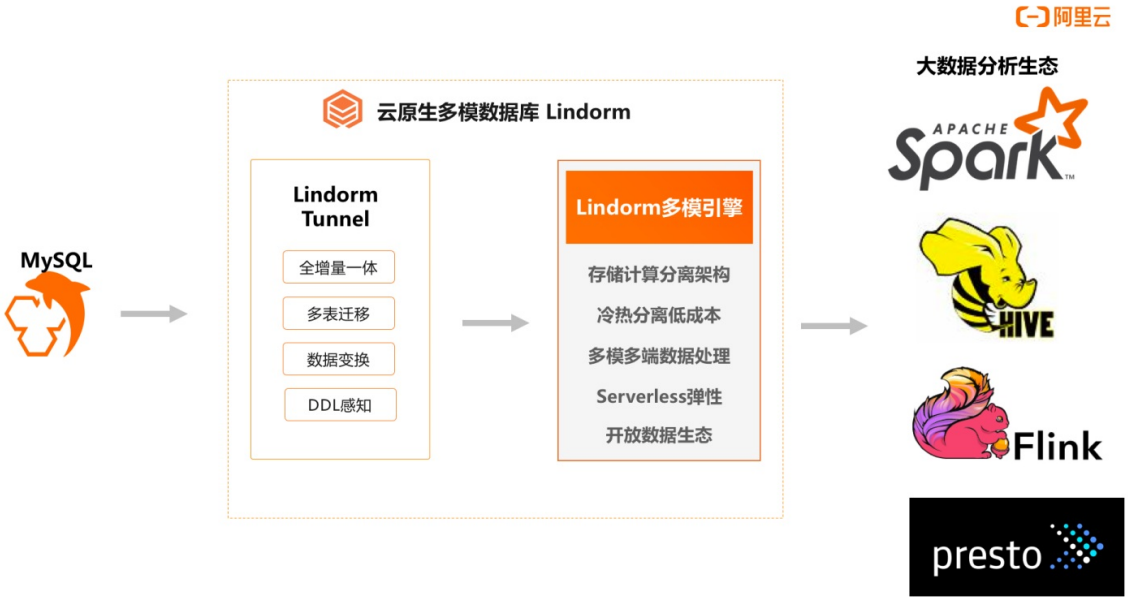
- **存储成本的升高**，存储成本和数据量成正比，数据量指数级增长带来了存储成本的指数级增加。
- **查询性能的下降**，单实例存储空间超过TB以后查询性能将会急剧下滑。
- **运维复杂度高**，通过分库分表的方式来解决数据量增加造成性能下降，给运维和业务开发增加了巨大的成本。

需求：

- **存储成本可控**，历史数据的存储成本是在线数据的10分之一。
- **能够弹性的伸缩**，计算和存储能力自动水平扩展，不再依赖分库分表解决运维难的痛点。
- **修改schema成本低**，支持快速修改schema或者动态schema，解决历史库schema变更时间长的问题。
- **改造成本低**，可以支持SQL进行访问。
- **满足实时查询需求**，例如消费账单、聊天记录等场景历史数据的查询RT要能够和在线数据的查询RT接近。
- **能够满足数据分析需求**，历史数据访问频度低，但是在某些业务场景下需要对全量数据进行挖掘分析，比如支付宝年账单。

云原生多模数据库Lindorm（HBase增强版）能够同时满足存储成本低廉、运维简单、弹性伸缩、性能稳定等需求可以和关系型数据库完美组合，是实时低成本历史库的最佳选择。

方案架构



- LTS数据同步服务对接MySQL等关系行数据库支持数据的全增量一体化同步，同时提供多表迁移、数据变化、DDL感知等企业级同步能力，帮助用户简单高效完成数据的迁移。
- 云原生多模数据库Lindorm提供海量数据的低成本存储能力（0.12元/GB/月）、弹性伸缩按需付费、多模多端处理能力满足用户多种场景下的数据存储需求，同时可以无缝对接Spark、Hive、Flink、Presto等开源分析生态满足用户对数据的复杂分析需求，最大程度挖掘数据价值。

方案优势

简单易用

- 数据搬迁配置可视化，分钟级完成迁移配置。
- 全增量一体化，最大程度降低用户使用成本。
- 支持多表迁移、数据变换等能力帮助用户轻松完成多表合并、字段组合变化等需求。
- 全面的监控报警，保证数据同步稳定性。

极致性价比

- 支持容量型**存储成本底至0.12元/GB/月**，在极致优化存储成本的同时通过系统内置缓冲加速层，让查询实时性仍有较大的保障，是实时历史库的最佳存储选择。
- 在性能上，Lindorm宽表引擎在吞吐延迟上做了非常多的突破，其**基准性能是开源HBase的7倍**，具体请参考测试结果。Lindorm时序引擎融入了许多创新型的高性能结构设计，其基准性能在目前的**信通院榜单中处于第一**的位置，大幅领先于其他专用时序数据库。
- **支持智能冷热分离**，针对数据随着时间线逐渐热变冷的场景，典型如监控、社交聊天、交易账单等，Lindorm内部将自动识别数据的冷热，并进行分离存储到高性能介质和低成本介质（两者之间的单价成本差可高达10:1），而用户读写访问保持完全透明，并且热数据的访问性能还能有所加速。
- 支持自适应压缩，针对数据的不同类型和特点，系统将自动选择混合的字典、前缀、Delta、熵编码等压缩算法，相比业界通用算法，**整体压缩率提升10%~30%。**

云原生弹性

- Lindorm基于**存储计算分离**的架构，支持计算资源、存储资源的独立弹性伸缩，最大程度避免资源浪费。
- **同时提供Serverless服务，实现按需即时弹性、按使用量付费的能力**。Lindorm Serverless基于多租户隔离、智能调度、弹性IaaS底座构建，具备企业级SLA保障，满足内部大部分业务的可用性要求，从而让一线同学大幅降低容量管理的运维负担，消除因流量波动导致的稳定性风险。

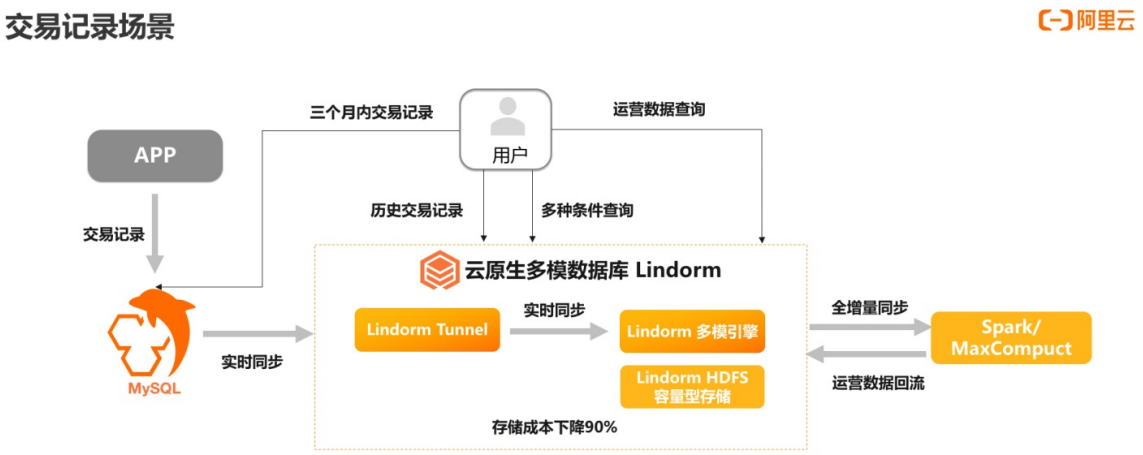
多模&检索

- **兼容HBase、Phoenix（SQL）、Cassandra（CQL）**等主流开源标准接口，最大程度降低用户改造成本，同时具备全局二级索引、多维检索、动态列、TTL等能力， 满足元数据、订单、账单、画像、社交、feed流、日志等多种场景需求。
- **支持一键开启搜索引擎**（兼容开源Solr标准接口）， 具备全文检索、聚合计算、复杂多维查询等能力，帮助加速检索查询，满足用户实时复杂分析需求。

大数据生态

- 无缝对接Spark、Hive、Flink、Presto等开源大数据生态产品，支持API访问和文件读取等多种对接方式，简单高效满足用户海量数据分析需求。

典型应用



- 用户交易记录通过App写入MySQL，LTS将MySQL中实时同步到Lindorm，近三个月状态不断变化的记录查询MySQL，三个月以上历史交易记录查询Lindorm，历史交易记录存储在容量型存储中，存储成本下降90%以上。
- 对于用户复杂条件实时检索需求，例如按时间、地点、金额大小、交易备注内容等任意组合查询，可以结合Lindorm搜索引擎的全文检索、聚合计算、复杂多维查询等能力，不需要业务改造就能轻松满足用户需求。
- 通过LTS支持将Lindorm账单数据同步到Spark/MapCompuct等离线计算平台进行计算按照业务需求分析生成运营报表数据，然后回流到Lindorm供用户进行实时查询。

使用说明

- RDS全增量同步
- 存的起，看得见—云原生多模数据库Lindorm技术解析

17.6. Lindorm for Cassandra应用实践依赖

本文介绍如何通过云原生多模数据库Lindorm扩展云数据库Cassandra的性能。

云原生多模数据库Lindorm简介

云原生多模数据库Lindorm是一款适用于任何规模、多种模型的云原生数据库服务，支持海量数据的低成本存储处理和弹性按需付费，提供宽表、时序、搜索、文件等多种数据模型，兼容HBase、Cassandra、Phoenix、OpenTSDB、Solr、SQL等多种开源标准接口，是互联网、IoT、车联网、广告、社交、监控、游戏、风控等场景首选数据库，也是为阿里巴巴核心业务提供关键支撑的数据库之一。



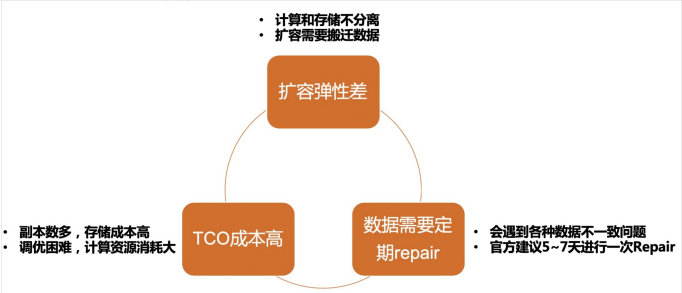
云原生多模数据库Lindorm基于存储计算分离、多模共享融合的云原生架构，具备弹性伸缩、低成本、简单易用、开放、稳定等优势，适合元数据、日志、账单、标签、消息、报表、维表、结果表、Feed流、用户画像、设备数据、监控数据、传感器数据、小文件、小图片等数据的存储和分析。其核心能力包括：

- 融合多模：支持宽表、时序、搜索、文件四种模型，提供统一联合查询和独立开源接口两种方式，模型之间数据互融互通，帮助应用开发更加敏捷、灵活、高效。
- 性价比：支持千万级高并发吞吐、毫秒级访问延迟，并通过高密度低成本存储介质、智能冷热分离、自适应压缩，大幅减少存储成本。
- 云原生弹性：支持计算资源、存储资源独立弹性伸缩，并提供按需即时弹性、按使用量付费的Serverless服务。
- 开放数据生态：提供简单易用的数据交换、处理、订阅等能力，能够高度兼容MySQL、Spark、Flink、Kafka等系统。

开源Cassandra挑战

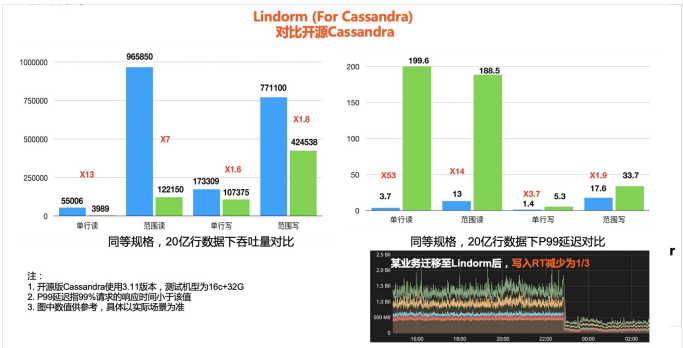
开源Cassandra是基于Amazon DynamoDB和Google Bigtable设计的一款分布式NoSQL数据库，具备无中心、一致性可调、提供类SQL查询语言CQL等优点。但在实际使用中，Cassandra存在一些难以解决的挑战。比如Cassandra存储和计算不分离，一旦需要扩容，需要搬迁数据，扩容持续时间长，无法快速应对业务突发流量。而Lindorm是存储计算分离架构，可以实现快速弹性。

另外，Cassandra 需要定期对所有副本进行全量数据修复，否则会出现”幽灵key“等问题，但是修复过程会因为消耗大量系统资源从而影响服务稳定性。而Lindorm由底层的存储组件负责一致性，当您的数据写入时，就会按照您设定的副本数写入，不会有数据不一致问题，也不用定期修复。

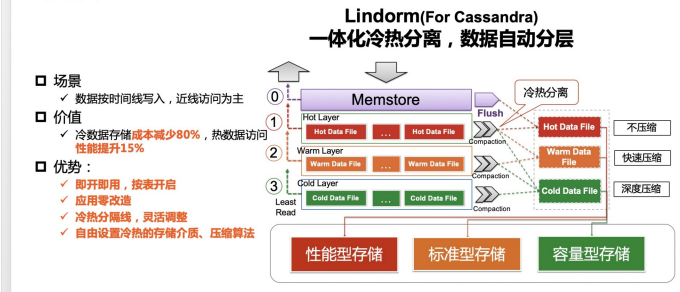


Lindorm For Cassandra特性

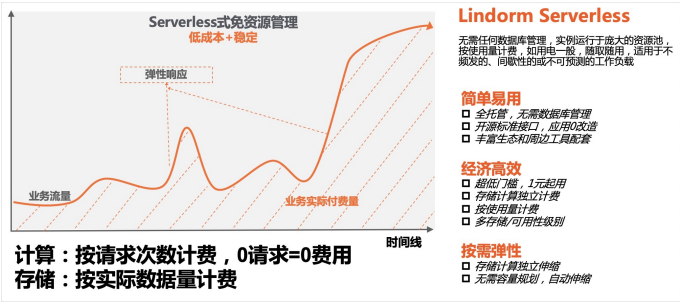
- 更强性能：相比开源Cassandra，Lindorm For Cassandra在大规模数据下吞吐量更高，延迟更低。



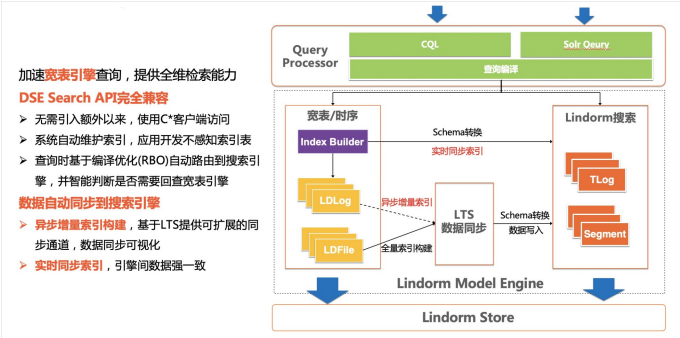
- 透明冷热分离：Lindorm For Cassandra采用自由设置冷热的存储介质、压缩算法，减少冷数据存储成本，提升热数据访问性能，实现一体化冷热分离、数据自动分层。



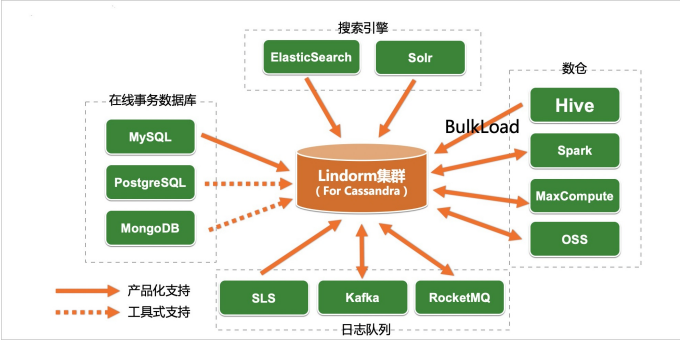
- 按需计费：Lindorm for Cassandra提供集群版和serverless版产品形态，serverless版采用serverless式免资源管理，根据业务需求量弹性响应，按请求次数、实际数据库量计费，实现简单易用、经济高效、按需弹性的计费特性。



- 搜索宽表一体化：Lindorm for Cassandra 通过全文索引加速宽表引擎查询，对外统一提供CQL语言的访问方式。



- 丰富的数据通道：Lindorm for Cassandra具备丰富的数据通道，比如搜索引擎、在线事物数据库、数仓、日志队列等。



- 更多企业级特性：Lindorm for Cassandra具备更多的企业级特性，详情请参见下表。更多企业级特性请参考Lindorm产品首页。

特性	Apache Cassandra	Lindorm for Cassandra
基础功能	KV	多模（KV/Table/SQL/Search/Timeseries/File）
性能	不涉及	最高14倍性能，10%的延迟，2倍压缩率。
开源标准	CQL	兼容CQL（大部分情况下无需改造代码）

特性	Apache Cassandra	Lindorm for Cassandra
一致性	可调一致性，需定期repair。	可调一致性（无需repair数据）
服务模式	自建	集群托管或者Serverless可选
可靠性	无SLA保障，开源软件BUG需要自行修复。	SLA保障，并具备主备双或、备份、异地容灾等能力。
冷热分离	不支持	透明冷热分离降低成本
全文检索	不支持	兼容CQL语法
其他企业级功能	不涉及	支持全球多活、备份恢复等企业级能力。
弹性	存储计算不分离，扩缩容需要搬迁数据，弹性能力差。	包括以下几点： ○ 支持Scale Up：升配和磁盘扩容。 ○ 支持Scale Out：产品化扩缩节点。 ○ 支持Serverless：自动弹性。
生态	开源生态，打通依赖工具或程序开发。	支持开源生态和阿里云生态，与MySQL、Spark、OSS、MaxCompute等深度产品化集成。
使用成本	高，取决于产品维护的研发投入。	低

如果您需要将Cassandra数据导入至Lindorm宽表，具体操作请参见将Cassandra数据导入至Lindorm宽表。

18.实践教程

18.1. 从云HBase标准版/自建开源HBase迁移到Lindorm宽表引擎

本文介绍将云数据库HBase标准版或者自建开源HBase的数据迁移到Lindorm宽表引擎表格中。

背景信息

Lindorm是一款适用于任何规模、多种模型的云原生数据库服务，支持海量数据的低成本存储处理和弹性按需付费，提供宽表、时序、搜索、文件等多种数据模型，兼容HBase、Cassandra、Phoenix、OpenTSDB、Solr、SQL等多种开源标准接口，是互联网、IoT、车联网、广告、社交、监控、游戏、风控等场景首选数据库，也是为阿里巴巴核心业务提供关键支撑的数据库之一。Lindorm基于存储计算分离、多模共享融合的云原生架构，具备弹性、低成本、简单易用、开放、稳定等优势，适合元数据、日志、账单、标签、消息、报表、维表、结果表、Feed流、用户画像、设备数据、监控数据、传感器数据、小文件、小图片等数据的存储和分析。

数据迁移

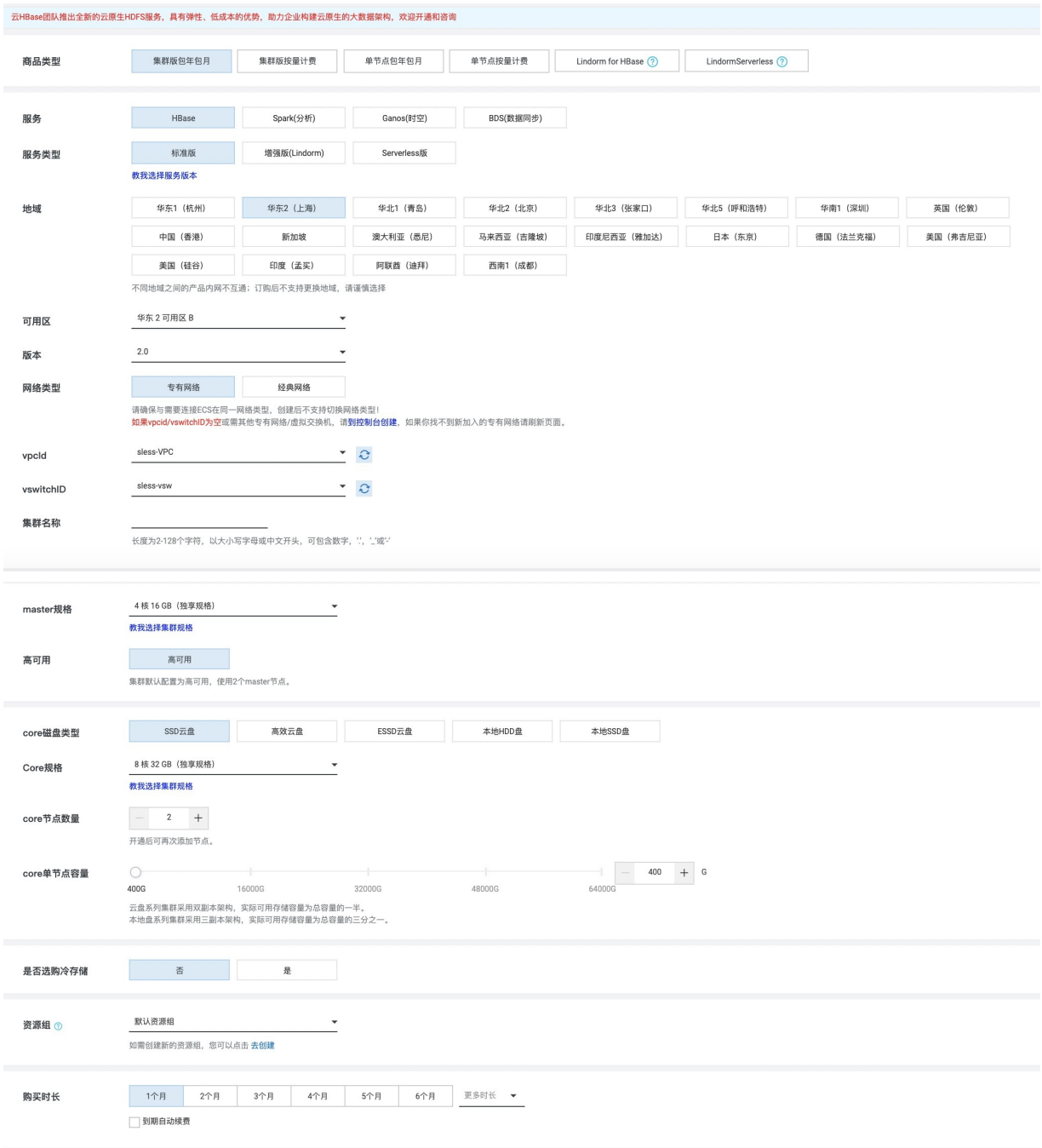
Lindorm Tunnel Service服务，简称LTS（原BDS），该服务是面向Lindorm或者HBase业务场景特点深度定制的数据生态服务。支持简单易用的数据交换、处理、订阅等能力，满足用户的数据迁移、实时订阅、数湖转存、数仓回流、单元化多活、备份恢复等需求，实现面向Lindorm的一站式数据生态服务。详情请参见LTS服务介绍。
通过LTS可以实现从云HBase标准版/HBase社区版向云HBase增强版的不停机迁移。

2.1 准备工作

创建待迁移的云HBase标准版集群或者社区版HBase，对于生产集群的迁移可忽略该步骤。
2.1.1 登录云HBase控制台，选择正确的地域并点击“创建HBase集群”按钮。



2.1.2 根据需要选择合适的集群创建选项。
1）商品类型：包年包月或集群版按量付费。
2）服务及服务类型：HBase、标准版。
3）地域和可用区：根据实际情况进行相应选择。
4）网络：专有网络，并选择相应的VPC。 关于VPC参见：什么是专有网络。
5）集群规格：根据需要选择合适的master、core节点规格、core节点数量及磁盘类型及磁盘容量。



2.1.3点击“立即购买”进入“确认订单”页面，勾选服务协议后点击立即开通开始集群创建。



集群创建约需要30分钟左右的时间，请耐心等待。

2.1.4 在同可用区同VPC下创建一台ECS，作为HBase客户端。注意需要带公有网络，以保证后续可以直接登录并通过AHBench完成测试表及数据导入。ECS实例创建完成后可以执行下重置密码，方便后续使用。



2.1.5 登录ECS，创建用户压测的目录，下载压测工具AHBench。压测工具详见：AHBench。

2.1.6 编辑AHBench/conf/hbase-site.xml文件，配置需要测试的HBase集群地址。 配置样例如下图：

```
hadoop@[redacted] ~/wuqi/AHBench/conf $ cat hbase-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
/**
 *
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
-->
<configuration>
    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>hb-[redacted]-master1-001.hbase.rds.aliyuncs.com:2181,hb-[redacted]-master3-001.hbase.rds.aliyuncs.com:2181,hb-[redacted]-master2-001.hbase.rds.aliyuncs.com:2181</value>
    </property>
</configuration>
```

HBase集群地址可通过HBase控制台集群页面里的数据库连接页面获取到，如下图所示：



2.1.7 编辑文件AHBench/conf/ahbench-env.properties，打开HBASE_VERSION参数。另外，需要保证Java可以调用。如下图所示：

```
~/wuqi/AHBench/conf $ which java
/opt/taobao/java/bin/java
~/wuqi/AHBench/conf $ cat ahbench-env.properties
# EDIT This file for test environment settings

# JDK installed path
# If java(executable) is in the system's PATH, The framework will set the JAVA_HOME automatically.
# JAVA_HOME=/usr/lib/jvm/

# Specify target hbase cluster's version here
#    for HBase1.x, set HBASE_VERSION=1
#    for HBase2.x or HBaseUE, set HBASE_VERSION=2
HBASE_VERSION=2
~/wuqi/AHBench/conf $
```

2.1.8 通过HBase控制台集群页面里访问控制页面维护HBase集群白名单，加入ECS实例IP。



2.1.9 启动测试，创建待迁移测试表及数据。表情况详见HBase集群Master页面。

User Tables | System Tables | Snapshots

9 table(s) in set. [Details]

Namespace	Table Name	Online Regions	Offline Regions	Failed Regions	Split Regions	Other Regions	Table Size	Table Files	Description
default	ahbenchtest-read	200	0	0	0	0	48.78 GB	199	'ahbenchtest-read', {NAME => 'f', DATA_BLOCK_ENCODING => 'DIFF', COMPRESSION => 'SNAPPY'}
default	ahbenchtest-read-1	200	0	0	0	0	48.78 GB	199	'ahbenchtest-read-1', {NAME => 'f', DATA_BLOCK_ENCODING => 'DIFF', COMPRESSION => 'SNAPPY'}
default	ahbenchtest-write	200	0	0	0	0	398 MB	199	'ahbenchtest-write', {NAME => 'f', DATA_BLOCK_ENCODING => 'DIFF', COMPRESSION => 'SNAPPY'}
default	ahbenchtest-write-1	200	0	0	0	0	398 MB	199	'ahbenchtest-write-1', {NAME => 'f', DATA_BLOCK_ENCODING => 'DIFF', COMPRESSION => 'SNAPPY'}

HBase Master页面可通过HBase控制台集群页面里数据库连接页面进入。



2.2 创建目标集群：Lindorm

2.2.1登录云原生多模数据库Lindorm控制台，选择正确的地域并点击“创建”按钮。



2.2.2 根据需要选择合适的集群创建选项。

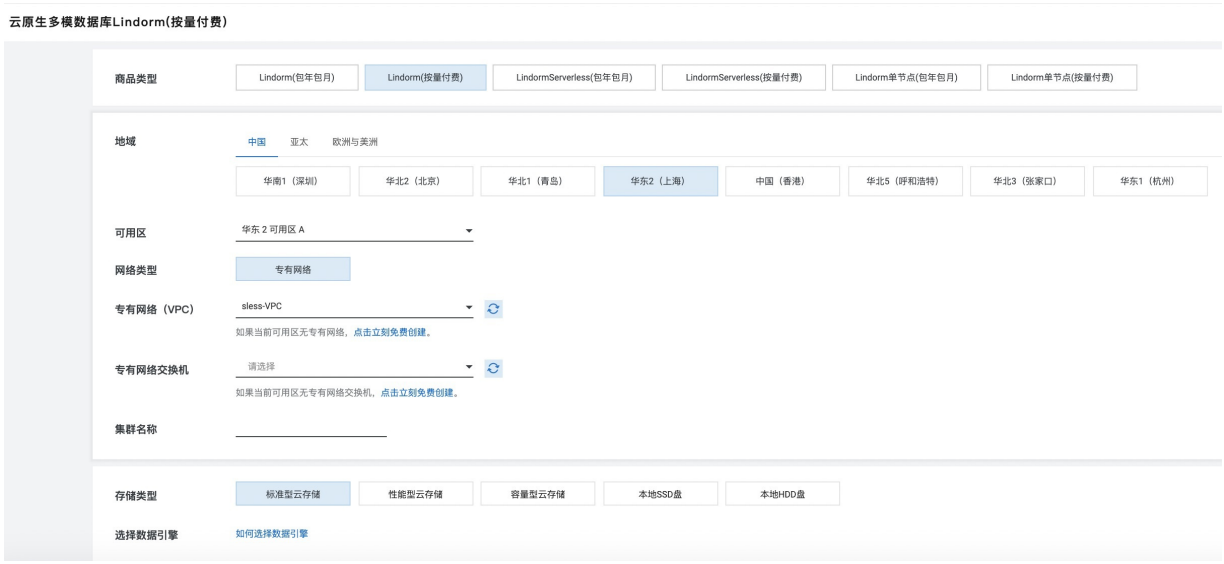
1） 商品类型：包年包月或集群版按量付费。

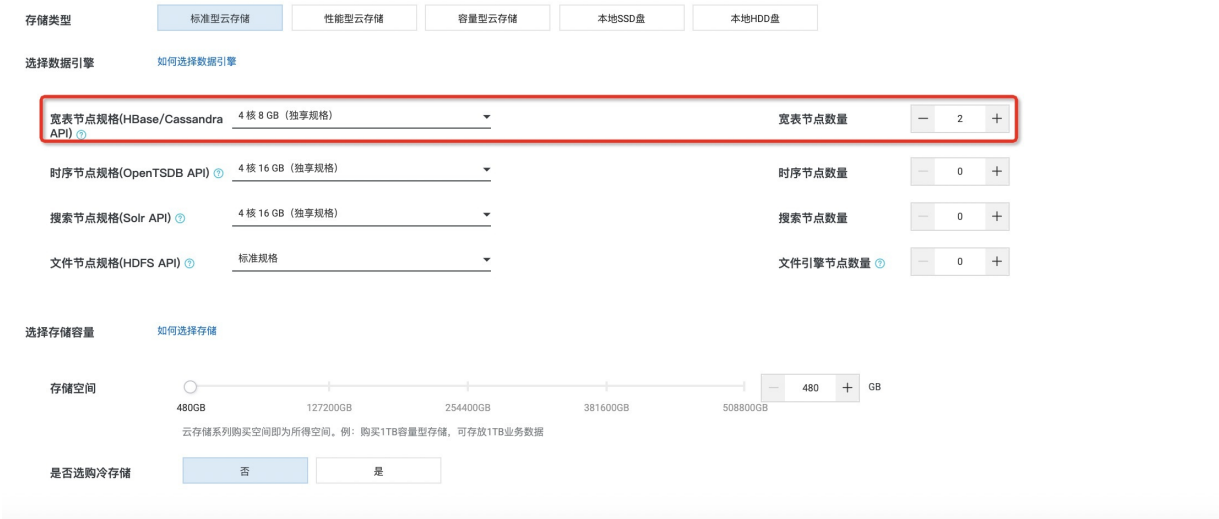
2） 地域和可用区：根据实际情况进行相应选择。

3） 网络：专有网络，并选择相应的VPC。 关于VPC参见：什么是专有网络。

4） 集群规格：根据需要选择合适的存储类型、“宽表“数据引擎及节点规格和数量、存储容量。

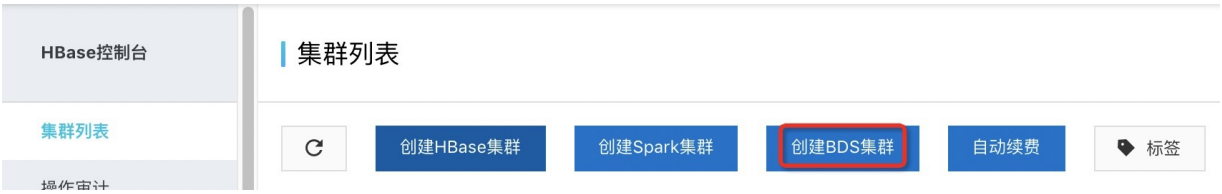
集群创建约需要30分钟左右的时间，请耐心等待。





2.3 创建LTS集群

2.3.1登录云HBase控制台，选择正确的地域并点击“创建BDS集群”按钮。



2.3.2根据需要选择合适的集群创建选项。

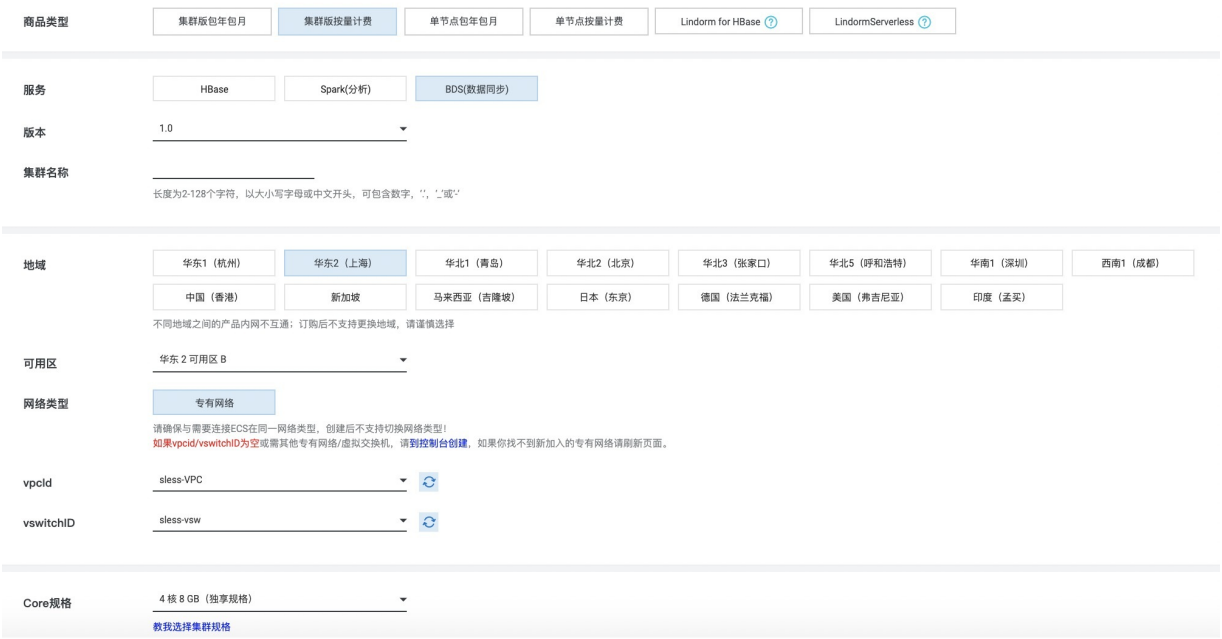
1）商品类型：包年包月或集群版按量付费。

2）服务：BDS。

3）地域和可用区：根据实际情况进行相应选择。

4）网络：专有网络，并选择相应的VPC，建议和HBase集群在同一VPC下。

5）集群规格：根据需要选择合适的core节点规格、core节点数量。



2.4 关联待迁移的源、目标集群

2.4.1登录HBase控制台，从控制台可以看到源HBase集群及LTS集群。



2.4.2 登录Lindorm控制台，从控制台可以看到目标Lindorm集群。



2.4.2打开LTS集群页面，进行数据源关联。



点击“添加数据源”，在弹出窗口中选择正确的地域及实例后点击确定按钮即可完成关联。
备注：
1）自动关联仅适用于LTS、HBase集群在同一VPC下的情况，如在不同的VPC下，需要通过手动操作来完成集群关联，详见：HBase数据源添加、HBase增强版数据源添加。
2）对于社区版HBase也需要通过手动操作来完成集群关联。

- 关联标准版集群。



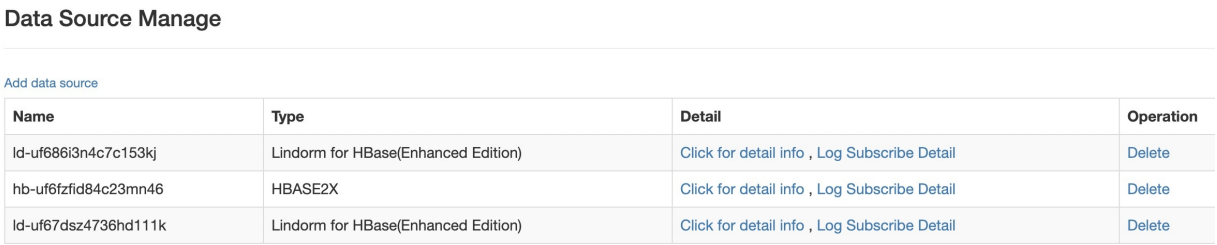
- 关联Lindorm集群。



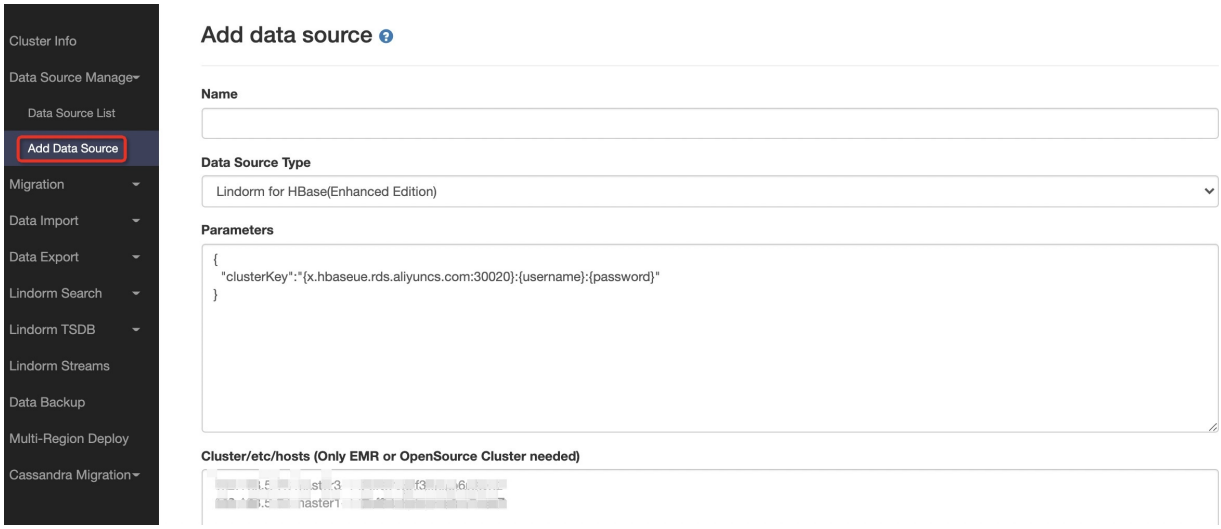
- 完成集群关联后的LTS数据源管理页面如下图所示：



- 完成集群关联后的LTS UI数据源管理页面如下图所示：



- 手动关联集群需要通过LTS UI界面的数据源管理-添加数据源页面完成，如下图所示：

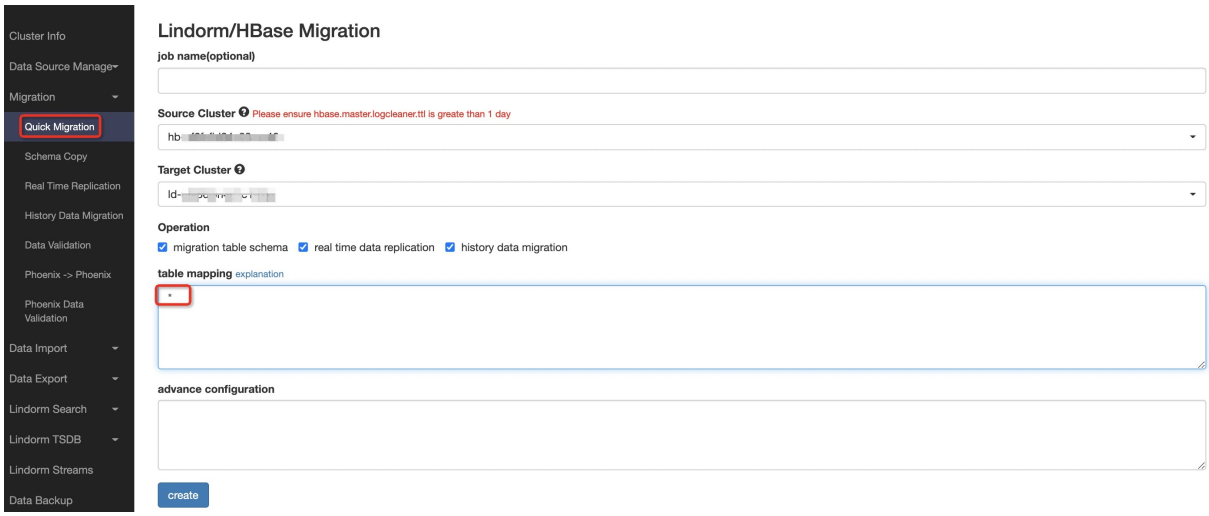


2.5 完成数据迁移

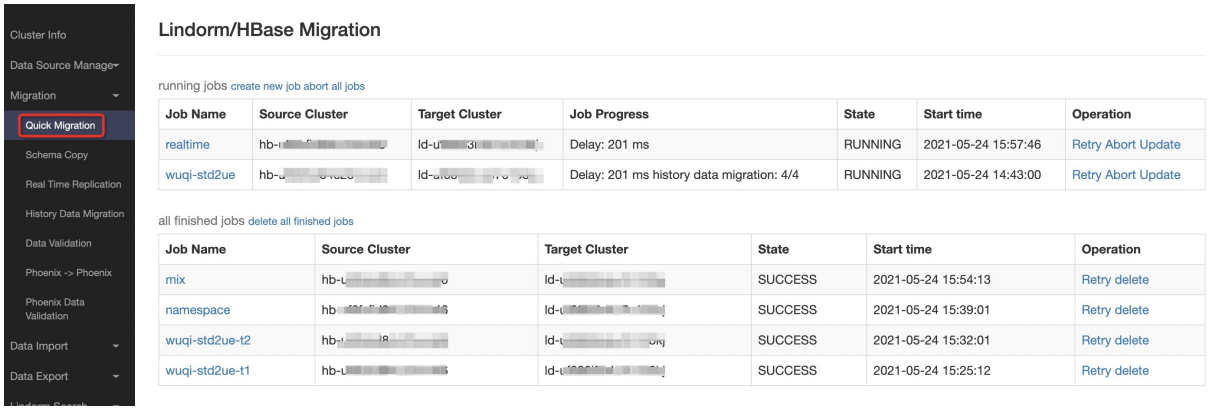
确认数据源关联完成后，下一步就可以创建迁移任务进行数据迁移了。
在LTS UI，打开快速迁移页面，点击创建新任务打开任务配置页面，如下图所示。根据实际情况。

- 定义迁移任务名称。
- 选择已经关联好的源、目标集群。
- 勾选迁移动作：对于集群的整体迁移可以同时勾选迁移表结构、实时数据复制、历史数据迁移。
- 可以支持的迁移方式。

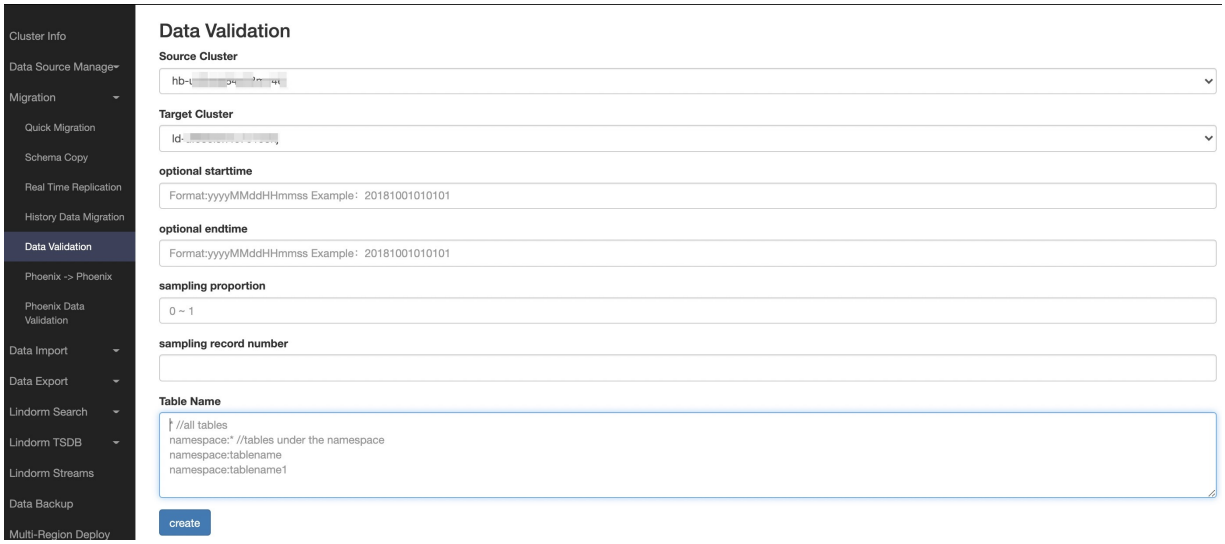
整个集群迁移： *
某个Namespace下所有表：namespace:*
迁移部分表：namespace1:table1 namespace2:table2 （不同表通过换行进行分割）
迁移表的部分列：tableA {"cols": ["f:col1", "f:col2"]}
迁移表期望做namespace和表名的rename： namespaceA:tableA/namespaceB:tableB
注意：如果没有写明namespace则为default。



- 填写完成后点击创建，可以在该页面看到提交后的任务运行情况



- 数据验证。数据迁移完成后可以创建数据抽样验证任务，确认源、目标集群的数据一致性。点击数据验证按钮，创建新任务，输入：源、目标集群、抽样比例、表名等



数据验证通过后，即代表集群迁移（包括历史数据和实时增量同步）完成，并且实时同步任务要始终保持运行中状态，直到流量切换到新集群。

- 为了保证集群迁移可回滚，需要配置一个反向的实时同步任务，即把源、目标交换的实时同步任务。
- 其他注意事项：

1）对于社区版HBase需要调整合适的HLog保留周期，以保证数据同步链路不会因为HLog被删除而中断。
2）对于自建HBase集群如果开通了kerberos的，请联系云HBase值班同学协助完成迁移。
3）如果有通过Bulkload方式做数据导入的，因为该方式下不产生HLog，意味着数据无法完成集群间的同步，请在新老集群同时配置数据导入任务，直至确认切流完成，老集群下线。

3. 客户端配置

对于云HBase标准版和云Lindorm都建议使用阿里云提供的客户端。详见：HBase Java SDK下载。
对于仍然希望使用社区版HBase客户端的客户，可以有两种选择：历史版本（Maven方式）或者历史版本（Jar包替换）。

4. 集群切换

正式切换流量简要步骤：
1） 应用端预发布环境可以正常读写新的云HBase增强版集群。
2） 写入新集群的数据可以通过反向同步链路将数据复制到老集群。
3） 新老集群的复制链路正常，实时同步延迟较低（正常应该小于1秒）。
4） 选择业务低峰期进行流量切换。

18.2. 通过LTS批量加载离线数据到Lindorm

本文介绍通过LTS批量将离线数据导入至云原生多模数据库Lindorm。

功能特点

云原生多模数据库Lindorm通过LTS（LindormTunnel Service）服务提供了从MaxCompute、Hive、AWS S3等离线存储以直接加载文件方式（bulkload）导入数据到Lindorm的能力，这种导入方式相比于普通的导入方式（例如开源的DataX）有如下优势：

- 效率高：数据直接写入Lindorm DFS文件，跳过在线写入需要生成WAL、Flush、RPC转发等步骤，资源消耗更低，相比API写入数据导入效率最高提升10倍以上，沉淀多个性能优化专利。
- 稳定性高：数据的转化、编码、压缩都在离线集群完成，极少占用线上服务资源，同时智能感知Lindorm表属性（block encoding，压缩），分区对齐、本地化率，文件数量监控等，使得完成海量数据导入的同时提供稳定的毫秒级在线查询服务。
- 更可控：阿里内部五年验证，分布式调度，速度线性可控。

准备工作

1. 购买LTS集群，保证LTS与云原生多模数据库Lindorm实例属于同一个VPC。
2. 联系Lindorm值班，开通Lindorm的HDFS端口，开启LTS的加载文件功能。
3. 准备配置文件。配置文件为JSON类型，分为读配置和写配置，S3和MaxCompute的读写配置文件示例如下。
   - 读配置
     - 数据源类型为S3的读配置

```
{
  "file.type":"csv",
  "accessKey": "test",
  "secret_key":"test",
  "endPoint":"http://localhost:8001",
  "bucket":"testbucket",
  "region":"test",
  "fileKey":"",
  "zipped":"false",
  "column.separator":",",
  "column": [
    "pk|String",
    "value|String"
  ],
  "pks":"pk|String"
}
```

     - **数据源类型为MaxCompute的读配置**

```
{
    "accessId":"****",
    "accessKey":"****",
    "project":"****",
    "odpsServer":""http://service.cn-shanghai.maxcompute.aliyun-inc.com/api ",
    "tunnelEndPoint":"http://dt.cn-shanghai.maxcompute.aliyun-inc.com"
    "partition": [
        "ds=???"
         ],
    "column": [
        "pk",
        "value"
    ],
    "table":"xx"
 }
```

> **说明**
> `odpsServer` 和 `tunnelEndPoint` 填写方法请参见各地域Endpoint对照表。

   - **目标数据库为Lindorm的写配置**

> **说明**
> 写配置比较复杂，如果您需要配置请联系技术支持，方法请参见技术支持。

```
{
    "configuration": [
      {
        "bulkerMode":"himporter2",
        "himporter.ipc.call.timeout": "600000",
        "zookeeper.znode.parent":"/test",
        "lindorm.zookeeper.quorum":"test.tbsite.net:2181",
        "lindorm.client.seedserver":"test.tbsite.net:30020",
        "lindorm.client.namespace":"test",
        "lindorm.client.username":"test",
        "lindorm.client.password":"test",
        "fs.defaultFS":"hdfs://test",
        "dfs.nameservices":"test",
        "dfs.client.failover.proxy.provider.test":"org.apache.hadoop.hdfs.server.namenode.ha.Configure
dFailoverProxyProvider",
        "dfs.ha.namenodes.hdfscluster-test":"nn1,nn2",
        "dfs.namenode.rpc-address.test.nn1":"test.tbsite.net:8020",
        "dfs.namenode.rpc-address.test.nn2":"test.tbsite.net:8020",
        "importer.job.concurrency": "15"
      }
    ],
   "columns": [
    "pk",
    "value"
    ],
    "nullMode":"skip",
    "namespace":"test",
    "writeBufferSize":2097152,
    "encoding":"utf-8",
    "lindormTable":"test_table"
    }
```

任务配置方式和前提

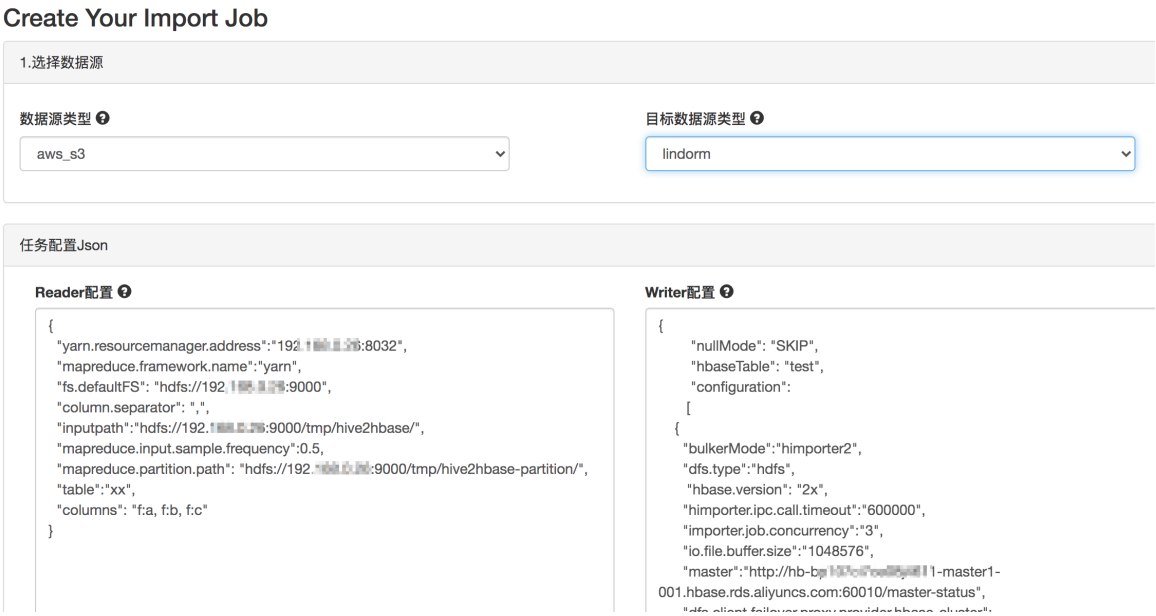
LTS Bulkload数据导入功能支持两种不同的任务配置方式，两种方式使用相同的配置文件。

配置方式	使用场景	前提条件
白屏化任务配置	用于调试任务	已登录LTS操作页面，具体操作请参见登录同步任务。
API任务配置	用于生产任务	LTS的访问地址已开通公网访问方式，开通方法请提交工单。

白屏化任务配置

1. 在LTS操作页面的左侧导航栏中，选择**导入Lindorm/HBase > Bulkload导入**。

2. 单击**创建新任务**，并配置以下参数项。



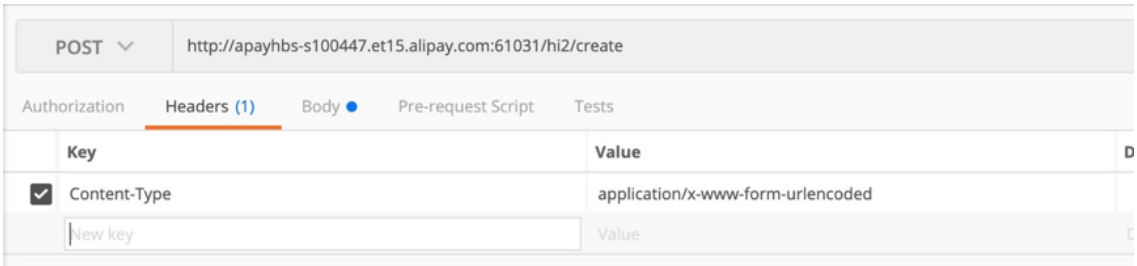
参数项	描述
数据源类型	选择数据源类型，例如**aws_s3**。
目标数据源类型	选择目标数据源类型**lindorm**。
Reader配置	参考准备工作的JSON示例代码输入读配置代码。
Writer配置	参考准备工作的JSON示例代码输入写配置代码。

3. 单击**创建Importer任务**。

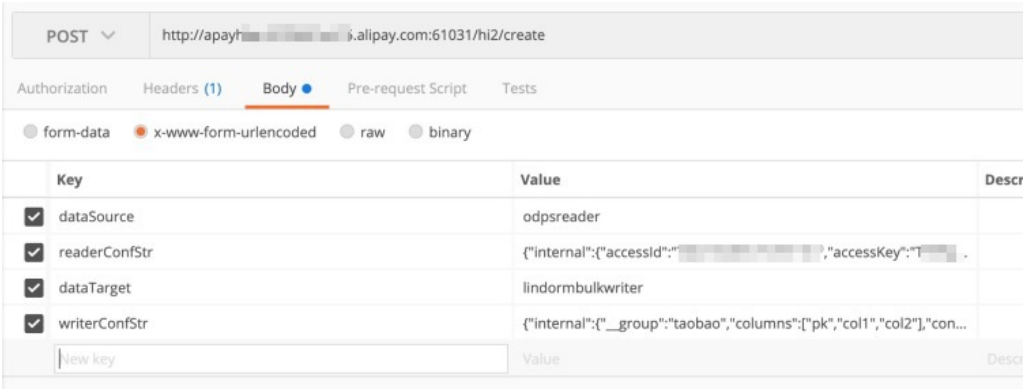
API任务配置

> 说明
> 由于LTS并没有调度能力，所以需要通过API方式在客户端实现数据导入发起操作。

1. 在LTS操作页面修改浏览器的访问地址。HTTP request需要通过公网访问LTS，将访问地址由 `https://bds-****/bds` 修改为 `https://bds-****/api` ，并回车。
2. 创建任务。
   i. 在**Headers**页签下，将**Content-Type**字段的Value设置为 `application/x-www-form-urlencoded` 。



ii. 在**Body**页签下，设置以下四个字段的Value。



Key	Value
dataSource	odpsreader。
readerConfStr	参考准备工作的JSON示例代码输入读配置代码。
dataTarget	lindormbulkwriter。
writerConfStr	参考准备工作的JSON示例代码输入写配置代码。

iii. 执行以下代码发起POST请求，返回**body**内的字符串为任务ID。

```
curl -X POST
http://yourhost:12311/hi2/create
-H 'content-type:application/x-www-form-urlencoded'
-d 'your config string here'
```

3. 查询任务状态。

i. 在**Headers**页签下，将**Content-Type**字段的Value设置为 `application/json` 。

ii. 执行以下代码发起GET请求。

```
curl -X GET
http://yourhost:12311/hi2/<任务ID>/detail
-H'content-type: application/json'
```

返回body内的字符串即任务详情，示例如下。

```
{
  "jobId":"0c84e8ec-6bf0-4e70-8b6a-f92beddb****",
  "status":{
 "status":"LOADED",
 "msg":"job 0c84e8ec-6bf0-4e70-8b6a-f92beddb**** is loaded",
 "progressRate":1.0,
 "version":1
  },
  "version":0
}
```

任务状态status说明如下表。

状态	说明
INIT	任务已创建，等待调度。
PREPARING	数据正在ODPS排序。
PREPARE_FAILED	任务配置错误，ODPS排序失败。
LOADING	正在写数据。
LOAD_FAILED	写数据失败。
LOADED	任务执行成功。
ABORTING	任务正在终止。
ABORT_FAILED	任务终止流程执行失败。
ABORTED	任务已终止。

4. 获取任务统计。执行以下代码发起请求。

```
curl -X GET
http://yourhost:12311/hi2/<任务ID>/statics
-H 'content-type: application/json'
```

> 说明
> - 这个接口适用于Master机器，通过zk获取数据，支持低频访问。
> - 返回参数中，`row` 表示行数，`taskNumber` 和任务的并发数量是一致的。

5. 中断任务。

i. 在Headers页签下，将Content-Type字段的Value设置为 `application/json` 。

ii. 执行以下代码发起POST请求。

```
curl -X POST \
  http://yourhost:61031/hi2/<任务ID>/abort
  -H 'content-type: application/json'
```

返回**body**内的字符串即请求结果，示例如下。

```
{
"message":"SUCCESS"
}
```