ALIBABA CLOUD

阿里云

表格存储Tablestore

功能介绍

文档版本：20220706

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.概述

表格存储（Tablestore）面向海量结构化数据提供Serverless表存储服务，适用于海量账单、IM消息、物联网、车联网、风控、推荐等场景中的结构化数据存储，提供海量数据低成本存储、毫秒级的在线数据查询和检索以及灵活的数据分析能力。

基本概念

在使用表格存储前，您需要了解以下基本概念。

术语	说明
地域	地域（Region）物理的数据中心，表格存储服务会部署在多个阿里云地域中，您可以根据自身的业务需求选择不同地域的表格存储服务。更多信息，请参见表格存储已经开通的Region。
读写吞吐量	读吞吐量和写吞吐量的单位为读服务能力单元和写服务能力单元，服务能力单元（Capacity Unit，简称CU）是数据读写操作的最小计费单位。更多信息，请参见读写吞吐量。
实例	实例（Instance）是使用和管理表格存储服务的实体，每个实例相当于一个数据库。表格存储对应用程序的访问控制和资源计量都在实例级别完成。更多信息，请参见实例。
服务地址	每个实例对应一个服务地址（EndPoint），应用程序在进行表和数据操作时需要指定服务地址。更多信息，请参见服务地址。
数据生命周期	数据生命周期（Time To Live，简称TTL）是数据表的一个属性，即数据的存活时间，单位为秒。表格存储会在后台对超过存活时间的数据进行清理，以减少您的数据存储空间，降低存储成本。更多信息，请参见数据版本和生命周期。

数据存储模型

表格存储提供多种数据存模型，您可以根据业务需求选择对应的模型进行应用。表格存储数据存储模型的详细说明请参见下表。

模型	描述
宽表模型	类Bigtable/HBase模型，可应用于元数据、大数据等多种场景，支持数据版本、生命周期、主键列自增、条件更新、局部事务、原子计数器、过滤器等功能。更多信息，请参见宽表模型。
时序模型	针对时间序列数据的特点进行设计的模型，可应用于物联网设备监控、设备采集数据、机器监控数据等场景，支持自动构建时序元数据索引、丰富的时序查询能力等功能。更多信息，请参见时序模型。
消息模型	针对消息数据场景设计的模型，可应用于IM、Feed流等消息场景。能满足消息场景对消息保序、海量消息存储、实时同步的需求，同时支持全文检索与多维度组合查询。更多信息，请参见消息模型。
Grid模型	针对多维网格数据设计的模型，可实现对多维网格数据的存储、查询和管理。更多信息，请参见Grid模型。

功能

表格存储提供的功能说明请参见下表。

功能	描述
主键列自增	设置非分区键的主键列为自增列后，在写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。
条件更新	只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。
局部事务	创建数据范围在一个分区键值内的局部事务。对局部事务中的数据进行读写操作后，可以根据实际提交或者丢弃局部事务。
原子计数器	将列当成一个原子计数器使用，对该列进行原子计数操作，可用于为某些在线应用提供实时统计功能，例如统计帖子的PV（实时浏览量）等。
过滤器	在服务端对读取的结果再进行一次过滤，根据过滤器中的条件决定返回哪些行。由于只返回符合条件的数据行，所以在大部分场景下，可以有效降低网络传输的数据量，减少响应时间。
多元索引	多元索引基于倒排索引和列式存储，可以解决大数据的复杂查询难题，包括非主键列查询、全文检索、前缀查询、模糊查询、多字段自由组合查询、嵌套查询、地理位置查询和统计聚合（max、min、count、sum、avg、distinct_count、group_by）等功能。
二级索引	支持在属性列创建索引。 通过创建一张或多张索引表，使用索引表的主键列查询，二级索引相当于把数据表的主键查询能力扩展到了不同的列。使用二级索引能加快数据查询的效率。
SQL查询	通过SQL查询功能，您可以对表格存储中数据进行复杂的查询和高效的分析，为多数据引擎提供统一的访问接口。
通道服务	通道服务提供了增量、全量、增量加全量三种类型的分布式数据实时消费通道。可以实现对表中历史存量和新增数据的消费处理。
数据湖投递	表格存储数据湖投递可以全量备份或实时投递数据到OSS数据湖中存储，以满足更低成本的历史数据存储，以及更大规模的离线和准实时数据分析需求。
HBase支持	开源HBase API的Java应用可以通过Tablestore HBase Client直接访问表格存储服务。

2.功能和地域支持列表

通过本文您可以了解表格存储正在邀测和非全地域支持的功能。

邀测功能及支持地域

目前，局部事务为邀测功能，已全地域支持，需要提交工单申请开启。

非全地域支持功能

目前多元索引、通道服务、时序模型、SQL查询和数据湖投递功能未全地域支持。未全地域支持功能的地域支持情况请参见下表。

> **说明** “✓”表示支持，“×”表示不支持。

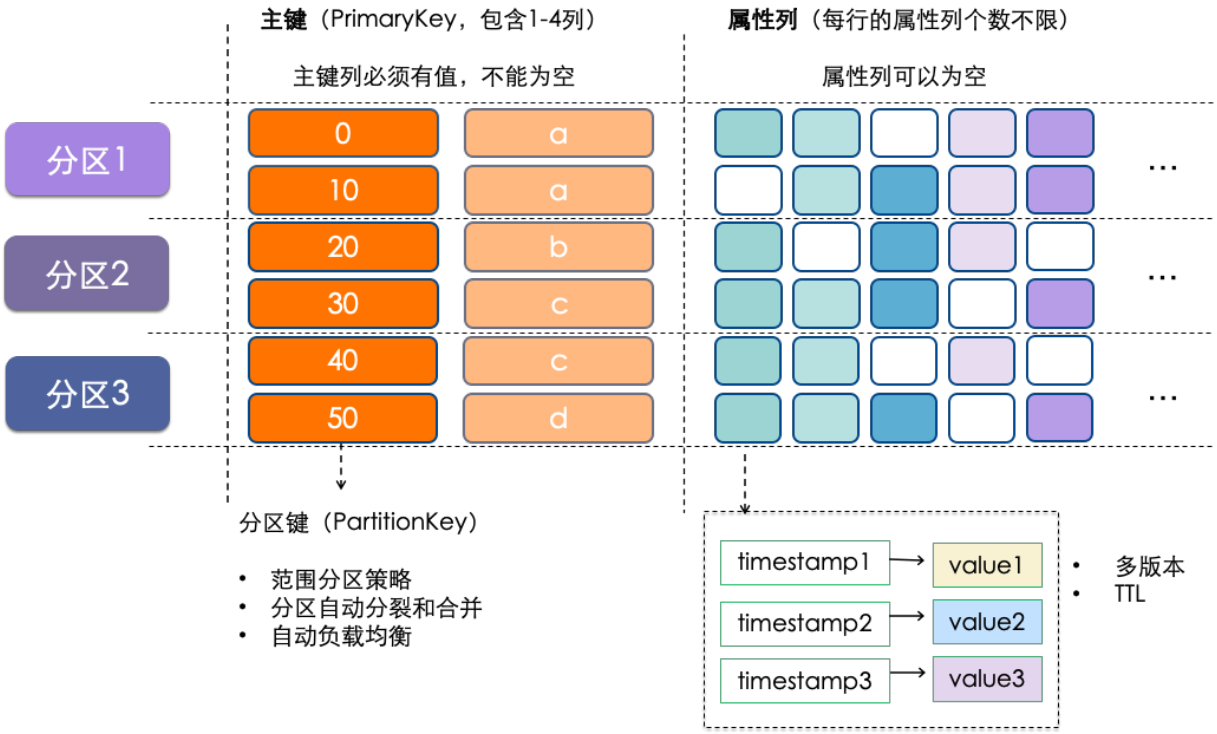
地域	多元索引	通道服务	时序模型	SQL查询	数据湖投递
华东1（杭州）	✓	✓	✓	✓	✓
华东1 金融云	✓	✓	×	×	×
华东2（上海）	✓	✓	✓	✓	✓
华东2 金融云	×	✓	×	×	×
华北1（青岛）	×	×	×	×	×
华北2（北京）	✓	✓	×	✓	✓
华北3（张家口）	✓	✓	×	✓	✓
华北5（呼和浩特）	×	×	×	×	×
华南1（深圳）	✓	✓	✓	✓	×
西南1（成都）	×	×	×	×	×
中国（香港）	✓	✓	×	×	×
新加坡	✓	✓	×	✓	×
澳大利亚（悉尼）	✓	✓	×	×	×
马来西亚（吉隆坡）	×	×	×	×	×
印度尼西亚（雅加达）	✓	✓	×	×	×
日本（东京）	✓	✓	×	×	×

地域	多元索引	通道服务	时序模型	SQL查询	数据湖投递
德国（法兰克福）	✓	✓	✓	✓	×
英国（伦敦）	✓	✓	×	×	×
美国（硅谷）	✓	✓	×	×	×
美国（弗吉尼亚）	✓	✓	×	×	×
印度（孟买）	✓	✓	×	×	×
阿联酋（迪拜）	×	×	×	×	×
菲律宾（马尼拉）	✓	✓	×	×	×

3.宽表模型
3.1. 模型介绍

宽表（WideColumn）模型是表格存储采用的模型之一。通过本文您可以了解宽表模型的构成以及与关系模型的区别。

模型构成



宽表模型如上图所示，由以下几个部分组成。

组成部分	描述
主键（Primary Key）	主键是数据表中每一行的唯一标识，主键由1到4个主键列组成。
分区键（Partition Key）	主键的第一列称为分区键。表格存储按照分区键对数据表的数据进行分区，拥有相同分区键的行会被划分到同一个分区，实现数据访问负载均衡。
属性列（Attribute Column）	一行中除主键列外，其余都是属性列。属性列会对应多个值，不同值对应不同的版本，每行的属性列个数没有限制。
版本（Version）	每一个值对应不同的版本，版本的值是一个时间戳，用于定义数据的生命周期。
数据类型（Data Type）	表格存储支持多种数据类型，包含String、Binary、Double、Integer和Boolean。
生命周期（Time To Live）	每个数据表可定义数据生命周期。例如生命周期配置为一个月，则表格存储会自动清理一个月前写入数据表的数据。

组成部分	描述
最大版本数（Max Versions）	每个数据表可定义每个属性列的数据最多保存的版本个数，用于控制属性列数据的版本个数。当一个属性列数据的版本个数超过Max Versions时，表格存储会异步删除较早版本的数据。

宽表模型与关系模型

宽表模型和关系模型有不同的特点请参见下表。

模型	特点
宽表模型	三维结构（行、列和时间）、Schema-Free、宽行、多版本数据以及生命周期管理。
关系模型	二维（行、列）以及固定的Schema。

3.2. 主键和属性

数据表、行、主键和属性是表格存储的核心组件。数据表是行的集合，而每个行是主键和属性的集合。组成主键的第一个主键列称为分区键。

主键

主键是数据表中每一行的唯一标识，主键由1到4个主键列组成。创建数据表时，必须指定主键的组成、每一个主键列的名称、数据类型以及主键的顺序。

表格存储根据数据表的主键索引数据，数据表中的行按照主键进行升序排序。

分区键

组成主键的第一个主键列又称为分区键。表格存储会根据数据表中每一行分区键的值所属的范围自动将一行数据分配到对应的分区和机器上，以达到负载均衡的目的。具有相同分区键值的行属于同一个数据分区，一个分区可能包含多个分区键值。表格存储服务会自动根据特定的规则对分区进行分裂和合并。

> **说明** 分区键值是最小的分区单位，相同的分区键值的数据无法再做切分。为了防止分区过大无法切分，单个分区键值所有行的大小总和建议不超过10 GB。

属性

属性由多个属性列组成。每行的属性列个数无限制，即每行的属性列可不同。属性列在某一行的值可为空。同一个属性列的值可以有多种数据类型。

属性列有版本特征，属性列的值可以根据需求保留多个版本，用于查询和使用；且属性列的值可以设置生命周期（TTL），详情请参见数据版本和生命周期。

3.3. 数据版本和生命周期

使用数据版本以及数据生命周期（TTL）功能，您可以有效的管理数据，减少数据存储空间，降低存储成本。

版本号

每次更新属性列的值均会为该值生成一个新版本，版本的值即为版本号（时间戳）。

在属性列中写入数据时，如果未设置版本号，则系统会自动生成数据的版本号，您也可以自定义数据的版本号。 版本号的单位为毫秒，在进行TTL比较和有效版本偏差计算时，需要除以1000换算成秒。

- 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为属性列值的版本号。
- 当自定义数据的版本号时，属性列值的版本号需要为64位的毫秒单位时间戳且在有效版本范围内。

使用版本号，您可以实现以下功能：

- 数据生命周期（TTL）

  版本号可以用来定义数据表中属性列值的生命周期。当属性列中数据的保留时长超过设置的TTL后，系统会自动清理对应版本号的数据。

  例如数据的版本号为1468944000000（即2016-07-20 00:00:00 UTC），如果设置数据表的TTL为86400（一天），则该版本号的数据会在2016-07-21 00:00:00 UTC过期，系统会自动删除该版本号的数据。

- 每行数据的版本读取

  读取一行数据时，您可以指定每列最多读取的版本个数或者读取的版本号范围。

最大版本数

最大版本数（Max Versions）表示数据表中的属性列能够保留数据的最大版本个数。当属性列中数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。

创建数据表时，如果未设置最大版本数，则系统会使用默认值1，您也可以自定义属性列的最大版本数。创建数据表后，您可以通过UpdateTable接口修改数据表的最大版本数。

超过最大版本数的数据版本为无效数据，即使系统还未删除数据，用户已无法读取对应数据。

- 当调小最大版本数时，如果数据版本个数超过新设的最大版本数，系统会异步删除较早版本的数据。
- 当调大最大版本数时，如果系统还未删除超过旧的最大版本数的对应版本数据，且对应版本数据在新设的最大版本数范围中，则对应版本的数据可以重新读取。

数据生命周期

数据生命周期是数据表的一个属性，即数据的保存时间，单位为秒。当属性列中数据的保留时间超过设置的TTL时，系统会自动清理超过该属性列的数据。如果一行中所有属性列中数据的保留时间均超过了TTL，则系统会自动清理该行数据。

例如数据表的TTL设置为86400（一天），在2016-07-21 00:00:00 UTC时，数据表中所有版本号小于1468944000000（除以1000换算成秒后即2016-07-20 00:00:00 UTC）的属性列数据均会过期，系统会自动清理对应过期数据。

创建数据表时，如果未设置TTL，则系统会使用默认值-1（表示数据永不过期），您也可以自定义TTL。创建数据表后，您可以通过UpdateTable接口修改TTL。

超过TTL的过期数据为无效数据，即使系统还未删除数据，用户已无法读取对应数据。

- 当调小TTL时，可能会有数据过期，系统会异步删除对应过期数据。
- 当调大TTL时，如果系统还未删除在旧的TTL之外的以前版本的数据，且对应版本数据在新设的TTL中，则对应版本的数据可以重新读取。

有效版本偏差

为了避免写入数据时，自定义的时间戳与当前时间的偏差超过了数据表的TTL，导致写入的数据立即过期的问题，您可以设置有效版本偏差。

有效版本偏差（Max Version Offset）是指定的数据版本号与系统当前时间偏差的允许最大值，单位为秒。有效版本偏差为正整数，可以大于1970-01-01 00:00:00 UTC时间到当前时间的秒数。

为了保证数据写入成功，在写入数据时系统会检查属性列数据的版本号。属性列数据的有效版本范围为**[max{数据写入时间-有效版本偏差,数据写入时间-数据生命周期}，数据写入时间+有效版本偏差)**。只有当属性列数据的版本号（单位为毫秒）除以1000换算为秒后在有效版本范围时，才能成功写入数据。

例如当数据表的有效版本偏差为86400（一天），在2016-07-21 00:00:00 UTC时，只能写入版本号大于1468944000000（换算成秒后即2016-07-20 00:00:00 UTC）并且小于1469116800000（换算成秒后即2016-07-22 00:00:00 UTC）的数据。当某一行的某个属性列数据的版本号为1468943999000（换算成秒后即2016-07-19 23:59:59 UTC）时，该行数据写入失败。

创建数据表时，如果未设置有效版本偏差，则系统会使用默认值86400，您也可以自定义数据有效版本偏差。创建数据表后，您可以通过UpdateTable接口修改有效版本偏差。

3.4. 命名规则和数据类型

通过本文您可以了解表格存储的表名和列名的命名规则，以及主键列和属性列支持的数据类型。

命名规则

表名和列名必须符合以下规则。

规范项	说明
组成	由英文字符（a~z）或（A~Z）、数字（0~9）和下划线（_）组成。
首字母	必须为英文字母（a~z）、（A~Z）或下划线（_）。
大小写	敏感。
长度	1~255个字符之间。
表名是否可重复	• 同一个实例下不能有相同名称的表。 • 不同实例内的表名称可以相同。

数据类型

主键列支持的数据类型包括String、Integer和Binary，属性列支持的数据类型包括String、Integer、Double、Boolean和Binary。

- 主键列支持的数据类型

数据类型	定义	大小限制
String	UTF-8，可为空	长度不超过1 KB
Integer	64 bit，整型，支持自增列	8 Bytes
Binary	二进制数据，可为空	长度不超过1 KB

- 属性列支持的数据类型

数据类型	定义	大小限制
String	UTF-8，可为空	请参见通用限制
Integer	64 bit，整型	8 Bytes
Double	64 bit，Double类型	8 Bytes
Boolean	True/False，布尔类型	1 Byte
Binary	二进制数据，可为空	请参见通用限制

3.5. 表操作

数据表主要用于数据的存储与查询。创建数据表后，您可以根据实际管理数据表，例如获取实例中的所有表名称、查询数据表的配置信息、更新数据表的配置信息等。

> **说明** 本文均以Java SDK为例介绍不同表操作的代码示例。

接口

表操作包括CreateTable、ListTable、UpdateTable、DescribeTable和DeleteTable接口，详细说明请参见下表。

接口	说明
CreateTable	创建一张数据表或者创建一张带有索引表的数据表。
ListTable	获取当前实例下已创建的所有表的表名。
UpdateTable	更新数据表的配置信息。
DescribeTable	查询数据表的配置信息。
DeleteTable	删除一张数据表。

创建数据表（CreateTable）

使用CreateTable接口创建数据表时，需要指定数据表的结构信息和配置信息，高性能实例中的数据表还可以根据需要设置预留读/写吞吐量。创建数据表的同时支持创建一个或者多个索引表。

使用SDK

您可以使用如下语言的SDK创建数据表。

- Java SDK：创建数据表
- Go SDK：创建数据表
- Python SDK：创建数据表
- Node.js SDK：创建数据表
- .NET SDK：创建数据表
- PHP SDK：创建数据表

参数

参数	说明
tableMeta	数据表的结构信息，包括如下内容： • tableName：数据表名称。 • primaryKey：数据表的主键定义。更多信息，请参见主键和属性。 **说明** 属性列不需要定义。表格存储每行的数据列都可以不同，属性列的列名在写入时指定。 ◦ 表格存储可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同，例如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照主键的大小为行排序，具体请参见表格存储数据模型和查询操作。 ◦ 第一列主键作为分区键。分区键相同的数据会存放在同一个分区内，所以相同分区键下最好不要超过10 GB以上数据，否则会导致单分区过大，无法分裂。另外，数据的读/写访问最好在不同的分区键上均匀分布，有利于负载均衡。 • definedColumns：预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。

参数	说明
tableOptions	数据表的配置信息。更多信息，请参见数据版本和生命周期。 配置信息包括如下内容： • timeToLive：数据生命周期，即数据的过期时间。当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。 创建数据表时，如果希望数据永不过期，可以设置数据生命周期为-1；创建数据表后，可以通过UpdateTable接口动态修改数据生命周期。 单位为秒。 **说明** 如果需要使用索引，则数据生命周期必须设置为-1（数据永不过期）。 • maxVersions：最大版本数，即属性列能够保留数据的最大版本个数。当属性列数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。 创建数据表时，可以自定义属性列的最大版本数；创建数据表后，可以通过UpdateTable接口动态修改数据表的最大版本数。 **说明** 如果需要使用索引，则最大版本数必须设置为1。 • maxTimeDeviation：有效版本偏差，即写入数据的时间戳与系统当前时间的偏差允许最大值。只有当写入数据所有列的版本号与写入时时间的差值在数据有效版本偏差范围内，数据才能成功写入。 属性列的有效版本范围为**[数据写入时间-有效版本偏差，数据写入时间+有效版本偏差)**。 创建数据表时，如果未设置有效版本偏差，系统会使用默认值86400；创建数据表后，可以通过UpdateTable接口动态修改有效版本偏差。 单位为秒。
reservedThroughtput	为数据表配置预留读吞吐量或预留写吞吐量。 容量型实例中的数据表的预留读/写吞吐量只能设置为0，不允许预留。 默认值为0，即完全按量计费。 单位为CU。 • 当预留读吞吐量或预留写吞吐量大于0时，表格存储会根据配置为数据表预留相应资源，且数据表创建成功后，将会立即按照预留吞吐量开始计费，超出预留的部分进行按量计费。更多信息，请参见计费概述。 • 当预留读吞吐量或预留写吞吐量设置为0时，表格存储不会为数据表预留相应资源。

参数	说明
indexMetas	索引表的结构信息，每个indexMeta都包括如下内容： • indexName：索引表名称。 • primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 • definedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 • indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ◦ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ◦ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 • indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ◦ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ◦ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。

示例

- 创建数据表（不带索引）

```
private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME, PrimaryKeyTy
pe.STRING)); //为主表添加主键列。
        int timeToLive = -1; //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据生命周期必须设置为-1。
        int maxVersions = 3; //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引表的数据表最大版本数必须设置为1。
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions);
        request.setReservedThroughput(new ReservedThroughput(new CapacityUnit(0, 0))); //
设置预留读写吞吐量，容量型实例中的数据表只能设置为0，高性能实例中的数据表可以设置为非零值。
        client.createTable(request);
}
```

- 创建数据表（带索引且索引类型为全局二级索引）

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta(TABLE_NAME);
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType
.STRING)); //为数据表添加主键列。
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType
.INTEGER)); //为数据表添加主键列。
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColumnT
ype.STRING)); //为数据表添加预定义列。
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColumnT
ype.INTEGER)); //为数据表添加预定义列。
    int timeToLive = -1; //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据生命周期必须设置为-1。
    int maxVersions = 1; //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引表的数据表最大版本数必须设置为1。
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME);
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列。
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); //为索引表添加属性列。
    indexMetas.add(indexMeta);
    CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMet
as); //创建数据表的同时创建索引表。
    client.createTable(request);
}
```

- 创建数据表（带索引且索引类型为本地二级索引）

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta(TABLE_NAME);
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType
.STRING)); //为数据表添加主键列。
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType
.INTEGER)); //为数据表添加主键列。
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColumnT
ype.STRING)); //为数据表添加预定义列。
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColumnT
ype.INTEGER)); //为数据表添加预定义列。
    int timeToLive = -1; //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据生命周期必须设置为-1。
    int maxVersions = 1; //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引表的数据表最大版本数必须设置为1。
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME);
    indexMeta.setIndexType(IT_LOCAL_INDEX);     //设置索引类型为本地二级索引（IT_LOCAL_INDEX）。
    indexMeta.setIndexUpdateMode(IUM_SYNC_INDEX);  //设置索引更新模式为同步更新（IUM_SYNC_INDEX）。当索引类型为本地二级索引时，索引更新模式必须为同步更新。
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1); //为索引表添加主键列。索引表的第一列主键必须与数据表的第一列主键相同。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列。
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); //为索引表添加属性列。
    indexMetas.add(indexMeta);
    CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMet
as); //创建数据表的同时创建索引表。
    client.createTable(request);
}
```

列出表名称（ListTable）

使用ListTable接口获取当前实例下已创建的所有表的表名。

使用SDK

您可以使用如下语言的SDK列出表名称。

- Java SDK：列出表名称
- Go SDK：列出表名称
- Python SDK：列出表名称
- Node.js SDK：列出表名称
- .NET SDK：列出表名称
- PHP SDK：列出表名称

示例

获取实例下所有表的表名。

```
private static void listTable(SyncClient client) {
    ListTableResponse response = client.listTable();
    System.out.println("表的列表如下：");
    for (String tableName : response.getTableNames()) {
        System.out.println(tableName);
    }
}
```

更新表信息（UpdateTable）

使用UpdateTable接口修改配置信息（TableOptions）、预留读吞吐量或者预留写吞吐量（ReservedThroughput）。

使用SDK

您可以使用如下语言的SDK更新表。

- Java SDK：更新表
- Go SDK：更新表
- Python SDK：更新表
- Node.js SDK：更新表
- .NET SDK：更新表
- PHP SDK：更新表

参数

- TableOptions

  TableOptions 包含表的 TTL、MaxVersions 和 MaxTimeDeviation 。

参数	定义	说明
TTL	TimeToLive，数据存活时间	○ 单位：秒。 ○ 如果期望数据永不过期，TTL 可设置为 -1。 ○ 数据是否过期是根据数据的时间戳、当前时间、表的 TTL三者进行判断的。`当前时间 - 数据的时间戳 > 表的 TTL` 时，数据会过期并被清理。 ○ 在使用 TTL 功能时需要注意写入时是否指定了时间戳，以及指定的时间戳是否合理。如需指定时间戳，建议设置MaxTimeDeviation。
MaxTimeDeviation	写入数据的时间戳与系统当前时间的偏差允许最大值	○ 默认情况下系统会为新写入的数据生成一个时间戳，数据自动过期功能需要根据这个时间戳判断数据是否过期。用户也可以指定写入数据的时间戳。如果用户写入的时间戳非常小，与当前时间偏差已经超过了表上设置的 TTL 时间，写入的数据会立即过期。设置 MaxTimeDeviation 可以避免这种情况。 ○ 单位：秒。
MaxVersions	每个属性列保留的最大版本数	如果写入的版本数超过 MaxVersions，服务端只会保留 MaxVersions 中指定的最大的版本。

- ReservedThroughtput

  表的预留读/写吞吐量配置。

  - ReservedThroughput 的调整有时间间隔限制，目前调整间隔为 1 分钟。
  - 设置 ReservedThroughtput 后，表格存储按照您预留读/写吞吐量进行计费。
  - 当 ReservedThroughtput 大于 0 时，表格存储会按照预留量和持续时间进行计费，超出预留的部分进行按量计费。更多信息参见计费，以免产生未期望的费用。
  - 默认值为 0，即完全按量计费。
  - 容量型实例的预留读/写吞吐量只能设置为 0，不允许预留。

示例

更新表的TTL和最大版本数。

```
private static void updateTable(SyncClient client) {
    int timeToLive = -1;
    int maxVersions = 5; // 将最大版本数更新为5。
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    UpdateTableRequest request = new UpdateTableRequest(TABLE_NAME);
    request.setTableOptionsForUpdate(tableOptions);
    client.updateTable(request);
}
```

查询表描述信息（DescribeTable）

使用DescribeTable接口可以查询指定表的结构、预留读/写吞吐量详情等信息。

使用SDK

您可以使用如下语言的SDK查询表描述信息。

- Java SDK：查询表描述信息
- Go SDK：查询表描述信息
- Python SDK：查询表描述信息
- Node.js SDK：查询表描述信息
- .NET SDK：查询表描述信息
- PHP SDK：查询表描述信息

参数

参数	说明
tableName	表名。

示例

```
private static void describeTable(SyncClient client) {
    DescribeTableRequest request = new DescribeTableRequest(TABLE_NAME);
    DescribeTableResponse response = client.describeTable(request);
    TableMeta tableMeta = response.getTableMeta();
    System.out.println("表的名称: " + tableMeta.getTableName());
    System.out.println("表的主键: ");
    for (PrimaryKeySchema primaryKeySchema : tableMeta.getPrimaryKeyList()) {
        System.out.println(primaryKeySchema);
    }
    TableOptions tableOptions = response.getTableOptions();
    System.out.println("表的TTL:" + tableOptions.getTimeToLive());
    System.out.println("表的MaxVersions:" + tableOptions.getMaxVersions());
    ReservedThroughputDetails reservedThroughputDetails = response.getReservedThroughputDet
ails();
    System.out.println("表的预留读吞吐量: "
            + reservedThroughputDetails.getCapacityUnit().getReadCapacityUnit());
    System.out.println("表的预留写吞吐量: "
            + reservedThroughputDetails.getCapacityUnit().getWriteCapacityUnit());
}
```

删除数据表（DeleteTable）

使用DeleteTable接口删除当前实例下指定数据表。

使用SDK

您可以使用如下语言的SDK删除数据表。

- Java SDK：删除数据表
- Go SDK：删除数据表
- Python SDK：删除数据表
- Node.js SDK：删除数据表
- .NET SDK：删除数据表
- PHP SDK：删除数据表

参数

参数	说明
tableName	数据表名称。

示例

```
private static void deleteTable(SyncClient client) {
    DeleteTableRequest request = new DeleteTableRequest(TABLE_NAME);
    client.deleteTable(request);
}
```

其他操作方式

您还可以使用控制台或者命令行工具进行表操作。

- 使用控制台
- 使用命令行工具

3.6. 基础数据操作

3.6.1. 单行数据操作

表格存储提供了PutRow、GetRow、UpdateRow和DeleteRow等单行操作的接口。

> **说明** 组成表的基本单位为行，行由主键和属性组成。其中主键是必须的，且每一行的主键列的名称和类型相同；属性不是必须的，且每一行的属性可以不同。更多信息，请参见模型介绍。

接口

单行数据操作包括PutRow、GetRow、UpdateRow和DeleteRow接口，详细说明请参见下表。

接口	说明
GetRow	读取一行数据。
PutRow	新写入一行数据。如果该行已存在，则先删除旧行数据，再写入新行数据。
UpdateRow	更新一行数据。可以增加、删除一行中的属性列，或者更新已存在的属性列的值。 如果更新的行不存在，则新增一行数据。但是当UpdateRow请求中只包含删除指定的列且该行不存在时，则该请求不会新增一行数据。
DeleteRow	删除一行数据。 如果删除的行不存在，则不会发生任何变化。

使用

您可以使用如下语言的SDK实现单行数据操作。

- Java SDK：单行数据操作
- Go SDK：单行数据操作
- Python SDK：单行数据操作
- Node.js SDK：单行数据操作
- .NET SDK：单行数据操作
- PHP SDK：单行数据操作

插入一行数据（PutRow）

PutRow接口用于新写入一行数据。如果该行已存在，则先删除原行数据（原行的所有列以及所有版本的数据），再写入新行数据。

CU消耗说明

PutRow操作消耗的读CU和写CU说明如下：

- 消耗的写CU为修改的行主键数据大小与属性列数据大小之和除以4 KB向上取整。

- 如果指定条件检查不为IGNORE，则消耗行主键数据大小除以4 KB向上取整的读CU。
- 如果操作不满足指定的行存在性检查条件，则操作失败并消耗1单位写CU和1单位读CU。

操作结果说明

不同的操作结果返回的结果不同。

- 如果操作成功，表格存储会返回操作消耗的服务能力单元（CU）。

  > **说明** 写入操作会根据指定的condition情况消耗一定的读CU。

  通过在单行写入请求中设置condition字段可以指定写入操作执行时，是否对行的存在性进行检查。condition有如下三种类型：

condition	说明
IGNORE	不做任何存在性检查。
EXPECT_EXIST	期望行存在。 ○ 如果该行存在，则操作成功。 ○ 如果该行不存在，则操作失败。
EXPECT_NOT_EXIST	期望行不存在。 ○ 如果该行不存在，则操作成功。 ○ 如果该行存在，则操作失败。 **说明** condition为EXPECT_NOT_EXIST的DeleteRow、UpdateRow操作是无意义的，即删除一个不存在的行是无意义的。如果需要更新不存在的行可以使用PutRow操作。

- 如果操作发生错误，例如参数检查失败、单行数据量过多、行存在性检查失败等，表格存储会返回错误码。

参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** • 设置的主键个数和类型必须和数据表的主键个数和类型一致。 • 当主键为自增列时，只需将自增列的值设置为占位符。更多信息，请参见主键列自增。

参数	说明
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件。更多信息，请参见条件更新。 说明 • RowExistenceExpectation.IGNORE表示无论此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。 • RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。 • RowExistenceExpectation.EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。
column	行的属性列。 • 每一项的顺序是属性名、属性值ColumnValue、属性类型ColumnType（可选）、时间戳（可选）。 • ColumnType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、BOOLEAN、DOUBLE五种，分别用ColumnType.INTEGER、ColumnType.STRING、ColumnType.BINARY、ColumnType.BOOLEAN、ColumnType.DOUBLE表示，其中BINARY不可省略，其他类型都可以省略。 • 时间戳即数据的版本号。更多信息，请参见数据版本和生命周期。 数据的版本号可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。 ◦ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ◦ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。

示例

- 示例1

  写入10列属性列，每列写入1个版本，由系统自动生成数据的版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    //加入一些属性列。
    for (int i = 0; i < 10; i++) {
        rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

- 示例2

  写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    //加入一些属性列。
    long ts = System.currentTimeMillis();
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 3; j++) {
            rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(j), ts + j)
);
        }
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

- 示例3

  期望原行不存在时，写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    //设置条件更新，行条件检查为期望原行不存在。
    rowPutChange.setCondition(new Condition(RowExistenceExpectation.EXPECT_NOT_EXIST));
    //加入一些属性列。
    long ts = System.currentTimeMillis();
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 3; j++) {
            rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(j), ts + j)
);
        }
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

- 示例4

  期望原行存在且Col0列的值大于100时，写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    //设置条件更新，期望原行存在且Col0列的值大于100时写入数据。
    Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
    condition.setColumnCondition(new SingleColumnValueCondition("Col0",
            SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(100)));
    rowPutChange.setCondition(condition);
    //加入一些属性列。
    long ts = System.currentTimeMillis();
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 3; j++) {
            rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(j), ts + j)
);
        }
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

读取一行数据（GetRow）

GetRow接口用于读取一行数据。

CU消耗说明

GetRow操作不消耗写CU，消耗的读CU为读取的行主键的数据大小与实际读取的属性列数据大小之和，按4 KB向上取整。如果操作指定的行不存在，则消耗1单位读CU。

操作结果说明

读取的结果可能有如下两种：

- 如果该行存在，则返回该行的各主键列以及属性列。
- 如果该行不存在，则返回中不包含行，并且不会报错。

参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。

参数	说明
columnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** • 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columnsToGet参数限制。如果将col0和col1加入到columnsToGet中，则只返回col0和col1列的值。 • 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
maxVersions	最多读取的版本数。 **说明** maxVersions与timeRange必须至少设置一个。 • 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 • 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 • 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。
timeRange	读取版本号范围或特定版本号的数据。更多信息，请参见TimeRange。 **说明** maxVersions与timeRange必须至少设置一个。 • 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 • 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 • 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 • 查询一个范围的数据，则需要设置start和end。start和end分别表示起始时间戳和结束时间戳，范围为前闭后开区间。 • 如果查询特定版本号的数据，则需要设置timestamp。timestamp表示特定的时间戳。 timestamp和[start, end)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为Long.MAX_VALUE。
filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行。更多信息，请参见过滤器。 **说明** 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。

示例

- 示例1

  读取一行，设置读取最新版本的数据和读取的列。

```
private static void getRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //读取一行数据，设置数据表名称。
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey);
    //设置读取最新版本。
    criteria.setMaxVersions(1);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("读取完毕，结果为： ");
    System.out.println(row);
    //设置读取某些列。
    criteria.addColumnsToGet("Col0");
    getRowResponse = client.getRow(new GetRowRequest(criteria));
    row = getRowResponse.getRow();
    System.out.println("读取完毕，结果为：");
    System.out.println(row);
}
```

- 示例2

  在读取一行数据时使用过滤器。

```
private static void getRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //读取一行数据，设置数据表名称。
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey);
    //设置读取最新版本。
    criteria.setMaxVersions(1);
    //设置过滤器，当Col0列的值为0时，返回该行。
    SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
            SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
    //如果Col0列不存在，则不返回该行。
    singleColumnValueFilter.setPassIfMissing(false);
    criteria.setFilter(singleColumnValueFilter);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("读取完毕，结果为：");
    System.out.println(row);
}
```

- 示例3

  读取一行中Col1列的数据，并对该列的数据执行正则过滤。

```
private static void getRow(SyncClient client, String pkValue) {
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(tableName);
    //构造主键。
    PrimaryKey primaryKey = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pkValue))
        .build();
    criteria.setPrimaryKey(primaryKey);
    // 设置读取最新版本。
    criteria.setMaxVersions(1);
    // 设置过滤器，当cast<int>(regex(Col1)) > 100时，返回该行。
    RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
    SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
        regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromL
ong(100));
    criteria.setFilter(filter);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("读取完毕，结果为：");
    System.out.println(row);
}
```

更新一行数据（UpdateRow）

UpdateRow接口用于更新一行数据，可以增加和删除一行中的属性列，删除属性列指定版本的数据，或者更新已存在的属性列的值。如果更新的行不存在，则新增一行数据。

> **说明** 当UpdateRow请求中只包含删除指定的列且该行不存在时，则该请求不会新增一行数据。

CU消耗说明

UpdateRow操作消耗的读CU和写CU说明如下：

- 消耗的写CU为修改的行主键数据大小与属性列数据大小之和除以4 KB向上取整。

  操作中包含的需要删除的属性列，只有属性列的列名计入属性列数据大小。
- 如果指定条件检查不为IGNORE，则消耗行主键数据大小除以4 KB向上取整的读CU。
- 如果操作不满足指定的行存在性检查条件，则操作失败并消耗1单位写CU和1单位读CU。

参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件。更多信息，请参见条件更新。

参数	说明
column	更新的属性列。 • 增加或更新数据时，需要设置属性名、属性值、属性类型（可选）、时间戳（可选）。 时间戳即数据的版本号，可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。更多信息，请参见数据版本和生命周期。 ○ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ○ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。 • 删除属性列特定版本的数据时，只需要设置属性名和时间戳。 时间戳是64位整数，单位为毫秒，表示某个特定版本的数据。 • 删除属性列时，只需要设置属性名。 说明 删除一行的全部属性列不等同于删除该行，如果需要删除该行，请使用DeleteRow操作。

示例

- 示例1

  更新一些列，删除某列的某一版本数据，删除某列。

```
private static void updateRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
    //更新一些列。
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //删除某列的某一版本。
    rowUpdateChange.deleteColumn("Col10", 1465373223000L);
    //删除某一列。
    rowUpdateChange.deleteColumns("Col11");
    client.updateRow(new UpdateRowRequest(rowUpdateChange));
}
```

- 示例2

  设置更新的条件。

```
private static void updateRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
    //设置条件更新，期望原行存在且Col0列的值大于100时更新数据。
    Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
    condition.setColumnCondition(new SingleColumnValueCondition("Col0",
            SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(100)));
    rowUpdateChange.setCondition(condition);
    //更新一些列。
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //删除某列的某一版本。
    rowUpdateChange.deleteColumn("Col10", 1465373223000L);
    //删除某一列。
    rowUpdateChange.deleteColumns("Col11");
    client.updateRow(new UpdateRowRequest(rowUpdateChange));
}
```

删除一行数据（DeleteRow）

DeleteRow接口用于删除一行数据。如果删除的行不存在，则不会发生任何变化。

CU消耗说明

DeleteRow操作消耗的读CU和写CU说明如下：

- 消耗的写CU为删除的行主键数据大小除以4 KB向上取整。
- 如果指定条件检查不为IGNORE，则消耗行主键数据大小除以4 KB向上取整的读CU。
- 如果操作不满足指定的行存在性检查条件，则操作失败并消耗1单位写CU。

参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 说明 设置的主键个数和类型必须和数据表的主键个数和类型一致。
condition	支持使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件。更多信息，请参见条件更新。

示例

- 示例1

  删除一行数据。

```
private static void deleteRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, primaryKey);
    client.deleteRow(new DeleteRowRequest(rowDeleteChange));
}
```

- 示例2

  设置删除条件。

```
private static void deleteRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, primaryKey);
    //设置条件更新，期望原行存在且Col0列的值大于100时删除该行。
    Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
    condition.setColumnCondition(new SingleColumnValueCondition("Col0",
            SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(100)));
    rowDeleteChange.setCondition(condition);
    client.deleteRow(new DeleteRowRequest(rowDeleteChange));
}
```

3.6.2. 多行数据操作

表格存储提供了BatchWriteRow、BatchGetRow、GetRange等多行数据操作的接口。

> **说明** 组成表的基本单位为行，行由主键和属性组成。其中主键是必须的，且每一行的主键列的名称和类型相同；属性不是必须的，且每一行的属性可以不同。更多信息，请参见模型介绍。

接口

多行数据操作包括BatchWriteRow、BatchGetRow和GetRange接口，详细说明请参见下表。

接口	说明
BatchGetRow	批量读取一个或多个表中多行数据。
BatchWriteRow	批量插入、批量更新或者批量删除一个或多个表中的多行数据。

接口	说明
GetRange	根据主键范围查询表中数据。

使用

您可以使用如下语言的SDK实现多行数据操作。

- Java SDK：多行数据操作
- Go SDK：多行数据操作
- Python SDK：多行数据操作
- Node.js SDK：多行数据操作
- .NET SDK：多行数据操作
- PHP SDK：多行数据操作

批量写（BatchWriteRow）

BatchWriteRow接口用于在一次请求中进行批量的写入操作，也支持一次对多张表进行写入。BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，构造子操作的过程与使用PutRow接口、UpdateRow接口和DeleteRow接口时相同，也支持设置更新条件。

BatchWriteRow操作的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量写入可能存在部分行失败的情况，失败行的Index及错误信息在返回的BatchWriteRowResponse中，但并不抛出异常。因此调用BatchWriteRow接口时，需要检查返回值，可通过BatchWriteRowResponse的isAllSucceed方法判断是否全部成功；如果不检查返回值，则可能会忽略掉部分操作的失败。

当服务端检查到某些操作出现参数错误时，BatchWriteRow接口可能会抛出参数错误的异常，此时该请求中所有的操作都未执行。

参数

更多信息，请参见单行数据操作。

示例

一次BatchWriteRow请求，包含2个PutRow操作、1个UpdateRow操作和1个DeleteRow操作。

```
private static void batchWriteRow(SyncClient client) {
    BatchWriteRowRequest batchWriteRowRequest = new BatchWriteRowRequest();
    //构造rowPutChange1。
    PrimaryKeyBuilder pk1Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk1Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk1"));
    RowPutChange rowPutChange1 = new RowPutChange(TABLE_NAME, pk1Builder.build());
    //添加一些列。
    for (int i = 0; i < 10; i++) {
        rowPutChange1.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //添加到batch操作中。
    batchWriteRowRequest.addRowChange(rowPutChange1);
    //构造rowPutChange2。
    PrimaryKeyBuilder pk2Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk2Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk2"));
    RowPutChange rowPutChange2 = new RowPutChange(TABLE_NAME, pk2Builder.build());
    //添加一些列。
```

```
    //添加一些列。
    for (int i = 0; i < 10; i++) {
        rowPutChange2.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //添加到batch操作中。
    batchWriteRowRequest.addRowChange(rowPutChange2);
    //构造rowUpdateChange。
    PrimaryKeyBuilder pk3Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk3Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk3"));
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, pk3Builder.build());
    //添加一些列。
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //删除一列。
    rowUpdateChange.deleteColumns("Col10");
    //添加到batch操作中。
    batchWriteRowRequest.addRowChange(rowUpdateChange);
    //构造rowDeleteChange。
    PrimaryKeyBuilder pk4Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk4Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk4"));
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, pk4Builder.build());
    //添加到batch操作中。
    batchWriteRowRequest.addRowChange(rowDeleteChange);
    BatchWriteRowResponse response = client.batchWriteRow(batchWriteRowRequest);
    System.out.println("是否全部成功：" + response.isAllSucceed());
    if (!response.isAllSucceed()) {
        for (BatchWriteRowResponse.RowResult rowResult : response.getFailedRows()) {
            System.out.println("失败的行：" + batchWriteRowRequest.getRowChange(rowResult.ge
tTableName(), rowResult.getIndex()).getPrimaryKey());
            System.out.println("失败原因：" + rowResult.getError());
        }
        /**
         * 可以通过createRequestForRetry方法再构造一个请求对失败的行进行重试。此处只给出构造重试请求
的部分。
         * 推荐的重试方法是使用SDK的自定义重试策略功能，支持对batch操作的部分行错误进行重试。设置重试
策略后，调用接口处无需增加重试代码。
         */
        BatchWriteRowRequest retryRequest = batchWriteRowRequest.createRequestForRetry(resp
onse.getFailedRows());
    }
}
```

批量读（BatchGetRow）

BatchGetRow接口用于一次请求读取多行数据，也支持一次对多张表进行读取。BatchGetRow操作由多个GetRow子操作组成。构造子操作的过程与使用GetRow接口时相同。

批量读取的所有行采用相同的参数条件，例如ColumnsToGet=[colA]，则要读取的所有行都只读取colA列。

BatchGetRow操作的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量读取可能存在部分行失败的情况，失败行的错误信息在返回的BatchGetRowResponse中，但并不抛出异常。因此调用BatchGetRow接口时，需要检查返回值，可通过BatchGetRowResponse的isAllSucceed方法判断是否所有行都获取成功；通过BatchGetRowResponse的getFailedRows方法获取失败行的信息。

参数

更多信息，请参见单行数据操作。

示例

读取10行，设置版本条件、要读取的列、过滤器等。

```
private static void batchGetRow(SyncClient client) {
    MultiRowQueryCriteria multiRowQueryCriteria = new MultiRowQueryCriteria(TABLE_NAME);
    //加入10个要读取的行。
    for (int i = 0; i < 10; i++) {
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
        primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(
"pk" + i));
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        multiRowQueryCriteria.addRow(primaryKey);
    }
    //添加条件。
    multiRowQueryCriteria.setMaxVersions(1);
    multiRowQueryCriteria.addColumnsToGet("Col0");
    multiRowQueryCriteria.addColumnsToGet("Col1");
    SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
            SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
    singleColumnValueFilter.setPassIfMissing(false);
    multiRowQueryCriteria.setFilter(singleColumnValueFilter);
    BatchGetRowRequest batchGetRowRequest = new BatchGetRowRequest();
    //BatchGetRow支持读取多个表的数据，一个multiRowQueryCriteria对应一个表的查询条件，可以添加多个m
ultiRowQueryCriteria。
    batchGetRowRequest.addMultiRowQueryCriteria(multiRowQueryCriteria);
    BatchGetRowResponse batchGetRowResponse = client.batchGetRow(batchGetRowRequest);
    System.out.println("是否全部成功：" + batchGetRowResponse.isAllSucceed());
    if (!batchGetRowResponse.isAllSucceed()) {
        for (BatchGetRowResponse.RowResult rowResult : batchGetRowResponse.getFailedRows())
{
            System.out.println("失败的行：" + batchGetRowRequest.getPrimaryKey(rowResult.get
TableName(), rowResult.getIndex()));
            System.out.println("失败原因：" + rowResult.getError());
        }
        /**
         * 可以通过createRequestForRetry方法再构造一个请求对失败的行进行重试。此处只给出构造重试请求
的部分。
         * 推荐的重试方法是使用SDK的自定义重试策略功能，支持对batch操作的部分行错误进行重试。设置重试
策略后，调用接口处无需增加重试代码。
         */
        BatchGetRowRequest retryRequest = batchGetRowRequest.createRequestForRetry(batchGet
RowResponse.getFailedRows());
    }
}
```

范围读（GetRange）

GetRange接口用于根据主键范围读取表中数据，主键范围为左闭右开的区间。

GetRange操作支持按照确定的主键范围进行正序读取和逆序读取，可以设置要读取的行数。如果范围较大，已扫描的行数或者数据量超过一定限制，会停止扫描，并返回已获取的行和下一个主键信息。您可以根据返回的下一个主键信息，继续发起请求，获取范围内剩余的行。

> **说明** 表格存储表中的行都是按照主键排序的，而主键是按照第一主键列、第二主键列等依次顺序组成的，所以不能理解为表格存储会按照某列主键排序，这是常见的误区。

注意事项

GetRange操作遵循最左匹配原则，读取数据时，依次比较第一主键列到第四主键列。例如数据表的主键包括PK1、PK2、PK3三个主键列，读取数据时，优先比较PK1是否在开始主键与结束主键的范围内，如果PK1在设置的主键范围内，则不会再比较其他的主键，返回在PK1主键范围内的数据；如果PK1不在设置的主键范围内，则继续比较PK2是否在开始主键与结束主键的范围内，以此类推。关于范围查询原理的更多信息，请参见GetRange范围查询详解。

GetRange操作可能在如下情况停止执行并返回数据。

- 扫描的行数据大小之和达到4 MB。
- 扫描的行数等于5000。
- 返回的行数等于最大返回行数。
- 当前剩余的预留读吞吐量已全部使用，余量不足以读取下一条数据。

CU消耗说明

GetRange操作消耗的读CU为从区间起始点到下一条未读数据的起始点，所有行主键数据大小与实际扫描的所有属性列数据大小之和按4 KB向上取整计算消耗的读CU。例如扫描范围中包含10行，每行主键数据大小与实际扫描的所有属性列数据之和为330 Byte，则消耗的读CU为1（数据总和3.3 KB，除以4 KB向上取整为1）。

参数

参数	说明
tableName	数据表名称。
direction	读取方向。 • 如果值为正序（FORWARD），则起始主键必须小于结束主键，返回的行按照主键由小到大的顺序进行排列。 • 如果值为逆序（BACKWARD），则起始主键必须大于结束主键，返回的行按照主键由大到小的顺序进行排列。 例如同一表中有两个主键A和B，A<B。如正序读取[A, B)，则按从A至B的顺序返回主键大于等于A、小于B的行；逆序读取[B, A)，则按从B至A的顺序返回大于A、小于等于B的数据。

参数	说明
inclusiveStartPrimaryKey exclusiveEndPrimaryKey	本次范围读取的起始主键和结束主键，起始主键和结束主键需要是有效的主键或者是由INF_MIN和INF_MAX类型组成的虚拟点，虚拟点的列数必须与主键相同。 其中INF_MIN表示无限小，任何类型的值都比它大；INF_MAX表示无限大，任何类型的值都比它小。 • inclusiveStartPrimaryKey表示起始主键，如果该行存在，则返回结果中一定会包含此行。 • exclusiveEndPrimaryKey表示结束主键，无论该行是否存在，返回结果中都不会包含此行。 数据表中的行按主键从小到大排序，读取范围是一个左闭右开的区间，正序读取时，返回的是大于等于起始主键且小于结束主键的所有的行。
limit	数据的最大返回行数，此值必须大于 0。 表格存储按照正序或者逆序返回指定的最大返回行数后即结束该操作的执行，即使该区间内仍有未返回的数据。此时可以通过返回结果中的nextStartPrimaryKey记录本次读取到的位置，用于下一次读取。
columnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** • 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columnsToGet参数限制。如果将col0和col1加入到columnsToGet中，则只返回col0和col1列的值。 • 如果某行数据的主键属于读取范围，但是该行数据不包含指定返回的列，那么返回结果中不包含该行数据。 • 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
maxVersions	最多读取的版本数。 **说明** maxVersions与timeRange必须至少设置一个。 • 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 • 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 • 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。

参数	说明
timeRange	读取版本号范围或特定版本号的数据。更多信息，请参见TimeRange。 **说明** maxVersions与timeRange必须至少设置一个。 • 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 • 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 • 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 • 如果查询一个范围的数据，则需要设置start和end。start和end分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[start, end)。 • 如果查询特定版本号的数据，则需要设置timestamp。timestamp表示特定的时间戳。 timestamp和[start, end)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为Long.MAX_VALUE。
filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行。更多信息，请参见过滤器。 **说明** 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
nextStartPrimaryKey	根据返回结果中的nextStartPrimaryKey判断数据是否全部读取。 • 当返回结果中nextStartPrimaryKey不为空时，可以使用此返回值作为下一次GetRange操作的起始点继续读取数据。 • 当返回结果中nextStartPrimaryKey为空，表示读取范围内的数据全部返回。

示例

- 示例1

  按照确定范围进行正序读取，判断nextStartPrimaryKey是否为空，读取完范围内的全部数据。

```
private static void getRange(SyncClient client, String startPkValue, String endPkValue) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    //设置起始主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(st
artPkValue));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    //设置结束主键。
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(en
dPkValue));
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("GetRange的结果为：");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果NextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- 示例2

  按照第一个主键列确定范围、第二主键列从最小值（INF_MIN）到最大值（INF_MAX）进行正序读取，判断nextStartPrimaryKey是否为null，读取完范围内的全部数据。

```
private static void getRange(SyncClient client, String startPkValue, String endPkValue) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    //设置起始主键，以两个主键为例。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1, PrimaryKeyValue.fromString(s
tartPkValue));//确定值。
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2, PrimaryKeyValue.INF_MIN);//
最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    //设置结束主键。
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1, PrimaryKeyValue.fromString(e
ndPkValue));//确定值。
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2, PrimaryKeyValue.INF_MAX);//
最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("GetRange的结果为：");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- 示例3

  读取主键范围为["pk:2020-01-01.log", "pk:2021-01-01.log")时Col1列的数据，并对该列的数据执行正则过滤。

```
private static void getRange(SyncClient client) {
    RangeRowQueryCriteria criteria = new RangeRowQueryCriteria(TABLE_NAME);
    // 设置主键范围为["pk:2020-01-01.log", "pk:2021-01-01.log")，读取范围为左闭右开的区间。
    PrimaryKey pk0 = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("pk", PrimaryKeyValue.fromString("2020-01-01.log"))
        .build();
    PrimaryKey pk1 = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("pk", PrimaryKeyValue.fromString("2021-01-01.log"))
        .build();
    criteria.setInclusiveStartPrimaryKey(pk0);
    criteria.setExclusiveEndPrimaryKey(pk1);
    // 设置读取最新版本。
    criteria.setMaxVersions(1);
    // 设置过滤器，当cast<int>(regex(Col1)) > 100时，返回该行。
    RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
    SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
        regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN,ColumnValue.fromLo
ng(100));
    criteria.setFilter(filter);
    while (true) {
        GetRangeResponse resp = client.getRange(new GetRangeRequest(criteria));
        for (Row row : resp.getRows()) {
            // do something
            System.out.println(row);
        }
        if (resp.getNextStartPrimaryKey() != null) {
            criteria.setInclusiveStartPrimaryKey(resp.getNextStartPrimaryKey())
        }
    }
}
```

3.7. 主键列自增

设置非分区键的主键列为自增列后，在写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。

特点

主键列自增具有如下特点：

- 自增列的值在分区键级别唯一且严格递增，但不保证连续。
- 自增列的数据类型为64位的有符号长整型。
- 自增列是数据表级别的，同一个实例下可以有自增列或者非自增列的数据表。

> **说明** 无论是否使用主键列自增功能，不影响条件更新的规则，条件更新的规则请参见条件更新。

场景

适用于系统设计中需要使用主键列自增功能的场景，例如电商网站的商品ID、大型网站的用户ID、论坛帖子的ID、聊天工具的消息ID等。

应用案例请参见Tablestore主键列自增功能在IM系统中的应用。

限制

主键列自增有如下限制：

- 每张数据表最多只能设置一个主键列为自增列，主键中的分区键不能设置为自增列。
- 只能在创建数据表时指定自增列，对于已存在的数据表不能创建自增列。
- 只有整型的主键列才能设置为自增列，系统自动生成的自增列值为64位的有符号长整型。
- 属性列不能设置为自增列。

接口

主键列自增的相关接口说明请参见下表。

接口	说明
CreateTable	创建数据表时，请设置非分区键的主键列为自增列，否则无法使用主键列自增功能。
UpdateTable	数据表创建后，不能通过UpdateTable修改数据表的主键列为自增列。
PutRow	写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。 通过设置ReturnType为RT_PK，可以获取完整的主键值，完整的主键值可以用于GetRow查询数据。
UpdateRow	
BatchWriteRow	
GetRow	使用GetRow时需要完整的主键值，通过设置PutRow、UpdateRow或者BatchWriteRow中的ReturnType为RT_PK可以获取完整的主键值。
BatchGetRow	

示例

主键自增列功能主要涉及创建表（CreateTable）和写数据（PutRow、UpdateRow和BatchWriteRow）两类接口。

1. 创建表

   创建表时，只需将自增的主键属性设置为AUTO_INCREMENT。

```
private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta("table_name");
        //第一列为分区键。
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("PK_1", PrimaryKeyType.STRIN
G));
        //第二列为自增列，类型为INTEGER，属性为AUTO_INCREMENT。
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("PK_2", PrimaryKeyType.INTEG
ER, PrimaryKeyOption.AUTO_INCREMENT));
        int timeToLive = -1;  //数据永不过期。
        int maxVersions = 1;  //只保存一个数据版本。
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions);
        client.createTable(request);
    }
```

2. 写数据

写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符AUTO_INCREMENT。

```
    private static void putRow(SyncClient client, String receive_id) {
        //构造主键。
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder
();
        //第一列的值为md5(receive_id)前4位。
        primaryKeyBuilder.addPrimaryKeyColumn(“PK_1”, PrimaryKeyValue.fromString("Hangz
hou");
        //第二列是主键自增列，此处无需填入具体值，只需要一个占位符AUTO_INCREMENT，表格存储会自动
生成此值。
        primaryKeyBuilder.addPrimaryKeyColumn("PK_2", PrimaryKeyValue.AUTO_INCREMENT);
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        RowPutChange rowPutChange = new RowPutChange("table_name", primaryKey);
        //此处设置返回类型为RT_PK，即在返回结果中包含PK列的值。如果不设置ReturnType，默认不返回
。
        rowPutChange.setReturnType(ReturnType.RT_PK);
        //加入属性列。
        rowPutChange.addColumn(new Column("content", ColumnValue.fromString(content)));
        //写入数据到表格存储。
        PutRowResponse response = client.putRow(new PutRowRequest(rowPutChange));
        //打印返回的PK列。
        Row returnRow = response.getRow();
        if (returnRow != null) {
            System.out.println("PrimaryKey:" + returnRow.getPrimaryKey().toString());
        }
        //打印消耗的CU。
        CapacityUnit  cu = response.getConsumedCapacity().getCapacityUnit();
        System.out.println("Read CapacityUnit:" + cu.getReadCapacityUnit());
        System.out.println("Write CapacityUnit:" + cu.getWriteCapacityUnit());
    }
```

使用

您可以使用如下语言的SDK实现主键列自增功能。

- Java SDK：主键列自增
- Go SDK：主键列自增
- Python SDK：主键列自增
- Node.js SDK：主键列自增
- .NET SDK：主键列自增
- PHP SDK：主键列自增

计费

使用主键列自增功能不影响现有计费规则，返回的主键列数据不会额外消耗读CU。

3.8. 条件更新

只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。

场景

适用于对高并发的应用进行更新的场景。

在高并发场景下，old_value可能被其他客户端更新，如果使用条件更新，则只有在value等于old_value时，才更新value为new_value。

> **说明** 在网页计数和游戏等高并发场景下，使用条件更新后，更新数据可能会失败，此时需要一定次数的重试。

```
//获取当前值。
old_value = Read();
//对当前值进行运算，例如加1操作。
new_value = func(old_value);
//使用最新值进行更新。
Update(new_value);
```

接口

条件更新支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），支持最多10个条件的组合。适用于PutRow、UpdateRow、DeleteRow和BatchWriteRow接口。

条件更新可以实现乐观锁功能，即在更新某行时，先获取某列的值，假设为列A，值为1，然后设置条件**列A = 1**，更新行使**列A = 2**。如果更新失败，表示有其他客户端已成功更新该行。

列判断条件包括列条件和行存在性条件。

列判断条件	说明
列条件	目前支持SingleColumnValueCondition和CompositeColumnValueCondition，是基于某一列或者某些列的列值进行条件判断，与过滤器Filter中的条件类似。
行存在性条件	对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。 行存在性条件包括如下类型。 • IGNORE：忽略 • EXPECT_EXIST：期望存在 • EXPECT_NOT_EXIST：期望不存在 通过不同接口操作数据表的数据时的行存在性条件更新规则的更多信息，请参见行存在性条件更新规则。

行存在性条件更新规则

> **说明** BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，所以通过BatchWriteRow接口操作数据表中的数据时，请根据操作类型查看对应接口的更新规则。

接口	IGNORE	EXPECT_EXIST	EXPECT_NOT_EXIST
PutRow（已存在行）	成功	成功	失败
PutRow（不存在行）	成功	失败	成功

接口	IGNORE	EXPECT_EXIST	EXPECT_NOT_EXIST
UpdateRow（已存在行）	成功	成功	失败
UpdateRow（不存在行）	成功	失败	成功
DeleteRow（已存在行）	成功	成功	失败
DeleteRow（不存在行）	成功	失败	成功

使用

您可以使用如下语言的SDK实现条件更新功能。

- Java SDK：条件更新
- Go SDK：条件更新
- Python SDK：条件更新
- Node.js SDK：条件更新
- .NET SDK：条件更新
- PHP SDK：条件更新

示例

使用列判断条件和乐观锁的示例代码如下：

- 构造SingleColumnValueCondition。

```
 //设置条件为Col0==0。
 SingleColumnValueCondition singleColumnValueCondition = new SingleColumnValueCondition("
Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(0));
 //如果不存在Col0列，条件检查不通过。
 singleColumnValueCondition.setPassIfMissing(false);
 //只判断最新版本。
 singleColumnValueCondition.setLatestVersionsOnly(true);
```

- 构造CompositeColumnValueCondition。

```
 //composite1的条件为(Col0 == 0) AND (Col1 > 100)。
 CompositeColumnValueCondition composite1 = new CompositeColumnValueCondition(CompositeCo
lumnValueCondition.LogicOperator.AND);
 SingleColumnValueCondition single1 = new SingleColumnValueCondition("Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(0));
 SingleColumnValueCondition single2 = new SingleColumnValueCondition("Col1",
         SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong(10
0));
 composite1.addCondition(single1);
 composite1.addCondition(single2);
 //composite2的条件为( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10)。
 CompositeColumnValueCondition composite2 = new CompositeColumnValueCondition(CompositeCo
lumnValueCondition.LogicOperator.OR);
 SingleColumnValueCondition single3 = new SingleColumnValueCondition("Col2",
         SingleColumnValueCondition.CompareOperator.LESS_EQUAL, ColumnValue.fromLong(10))
;
 composite2.addCondition(composite1);
 composite2.addCondition(single3);
```

- 通过Condition实现乐观锁机制，递增一列。

```
 private static void updateRowWithCondition(SyncClient client, String pkValue) {
     //构造主键。
     PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
     primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(p
kValue));
     PrimaryKey primaryKey = primaryKeyBuilder.build();
     //读取一行数据。
     SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey)
;
     criteria.setMaxVersions(1);
     GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
     Row row = getRowResponse.getRow();
     long col0Value = row.getLatestColumn("Col0").getValue().asLong();
     //条件更新Col0列，使列值加1。
     RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
     Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
     ColumnCondition columnCondition = new SingleColumnValueCondition("Col0", SingleColum
nValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(col0Value));
     condition.setColumnCondition(columnCondition);
     rowUpdateChange.setCondition(condition);
     rowUpdateChange.put(new Column("Col0", ColumnValue.fromLong(col0Value + 1)));
     try {
         client.updateRow(new UpdateRowRequest(rowUpdateChange));
     } catch (TableStoreException ex) {
         System.out.println(ex.toString());
     }
 }
```

计费

数据写入或者更新成功，不影响各个接口的CU计算规则，如果条件更新失败，则会消耗1单位写CU和1单位读CU。

3.9. 局部事务

使用局部事务功能，创建数据范围在一个分区键值内的局部事务。对局部事务中的数据进行读写操作后，可以根据实际提交或者丢弃局部事务。局部事务通过悲观锁（Pessimistic Lock）实现并发控制。

目前局部事务功能处于邀测中，默认关闭。如果需要使用该功能，请提交工单进行申请或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。

使用局部事务功能，可以实现单行或多行读写的原子操作，扩展了使用场景。

场景

- 读-写场景（简单场景）

  当需要进行读取-修改-写回（Read-Modify-Write）操作时，可以选择如下两种方式，但两种方式有一些限制。

  - 条件更新：只能处理单行单次请求，不能处理数据分布在多行，或者需要多次写入的情况。更多信息，请参见条件更新。
  - 原子计数器：只能处理单行单次请求，且只能进行列值的累加操作。更多信息，请参见原子计数器。

  使用局部事务可以实现一个分区键值范围内的通用读取-修改-写回流程。

  i. 使用StartLocalTransaction为分区键值创建一个局部事务，并获取局部事务ID。
  ii. 使用GetRow或GetRange接口获取数据，且请求中需要带上局部事务ID。
  iii. 客户端本地修改数据。
  iv. 使用PutRow、UpdateRow、DeleteRow或BatchWriteRow接口将修改后的数据写回，且请求中需要带上事务ID。
  v. 使用CommitTransaction提交局部事务。

- 邮箱场景（复杂场景）

  使用局部事务可以实现对同一个用户邮件的原子操作。

  为了能正常使用局部事务功能，在一张物理表上同时使用了一张主表和两张索引表，其主键列请参见下表。

  其中使用Type列区分主表和不同的索引表，不同的索引行使用IndexField列保存不同含义的字段，而主表无IndexField列。

表	UserID	Type	IndexField	MailID
主表	用户ID	"Main"	"N/A"	邮件ID
Folder表	用户ID	"Folder"	$Folder	邮件ID
SendTime表	用户ID	"SendTime"	$SendTime	邮件ID

  使用局部事务功能可以完成如下操作。

- 列出某个用户发送的最近100封邮件，操作步骤如下：
  a. 使用UserID创建一个局部事务，并获取局部事务ID。
  b. 使用GetRange接口从SendTime表获取100封邮件，且请求中需要带上局部事务ID。
  c. 使用BatchGetRow接口从主表获取100封邮件的详细信息，且请求中需要带上局部事务ID。
  d. 使用CommitTransaction提交局部事务或者使用AbortTransaction丢弃局部事务。

     由于未对该局部事务进行写操作，所以提交或丢弃局部事务的操作是等同的。
- 将某个目录下的所有邮件移到另一个目录下，操作步骤如下：
  a. 使用UserID创建一个局部事务，并获取局部事务ID。
  b. 使用GetRange接口从Folder表获取若干封邮件，且请求中需要带上局部事务ID。
  c. 使用BatchWriteRow接口对Folder表进行写操作，且请求中需要带上局部事务ID。

     每封邮件对应两行写操作，一行是将对应旧Folder的行删掉，另一行是对应新Folder增加一行。
  d. 使用CommitTransaction提交局部事务。
- 统计某个目录下已读邮件与未读邮件的数量（非最优方案，详情说明请参见下文），操作步骤如下：
  a. 使用UserID创建一个局部事务，并获取局部事务ID。
  b. 使用GetRange接口从Folder表获取若干封邮件，且请求中需要带上局部事务ID。
  c. 使用BatchGetRow接口从主表获取每封邮件的已读状态。
  d. 使用CommitTransaction提交局部事务或者使用AbortTransaction丢弃局部事务。

     由于未对该局部事务进行写操作，所以提交或丢弃局部事务的操作是等同的。

在此场景中，可以通过增加新的索引表加速常用操作。使用局部事务后，无需担心主表与索引表的状态不一致，降低开发难度。例如“统计邮件数量”功能在上面的方案中需要读取很多封邮件，开销较大，可以使用一个新的索引表保存已读和未读邮件的数量，从而降低开销，加速查询。

限制

- 每个局部事务从创建开始生命周期最长为60秒。

  如果超过60秒未提交或丢弃局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
- 如果创建局部事务时超时，此请求可能在表格存储服务端已执行成功，此时用户需要等待该局部事务超时后重新创建。
- 未提交的局部事务可能失效，如果出现此情况，需要重试该局部事务内的操作。
- 在局部事务中读写数据有如下限制：
  - 不能使用局部事务ID访问局部事务范围（即创建时使用的分区键值）以外的数据。
  - 同一个局部事务中所有写请求的分区键值必须与创建局部事务时的分区键值相同，读请求则无此限制。
  - 一个局部事务同时只能用于一个请求中，在使用局部事务期间，其它使用此局部事务ID的操作均会失败。
  - 每个局部事务中两次读写操作的最大间隔为60秒。

    如果超过60秒未操作局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
  - 每个局部事务中写入的数据量最大为4 MB，按正常的写请求数据量计算规则累加。
  - 如果在局部事务中写入了未指定版本号的Cell，该Cell的版本号会在写入时（而非提交时）由表格存储服务端自动生成，生成规则与正常写入一个未指定版本号的Cell相同。
  - 如果BatchWriteRow请求中带有局部事务ID，则此请求中所有行只能操作该局部事务ID对应的表。

- 在使用局部事务期间，对应分区键值的数据相当于被锁上，只有持有局部事务ID在局部事务范围内的写请求才会成功，其它不持有局部事务ID在局部事务范围内的写请求均会失败。在局部事务提交、丢弃或超时后，对应的锁也会被释放。
- 带有局部事务ID的读写请求失败不会影响局部事务本身的存活情况，您可以按照正常的无局部事务ID的读写请求重试规则进行重试，或者主动丢弃当前局部事务。

接口

支持对局部事务进行操作的接口请参见下表。

接口	说明
StartLocalTransaction	创建一个局部事务。
CommitTransaction	提交一个局部事务。
AbortTransaction	丢弃一个局部事务。
GetRow	对局部事务范围内的数据进行读写操作。具体操作，请参见单行数据操作和多行数据操作。 说明 当前局部事务范围在一个分区键值内，分区键的更多信息，请参见主键和属性。
PutRow	
UpdateRow	
DeleteRow	
BatchWriteRow	
GetRange	

使用

您可以使用如下语言的SDK实现局部事务功能。

- Java SDK：局部事务
- Go SDK：局部事务
- Python SDK：局部事务
- Node.js SDK：局部事务
- PHP SDK：局部事务

参数

参数	说明
TableName	数据表名称。
PrimaryKey	数据表主键。 • 创建局部事务时，只需要指定局部事务对应的分区键值。 • 创建局部事务后，对局部事务范围内的数据进行读写操作时，需要指定完整主键。

参数	说明
TransactionId	局部事务ID，用于唯一标识一个局部事务。 创建局部事务后，操作局部事务时均需要带上局部事务ID。

示例

1. 调用AsyncClient或SyncClient的startLocalTransaction方法使用指定分区键值创建一个局部事务，并获取局部事务ID。

```
PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
PrimaryKey primaryKey = primaryKeyBuilder.build();
StartLocalTransactionRequest request = new StartLocalTransactionRequest(tableName, prim
aryKey);
String txnId = client.startLocalTransaction(request).getTransactionID();
```

2. 对局部事务范围内的数据进行读写操作。

   对局部事务范围内数据的读写操作与正常读写数据操作基本相同，只需填入局部事务ID即可。

   - 写入一行数据。

```
PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
primaryKeyBuilder.addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong("userId"));
PrimaryKey primaryKey = primaryKeyBuilder.build();
RowPutChange rowPutChange = new RowPutChange(tableName, primaryKey);
rowPutChange.addColumn(new Column("Col", ColumnValue.fromLong(columnValue)));
PutRowRequest request = new PutRowRequest(rowPutChange);
request.setTransactionId(txnId);
client.putRow(request);
```

   - 读取此行数据。

```
PrimaryKeyBuilder primaryKeyBuilder;
primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
primaryKeyBuilder.addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong("userId"));
PrimaryKey primaryKey = primaryKeyBuilder.build();
SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(tableName, primaryKey);
criteria.setMaxVersions(1); //设置读取最新版本的数据。
GetRowRequest  request = new GetRowRequest(criteria);
request.setTransactionId(txnId);
GetRowResponse getRowResponse = client.getRow(request);
```

3. 提交或丢弃局部事务。

   - 提交局部事务，使局部事务中的所有数据修改生效。

```
CommitTransactionRequest commitRequest = new CommitTransactionRequest(txnId);
client.commitTransaction(commitRequest);
```

   - 丢弃局部事务，局部事务中的所有数据修改均不会应用到原有数据。

```
AbortTransactionRequest abortRequest = new AbortTransactionRequest(txnId);
client.abortTransaction(abortRequest);
```

计费

- StartLocalTransaction、CommitTransaction和AbortTransaction操作分别消耗1单位写CU。
- 读写操作的计费与正常的读写请求相同，计费的更多信息，请参见计费概述。

错误码

错误码	说明
OTSRowOperationConflict	该分区键值已被其它局部事务占用。
OTSSessionNotExist	事务ID对应的事务不存在，或该事务已失效或超时。
OTSSessionBusy	该事务的上一次请求尚未结束。
OTSOutOfTransactionDataSizeLimit	事务内的数据量超过上限。
OTSDataOutOfRange	用户操作数据的分区键超出了事务创建的分区键范围。

3.10. 原子计数器

将列当成一个原子计数器使用，对该列进行原子计数操作，可用于为某些在线应用提供实时统计功能，例如统计帖子的PV（实时浏览量）等。

原子计数器可以解决由强一致性导致的写入性能开销的问题。一个RMW（Read-Modify-Write）操作，通过一次网络请求发送到服务器端，服务器端使用内部行锁机制在本地完成RMW的操作。通过原子计数器将分布式计算器的计算逻辑下推到服务器端，在保证强一致性的情况下，提升原子计数器的写入性能。

场景

使用原子计数器对某一行中的数据做实时统计。

假设您需要使用表格存储来存储图片元信息并统计图片数信息，数据表内每一行对应某一个用户ID，行上的其中一列用于存储上传的图片，另一列用于实时统计上传的图片数。

- 使用UpdateRow接口增加一张新图片时，原子计数器+1。
- 使用UpdateRow接口删除一张旧图片时，原子计数器-1。
- 使用GetRow接口读取原子计数器的值，获取当前用户的图片数。

上述行为具有强一致性，即当增加一张新图片时，原子计数器会相应+1，而不会出现-1的情况。

> **说明** 原子计数操作可能会由于网络超时、系统错误等导致失败。此时只需重试操作即可，但是可能会更新两次原子计数器，导致原子计数器偏多或偏少。针对此类异常场景，建议使用条件更新精确变更列值。

限制

- 只支持对整型列的列值进行原子计数操作。

- 作为原子计数器的列，如果写入数据前该列不存在，则默认值为0；如果写入数据前该列已存在且列值非整型，则产生OTSParameterInvalid错误。
- 增量值可以是正数或负数，但不能出现计算溢出。如果出现计算溢出，则产生OTSParameterInvalid错误。
- 默认不返回进行原子计数操作的列值，可以通过相应操作指定返回进行原子计数操作的列值。
- 在单次更新请求中，不能对某一列同时进行更新和原子计数操作。假设列A已经执行原子计数操作，则列A不能再执行其他操作（例如列的覆盖写，列删除等）。
- 在一次BatchWriteRow请求中，支持对同一行进行多次更新操作。但是如果某一行已进行原子计数操作，则该行在此批量请求中只能出现一次。
- 原子计数操作只能作用在列值的最新版本，不支持对列值的特定版本做原子计数操作。更新完成后，原子计数操作会插入一个新的数据版本。

接口

rowUpdateChange类中新增了原子计数器的操作接口，操作接口说明请参见下表。

接口	说明
RowUpdateChange increment(Column column)	对列执行增量变更，例如+X，-X等。
void addReturnColumn(String columnName)	对于进行原子计数操作的列，设置需要返回列值的列名。
void setReturnType(ReturnType returnType)	设置返回类型，返回进行原子计数操作的列的新值。

使用

您可以使用如下语言的SDK实现原子计数器功能。

- Java SDK：原子计数器
- Go SDK：原子计数器
- Python SDK：原子计数器
- Node.js SDK：原子计数器
- .NET SDK：原子计数器

参数

参数	说明
tableName	数据表名称。
columnName	进行原子计数操作的列名。只支持对整型列的列值进行原子计数器操作。
value	对列进行增量变更的值。
returnType	设置返回类型为ReturnType.RT_AFTER_MODIFY，将进行原子计数操作的列值返回。

示例

写入数据时，使用rowUpdateChange接口对整型列做列值的增量变更，然后读取更新后的新值。

```
private static void incrementByUpdateRowApi(SyncClient client) {
        //构造主键。
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
        primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(
"pk0"));
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        //设置数据表名称。
        RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
        //将进行原子计数操作的price列的列值+10，不能设置时间戳。
        rowUpdateChange.increment(new Column("price", ColumnValue.fromLong(10)));
        //设置returnType为ReturnType.RT_AFTER_MODIFY，将进行原子计数操作的列值返回。
        rowUpdateChange.addReturnColumn("price");
        rowUpdateChange.setReturnType(ReturnType.RT_AFTER_MODIFY);
        //对price列进行原子计数操作。
        UpdateRowResponse response = client.updateRow(new UpdateRowRequest(rowUpdateChange)
);
        //打印更新后的新值。
        Row row = response.getRow();
        System.out.println(row);
    }
```

计费

使用原子计数器功能不影响现有计费规则。

3.11. 过滤器

在服务端对读取结果再进行一次过滤，根据过滤器（Filter）中的条件决定返回的行。使用过滤器后，只返回符合条件的数据行，从而有效降低网络传输的数据量，减少响应时间。

场景

- 直接过滤结果

  以物联网中的智能电表为例，智能电表按一定的频率（例如每隔15秒）将当前的电压、电流、度数等信息写入表格存储。在按天做分析时，需要获取某一个电表当天是否出现过电压异常以及出现时的其他状态数据，用于判断是否需要对某条线路进行检修。

  按照目前的方案，使用GetRange读取一个电表一天内的所有的监控数据，共有5760条，然后再对5760条信息进行过滤，最终获取了10个电压出现不稳定时的监控信息。

  使用过滤器只返回了实际需要的10条数据，有效降低了返回的数据量。而且无需再对结果进行初步的过滤处理，节省了开发成本。

- 正则匹配并转换数据类型后再过滤结果

  当某些列中存储了自定义格式数据（例如JSON格式字符串）时，如果用户希望过滤查询该列的某个子字段值，则可以通过正则表达式匹配并转换子字段值类型后，再使用过滤器来过滤需要的数据。

  例如列中存储的数据为 `{cluster_name:name1,lastupdatetime:12345}` 格式，如果需要过滤查询lastupdatetime>12345的行数据，此时您可以通过正则表达式 `lastupdatetime:([0-9]+)}` 来匹配该列中子字段的数据，然后将匹配结果使用CAST转换为数值类型，再进行数值类型的比较，从而过滤得到所需要的数据行。

限制

- 过滤器的条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。
- 过滤器中的参考列必须在读取的结果内。如果指定的要读取的列中不包含参考列，则过滤器无法获取参考列的值。
- 使用GetRange接口时，一次扫描数据的行数不能超过5000行或者数据大小不能超过4 MB。

  当在该次扫描的5000行或者4 MB数据中没有满足过滤器条件的数据时，得到的Response中的Rows为空，但是NextStartPrimaryKey可能不为空，此时需要使用NextStartPrimaryKey继续读取数据，直到NextStartPrimaryKey为空。

接口

过滤器可以用于GetRow、BatchGetRow和GetRange。在GetRow、BatchGetRow和GetRange接口中使用过滤器不会改变接口的原生语义和限制项，具体操作，请参见单行数据操作和多行数据操作。

过滤器目前包括SingleColumnValueFilter、SingleColumnValueRegexFilter和CompositeColumnValueFilter，是基于一个或者多个参考列的列值决定是否过滤某行。

- SingleColumnValueFilter：只判断某个参考列的列值。

  当参考列不存在时，可以使用PassIfMissing参数决定此时是否满足条件，即设置当参考列不存在时的行为。

- SingleColumnValueRegexFilter：支持对类型为String的列值，使用正则表达式进行子字符串匹配，然后根据实际将匹配到的子字符串转换为String、Integer或者Double类型，再使用过滤器进行过滤。

  > 注意 只有Java SDK支持使用SingleColumnValueRegexFilter过滤器。

- CompositeColumnValueFilter：根据多个参考列的列值的判断结果进行逻辑组合，决定是否过滤某行。

使用

您可以使用如下语言的SDK实现过滤器功能。

- Java SDK：过滤器
- Go SDK：过滤器
- Python SDK：过滤器
- Node.js SDK：过滤器
- .NET SDK：过滤器
- PHP SDK：过滤器

参数

参数	说明
ColumnName	过滤器中参考列的名称。
ColumnValue	过滤器中参考列的对比值。
CompareOperator	过滤器中的关系运算符。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=）。

参数	说明
LogicOperator	过滤器中的逻辑运算符。 逻辑运算符包括NOT、AND和OR。
PassIfMissing	当参考列在某行中不存在时，是否返回该行。取值范围如下： • true（默认）：如果参考列在某行中不存在时，则返回该行。 • false：如果参考列在某行中不存在时，则不返回该行。
LatestVersionsOnly	当参考列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果参考列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置LatestVersionsOnly为false时，如果参考列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就返回该行。
Regex	正则表达式，用于匹配子字段值。正则表达式必须满足以下条件： • 长度不能超过256个字节。 • 支持perl regular语法。 • 支持单字节正则表达式。 • 不支持中文的正则匹配。 • 支持正则表达式的全匹配模式和部分匹配模式。 部分匹配的正则表达式在模式中由一对括号（...）分隔。 如果正则表达式为全匹配模式，则返回第一个匹配结果；如果正则表达式中包含部分匹配语法，则返回第一个满足的子匹配结果。例如列值为1aaa51bbb5，如果正则表达式为1[a-z]+5时，则返回值为1aaa5；如果正则表达式为1([a-z]+)5，则返回值为aaa。
VariantType	使用正则表达式匹配到子字段值后，子字段值转换为的类型。取值范围为VT_INTEGER（整型）、VT_STRING（字符串类型）和VT_DOUBLE（双精度浮点型）。

示例

- 构造SingleColumnValueFilter。

```
//设置过滤器，当Col0列的值为0时，返回该行。
SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
        SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
//如果不存在Col0列，也不返回该行。
singleColumnValueFilter.setPassIfMissing(false);
//只使用该列最新版本的值进行比较。
singleColumnValueFilter.setLatestVersionsOnly(true);
```

- 构造SingleColumnValueRegexFilter。

```
 //构造正则抽取规则。
 RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
 //设置过滤器，实现cast<int>(regex(col1)) > 0。
 //构造SingleColumnValueRegexFilter，格式为"列名,正则规则,比较符,比较值"。
 SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
     regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(0));
 //如果不存在Col0列，也不返回该行。
 filter.setPassIfMissing(false);
```

- 构造CompositeColumnValueFilter。

```
  //composite1的条件为(Col0 == 0) AND (Col1 > 100)。
  CompositeColumnValueFilter composite1 = new CompositeColumnValueFilter(CompositeColumnV
alueFilter.LogicOperator.AND);
  SingleColumnValueFilter single1 = new SingleColumnValueFilter("Col0",
          SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
  SingleColumnValueFilter single2 = new SingleColumnValueFilter("Col1",
          SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromLong(100)
);
  composite1.addFilter(single1);
  composite1.addFilter(single2);
  //composite2的条件为( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10)。
  CompositeColumnValueFilter composite2 = new CompositeColumnValueFilter(CompositeColumnV
alueFilter.LogicOperator.OR);
  SingleColumnValueFilter single3 = new SingleColumnValueFilter("Col2",
          SingleColumnValueFilter.CompareOperator.LESS_EQUAL, ColumnValue.fromLong(10));
  composite2.addFilter(composite1);
  composite2.addFilter(single3);
```

计费

使用过滤器功能不影响现有计费规则。

使用过滤器后，可以有效降低返回的数据量，但是由于过滤计算是服务器端在返回数据前进行的，并未降低磁盘IO次数，所以消耗的读CU与不使用过滤器时相同。例如使用GetRange读取到100条记录，共200 KB数据，消耗了50单位读CU，在使用过滤器后，实际只返回了10条数据，共20 KB，但是仍然会消耗50单位读CU。

错误码

更多信息，请参见错误码参考。

4.消息模型

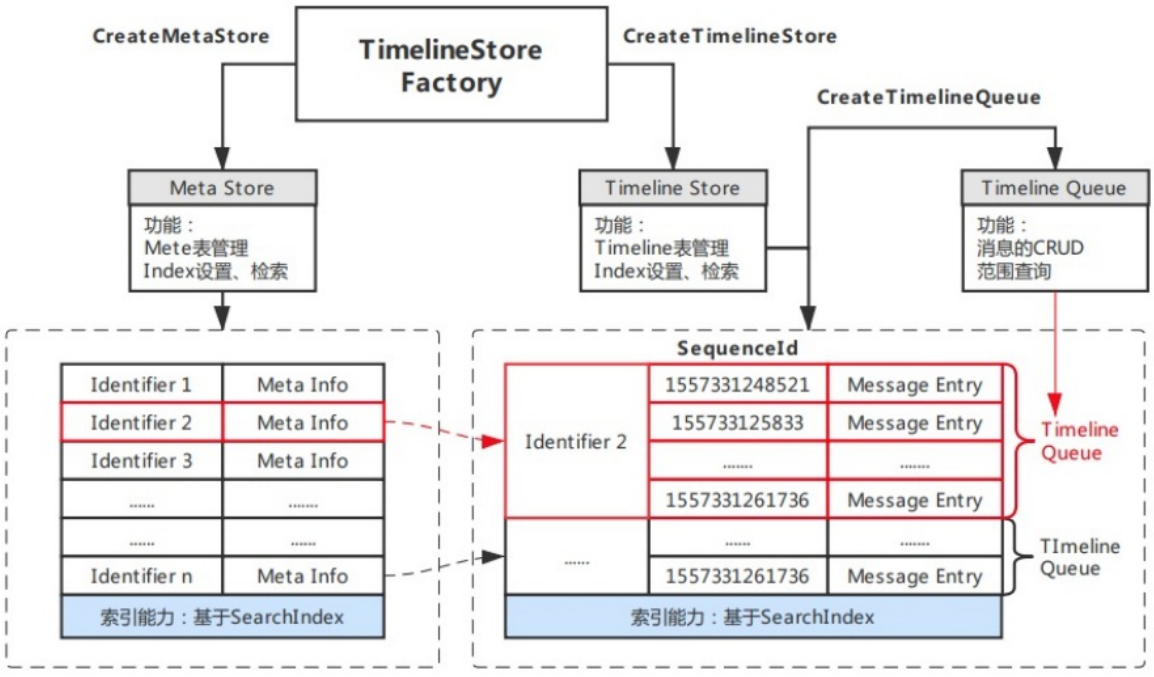
4.1. 模型介绍

消息（Timeline）模型是针对消息数据场景所设计的，能够满足消息数据场景对消息保序、海量消息存储、实时同步的业务需求，同时支持全文检索与多维度组合查询。适用于IM、Feed流等消息场景。

模型结构

消息模型以简单为设计目标，核心模块构成比较清晰明了。消息模型尽量提升使用的自由度，让您能够根据自身场景需求选择更为合适的实现。消息模型的架构主要包括：

- Store：Timeline存储库，类似数据库的表的概念。
- Identifier：用于区分Timeline的唯一标识。
- Meta：用于描述Timeline的元数据，元数据描述采用free-schema结构，可自由包含任意列。
- Queue：一个Timeline内所有Message存储在Queue内。
- SequenceId：Queue中消息体的序列号，需保证递增、唯一。模型支持自增列、自定义两种实现模式。
- Message：Timeline内传递的消息体，是一个free-schema的结构，可自由包含任意列。
- Index：包含Meta Index和Message Index，可对Meta或Message内的任意列自定义索引，提供灵活的多条件组合查询和搜索。



功能介绍

消息模型支持以下功能。

- 支持Meta、消息的基本管理（数据的CRUD）。
- 支持Meta、消息的多维组合查询、全文检索。
- 支持SequenceId的两种设置：自增列、手动设置。
- 支持多列的Timeline Identifier。

- 兼容Timeline 1.X模型，提供的TimelineMessageForV1样例可直接读写V1版本消息。

Tablestore Java SDK（消息模型已经合入到SDK中）

```
<dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>tablestore</artifactId>
    <version>4.12.1</version>
</dependency>
```

如果使用了4.12.1之前SDK，则需要单独引用以下依赖。

```
<dependency>
    <groupId>com.aliyun.openservices.tablestore</groupId>
    <artifactId>Timeline</artifactId>
    <version>2.0.0</version>
</dependency>
```

4.2. 快速入门

本文主要为您介绍如何通过示例代码快速使用消息（Timeline）模型。

操作步骤

1. 登录表格存储控制台，并创建表格存储实例。具体操作，请参见创建实例。
2. 下载并安装表格存储Java SDK包。具体操作，请参见安装。
3. 使用实例服务地址及AccessKey初始化对接实例。具体操作，请参见初始化。
4. 通过示例代码快速使用消息模型。更多信息，请参见示例代码。

4.3. 基础操作

4.3.1. 概述

消息（Timeline）模型是针对消息数据场景所设计的，能够满足消息数据场景对消息保序、海量消息存储、实时同步的业务需求，同时支持全文检索与多维度组合查询。适用于IM、Feed流等消息场景。

消息模型Java SDK包含以下操作：

- 初始化
- Meta管理
- Timeline管理
- Queue管理

4.3.2. 初始化

本文介绍使用消息模型时如何初始化。

初始化Factory

将SyncClient作为参数，初始化StoreFactory。通过Store工厂创建Meta数据和Timeline数据的管理Store。

实现错误重试需要依赖SyncClient的重试策略，您可以通过配置SyncClient实现重试。如果有特殊需求，可自定义策略（只需实现RetryStrategy接口）。

```
/**
 * 重试策略配置。
 * Code: configuration.setRetryStrategy(new DefaultRetryStrategy());
 **/
ClientConfiguration configuration = new ClientConfiguration();
SyncClient client = new SyncClient(
        "http://instanceName.cn-shanghai.ots.aliyuncs.com",
        "accessKeyId",
        "accessKeySecret",
        "instanceName", configuration);
TimelineStoreFactory serviceFactory = new TimelineStoreFactoryImpl(client);
```

初始化MetaStore

构建Meta表的Schema，包含Identifier、MetaIndex等参数，通过Store工厂创建并获取Meta的管理Store，配置参数包含Meta表名、索引名、主键字段、索引类型等。

```
TimelineIdentifierSchema idSchema = new TimelineIdentifierSchema.Builder()
        .addStringField("timeline_id").build();
IndexSchema metaIndex = new IndexSchema();
metaIndex.addFieldSchema( //配置索引字段以及字段类型。
        new FieldSchema("group_name", FieldType.TEXT).setIndex(true).setAnalyzer(FieldSchem
a.Analyzer.MaxWord),
        new FieldSchema("create_time", FieldType.Long).setIndex(true)
);
TimelineMetaSchema metaSchema = new TimelineMetaSchema("groupMeta", idSchema)
        .withIndex("metaIndex", metaIndex); //配置索引。
TimelineMetaStore timelineMetaStore = serviceFactory.createMetaStore(metaSchema);
```

- 建表

  根据metaSchema的参数创建表。如果metaSchema中配置了索引，建表成功后会创建索引。

  ```
  timelineMetaStore.prepareTables();
  ```

- 删表

  当表存在索引时，在删除表前，系统会先删除索引，再删除Store相应表。

  ```
  timelineMetaStore.dropAllTables();
  ```

初始化TimelineStore

构建timeline表的Schema配置，包含Identifier、TimelineIndex等参数，通过Store工厂创建并获取Timeline的管理Store；配置参数包含Timeline表名、索引名、主键字段、索引类型等。

消息的批量写入，基于Tablestore的DefaultTableStoreWriter提升并发，用户可以根据自己需求设置线程池数目。

```
TimelineIdentifierSchema idSchema = new TimelineIdentifierSchema.Builder()
        .addStringField("timeline_id").build();
IndexSchema timelineIndex = new IndexSchema();
timelineIndex.setFieldSchemas(Arrays.asList(//配置索引的字段以及字段类型。
        new FieldSchema("text", FieldType.TEXT).setIndex(true).setAnalyzer(FieldSchema.Anal
yzer.MaxWord),
        new FieldSchema("receivers", FieldType.KEYWORD).setIndex(true).setIsArray(true)
));
TimelineSchema timelineSchema = new TimelineSchema("timeline", idSchema)
        .autoGenerateSeqId() //SequenceId 设置为自增列方式。
        .setCallbackExecuteThreads(5) //设置Writer初始线程数为5。
        .withIndex("metaIndex", timelineIndex); //设置索引。
TimelineStore timelineStore = serviceFactory.createTimelineStore(timelineSchema);
```

- 建表

  根据TimelineSchema的参数创建表。如果TimelineSchema中配置了索引，建表成功后会创建索引。

  ```
  timelineStore.prepareTables();
  ```

- 删表

  当表存在索引时，在删除表前，系统会先删除索引，再删除Store相应的表。

  ```
  timelineStore.dropAllTables();
  ```

4.3.3. Meta管理

Meta管理提供了增、删、改、单行读、多条件组合查询等接口。

Meta管理的多条件组合查询功能基于多元索引，只有设置了IndexSchema的MetaStore才支持。索引类型支持LONG、DOUBLE、BOOLEAN、KEYWORD、GEO_POINT等类型，属性包含Index、Store和Array，其含义与多元索引相同。更多信息，请参见数据类型映射。

Insert

TimelineIdentifier是区分Timeline的唯一标识，重复的Identifier会被覆盖。

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group")
        .build();
TimelineMeta meta = new TimelineMeta(identifier)
        .setField("filedName", "fieldValue");
timelineMetaStore.insert(meta);
```

Read

根据Identifier读取单行TimelineMeta数据。

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group")
        .build();
timelineMetaStore.read(identifier);
```

Update

更新TimelineIdentifier所对应的Meta属性。

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group")
        .build();
TimelineMeta meta = new TimelineMeta(identifier)
        .setField("filedName", "new value");
timelineMetaStore.update(meta);
```

Delete

根据Identifier删除单行TimelineMeta数据。

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group")
        .build();
timelineMetaStore.delete(identifier);
```

Search

提供两种查询参数，SearchParameter以及SDK原生类SearchQuery，返回Iterator<TimelineMeta>，通过迭代器遍历。

```
/**
 * Search meta by SearchParameter.
 * */
SearchParameter parameter = new SearchParameter(
        field("fieldName").equals("fieldValue")
);
timelineMetaStore.search(parameter);
/**
 * Search meta by SearchQuery.
 * */
TermQuery query = new TermQuery();
query.setFieldName("fieldName");
query.setTerm(ColumnValue.fromString("fieldValue"));
SearchQuery searchQuery = new SearchQuery().setQuery(query);
timelineMetaStore.search(searchQuery);
```

4.3.4. Timeline管理

Timeline管理提供了消息模糊查询、多条件组合查询接口。

Timeline管理的查询功能基于多元索引，只有设置了IndexSchema的TimelineStore才支持。索引类型支持LONG、DOUBLE、BOOLEAN、KEYWORD、GEO_POINT、TEXT等类型，属性包含Index、Store、Array以及分词器，其含义与多元索引相同。更多信息，请参见数据类型映射。

Search

多维度组合查询，需要模糊查询的字段，按照需求创建TEXT类型，并设置需要的分词器。

```
/**
 * Search timeline by SearchParameter.
 * */
SearchParameter searchParameter = new SearchParameter(
        field("text").equals("fieldValue")
);
timelineStore.search(searchParameter);
/**
 * Search timeline by SearchQuery.
 * */
TermQuery query = new TermQuery();
query.setFieldName("text");
query.setTerm(ColumnValue.fromString("fieldValue"));
SearchQuery searchQuery = new SearchQuery().setQuery(query).setLimit(10);
timelineStore.search(searchQuery);
```

Flush

批量写基于表格存储SDK中DefaultTableStoreWriter实现，可以主动调用flush接口，将Buffer中尚未发出的请求主动触发发送，同步等待至所有消息写入成功。

```
/**
 * Flush messages in buffer, and wait until all messages are stored.
 * */
timelineStore.flush();
```

4.3.5. Queue管理

本文介绍使用消息模型时如何进行Queue管理。

获取Queue实例

Queue是单个消息队列的抽象概念，对应TimelineStore下单个Identifier的所有消息。获取Queue实例时通过TimelineStore的接口创建。

```
TimelineIdentifier identifier = new TimelineIdentifier.Builder()
        .addField("timeline_id", "group_1")
        .build();
//单TimelineStore下单identifier对应的消息队列（Queue）。
TimelineQueue timelineQueue = timelineStore.createTimelineQueue(identifier);
```

Queue是单存储库下单Identifier对应的消息队列的管理实例，主要有同步写、异步写、批量写、删、同步改、异步改、单行读、范围读等接口。

Store

同步存储消息，两个接口分别支持SequenceId的两种实现方式自增列和手动设置，相关配置在TimelineSchema中。

```
timelineQueue.store(message); //自增列实现的SequenceId。
timelineQueue.store(sequenceId, message); //手动设置SequenceId。
```

StoreAsync

异步存储消息，您可以自定义回调，对成功或者失败做自定义处理。接口返回Future<TimelineEntry>。

```
TimelineCallback callback = new TimelineCallback() {
    @Override
    public void onCompleted(TimelineIdentifier i, TimelineMessage m, TimelineEntry t) {
        // do something when succeed.
    }
    @Override
    public void onFailed(TimelineIdentifier i, TimelineMessage m, Exception e) {
        // do something when failed.
    }
};
timelineQueue.storeAsync(message, callback); //自增列实现的SequenceId。
timelineQueue.storeAsync(sequenceId, message, callback); //手动设置SequenceId。
```

BatchStore

批量存储消息，支持无回调和有回调两种。您可以自定义回调，对成功或者失败做自定义处理。

```
timelineQueue.batchStore(message); //自增列实现的SequenceId。
timelineQueue.batchStore(sequenceId, message); //手动设置SequenceId。
timelineQueue.batchStore(message, callback); //自增列实现的SequenceId。
timelineQueue.batchStore(sequenceId, message, callback); //手动设置SequenceId。
```

Get

通过SequenceId读取单行消息。当消息不存在时不抛错，返回null。

```
timelineQueue.get(sequenceId);
```

GetLatestTimelineEntry

读取最新一条消息。当消息不存在时不抛错，返回null。

```
timelineQueue.getLatestTimelineEntry();
```

GetLatestSequenceId

获取最新一条消息的SequenceId。当消息不存在时不抛错，返回0。

```
timelineQueue.getLatestSequenceId();
```

Update

通过SequenceId同步更新消息内容。

```
TimelineMessage message = new TimelineMessage().setField("text", "Timeline is fine.");
//update message with new field
message.setField("text", "new value");
timelineQueue.update(sequenceId, message);
```

UpdateAsync

通过SequenceId异步更新消息。您可以自定义回调，对成功或者失败做自定义处理。接口返回Future<TimelineEntry>。

```
TimelineMessage oldMessage = new TimelineMessage().setField("text", "Timeline is fine.");
TimelineCallback callback = new TimelineCallback() {
    @Override
    public void onCompleted(TimelineIdentifier i, TimelineMessage m, TimelineEntry t) {
        // do something when succeed.
    }
    @Override
    public void onFailed(TimelineIdentifier i, TimelineMessage m, Exception e) {
        // do something when failed.
    }
};
TimelineMessage newMessage = oldMessage;
newMessage.setField("text", "new value");
timelineQueue.updateAsync(sequenceId, newMessage, callback);
```

Delete

根据SequenceId删除单行消息。

```
timelineQueue.delete(sequenceId);
```

Scan

根据Scan参数正序（或逆序）范围读取单个Queue下的消息，返回Iterator<TimelineEntry>，通过迭代器遍历。

```
ScanParameter scanParameter = new ScanParameter().scanBackward(Long.MAX_VALUE, 0);
timelineQueue.scan(scanParameter);
```

5.时序模型

5.1. 时序模型概述

通过时序模型，您可以对时间序列进行存储、查询和分析。

注意事项

- 时序模型功能将从2022年05月26日正式开始收费。
- 目前支持使用时序模型功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

如果使用过程中遇到问题，请通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

背景信息

表格存储是阿里云自研的多模型结构化数据存储，支持多种数据模型，包括时序模型。

表格存储的时序模型是针对时间序列数据的特点进行设计，适用于物联网设备监控、设备采集数据、机器监控数据等场景。主要优势如下：

- 通用的时序数据建模方式，用户无须预定义表结构。
- 支持自动构建时间序列的元数据索引，支持按照多种组合条件检索时间序列。
- 支持SQL查询以及通过SQL进行聚合统计操作。
- 服务能力自动水平扩展、支持高并发写入和查询以及PB级海量数据的低成本存储。

基础概念

概念	描述
时序数据	由多个时间序列组成，每个时间序列表示一组按照时间顺序排列的数据点。除了数据点之外，还需要一些元数据用来标识一个时间序列。因此时序数据由元数据和数据两部分组成。 • 元数据：记录了所有时间序列的标识信息和属性信息。 • 数据：记录了所有时间序列的数据点，数据点包括产生数据点的时间和对应的数据值。
时间线	在时序数据场景中，常用时间线来代指一个时间序列。在表格存储时序模型中，时间线与时间序列也是等价的说法。
时间线元数据	时间线元数据也称为时间序列元数据，表示一条时间线的标识和属性信息。时间线标识用来唯一确定一条时间线，属性信息支持修改，可用于时间线检索。
时间线标识	时间线标识也称为时间序列标识，用来唯一标识一条时间线。在表格存储的时序模型中，时间线标识由度量名称、数据源和标签三部分组成。
度量名称	时间线数据所度量的物理量或者监控指标的名称，例如cpu或net，用于表示该时间序列记录的是cpu或者网络使用率等。
数据源	产生时间线的数据源标识，可以为空。

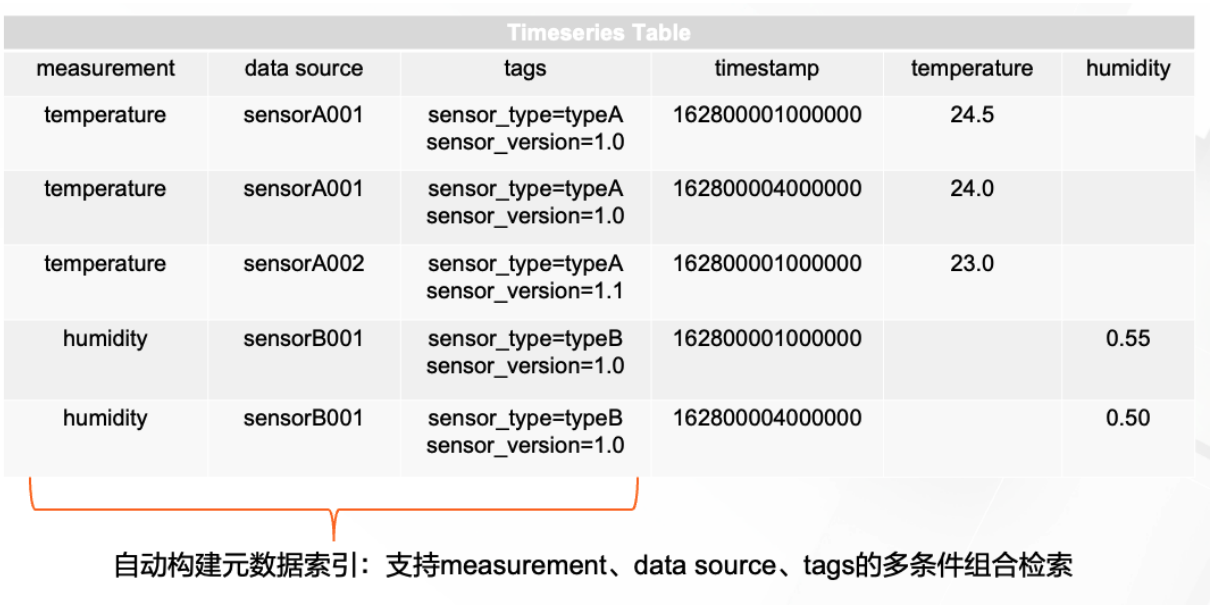
概念	描述
标签	时间线的标签信息。您可以自定义多个字符串类型的key-value对。
属性	属性属于时间线元数据的一部分，可用于记录该时间线的一些可变属性信息，但不作为时间线的标识，不用于唯一确定一个时间序列。时间线的属性在格式上类似于标签，为多个字符串类型的key-value对。您可以设置或者更新某个时间线的属性，用于后续通过属性进行时间线的检索。
时间线数据	一条时间线的数据点由产生数据的时间和数据值两部分组成。如果每个时间线每个时刻仅产生一个值，则为单值模型；如果每个时刻对应多个值，则为多值模型。 表格存储的时序模型为多值模型，在一个时间点上可以设置多个数据值。每个值对应数据库中的一列，包括列名和列值。列值支持多种数据类型，包括布尔、整型、浮点数、字符串和二进制。

数据模型

在表格存储的时序模型中，采用一张二维的时序表来存储时序数据。

每行代表一个时间线在某个时间点的数据，该行的主键部分为时间线标识和时间戳，该行的数据列部分为该时间线在该时间戳下的数据点，可以有多个数据列。主键结构和数据列的结构无须用户进行预先定义，用户仅需要在写入时指定数据列的列名即可。

一个时序表支持存储不同度量类别的时序数据。以下图为例，时序表中存储了温度（temperature）和湿度（humidity）两种度量类别的数据。



图中度量名称（measurement）、数据源（data source）和标签（tags）组成了一个时间线标识。此外，您还可以通过接口更新某个时间序列的元数据属性（attributes），该元数据信息可以用于时间线的检索。

数据写入后，系统会自动提取该时间线的元数据信息并自动构建索引，支持按照度量名称、数据源以及标签的组合条件进行时间线检索。

功能特性

- 创建和管理时序表

通过控制台、SDK、CLI工具等方式列出实例中的全部时序表、创建一张时序表、查询时序表的配置信息、更新时序表的配置信息以及删除一张时序表。

在创建或者更新时序表的配置信息时，您可以设置时序表中数据自动过期时间（TimeToLive），系统将自动判断当前时间与时序数据中用户写入的时间戳，超过过期时间后数据会被自动删除。

- 读写时序数据

  通过控制台、SDK、CLI工具等方式将时序数据批量写入一张时序表中。数据写入后，您可以通过指定时间线标识来查询一条时间线在某段时间范围内的数据。

- 时间序列检索

  通过控制台、SDK、CLI工具等方式检索一张时序表中的时间线。检索条件支持多种条件组合，例如查询“度量名称为cpu”、“标签中包含名称为region和值为hangzhou的标签”且“属性中包含名称为status和值为online”的所有时间线。检索到时间线后，可以通过调用接口进一步查询该时间线中的数据。

- SQL查询分析

  时序表支持通过SQL进行查询，SQL中支持通过指定时间线的元数据条件筛选时间线以及通过统计聚合操作按照不同维度对数据进行聚合操作，例如查询某一批设备采样数据的平均值、将秒级数据聚合为分钟级数据等。

  此外，SQL还支持仅对时间线的元数据进行查询，方便通过SQL进行时间线的元数据管理。

使用限制

更多信息，请参见时序模型限制。

使用方式

您可以使用控制台、命令行工具或者SDK快速体验时序模型。

- 使用控制台
- 使用命令行工具
- 使用SDK

计费

更多信息，请参见时序模型计量计费。

5.2. 创建时序模型实例

创建时序模型实例后，您可以使用控制台、CLI工具或者SDK快速体验时序模型功能。

操作步骤

1. 登录表格存储控制台。
2. 在**概览**页面，单击**创建时序模型实例**。
3. 在页面上方，选择地域，例如华东1（杭州）、华南1（深圳）等。
4. 在**创建时序模型实例**对话框，选择实例规格，输入实例名称并根据业务需求填写实例注释。

   关于实例命名规则以及如何选择实例规格，请参见实例。

> 注意
> - 实例规格在实例创建后无法修改。
> - 单个阿里云账户最多可以创建10个实例，且在同一地域中实例名称必须唯一。

5. 单击**确定**。

后续步骤

- 使用控制台快速体验时序模型功能。具体操作，请参见使用控制台。
- 使用命令行工具快速体验时序模型功能。具体操作，请参见使用命令行工具。
- 使用SDK快速体验时序模型功能。具体操作，请参见使用SDK。

5.3. 快速入门

5.3.1. 使用控制台

通过控制台创建时序表后，您可以写入时序数据并对时间线进行检索，以及查询时序数据。

前提条件

已创建时序模型实例。具体操作，请参见创建时序模型实例。

注意事项

- 时序模型功能将从2022年05月26日正式开始收费。
- 目前支持使用时序模型功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

如果使用过程中遇到问题，请通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

步骤一：创建时序表

通过控制台创建一张时序表。

1. 登录表格存储控制台。
2. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
3. 在**实例详情**页签，单击**时序表列表**。
4. 在**时序表列表**页签，单击**创建时序表**。

> **说明** 您也可以单击**一键生成样例**，创建一张测试表以及样例数据来快速体验功能。对于新建的测试表，系统会进行一些初始化操作，因此需要等待几十秒左右才能看到时间线。

5. 在**创建时序表**对话框，按照如下说明配置**时序表名称**和**数据生命周期**。

参数	描述

参数	描述
时序表名称	时序表的名称，用于在实例中唯一标识一张时序表。 时序表的命名规则为由大小写字母、数字或下划线（_）组成，且只能以字母或下划线（_）开头，长度在1~128个字符之间。 时序表名称不能与已存在的数据表名称重复。
数据生命周期	时序表中数据的过期时间，单位为秒。当系统判断当前时间减去用户传入数据列的时间已经超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 **注意** 在时序表中，系统判断数据产生时间以用户传入的时间列为准，并非数据写入表中的时间。 取值必须大于等于86400秒（一天）或者必须为-1（数据永不过期）。

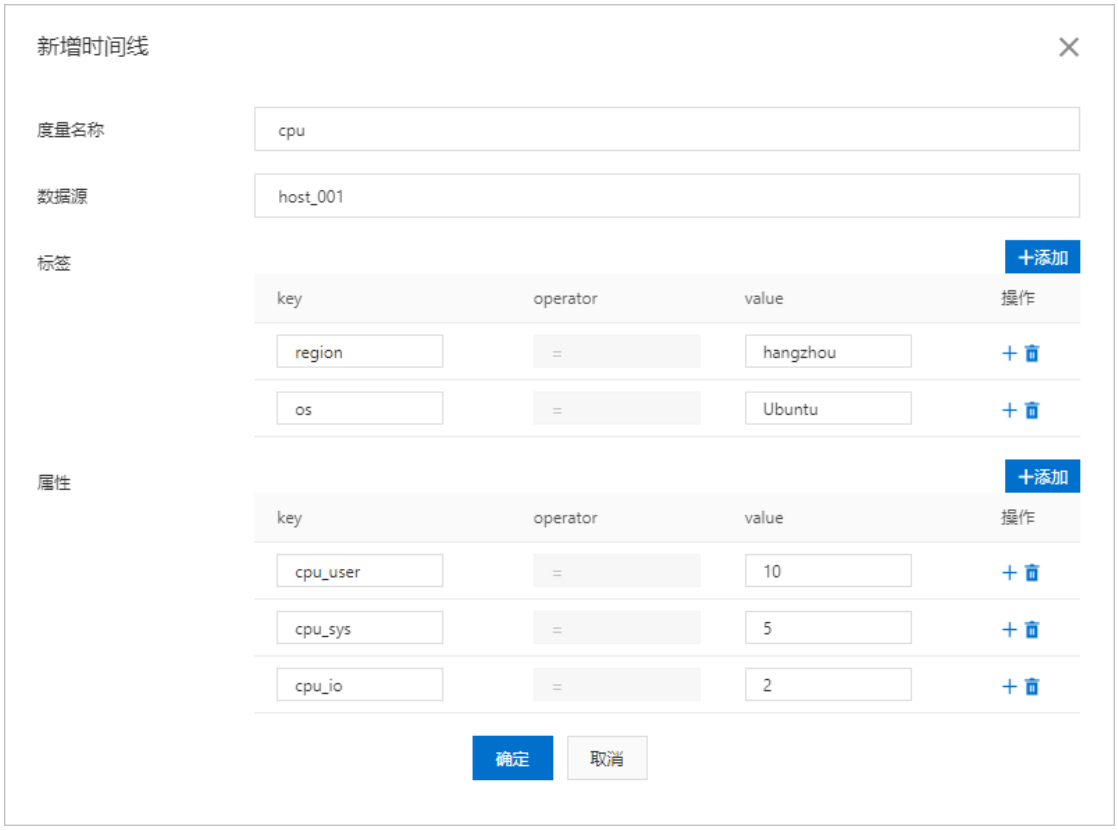
6. 单击**确定**。

   时序表创建完成后，在**时序表列表**页签，可以看到已经创建的时序表。如果新建的表未显示在列表中，可单击 图标，刷新时序表列表。

步骤二：写入数据

通过控制台向时序表中写入时序数据。时序数据由元数据和数据两部分组成，如果未预先新建元数据，则系统会根据写入的数据自动提取元数据。

1. 在**时序表列表**页签，单击时序表名称后选择**数据管理**页签或在操作列单击**数据管理**。
2. （可选）新建时间线。
   i. 在**数据管理**页签，单击**新增时间线**。
   ii. 在**新增时间线**对话框，新增时间线元数据。



详细参数说明请参见下表。

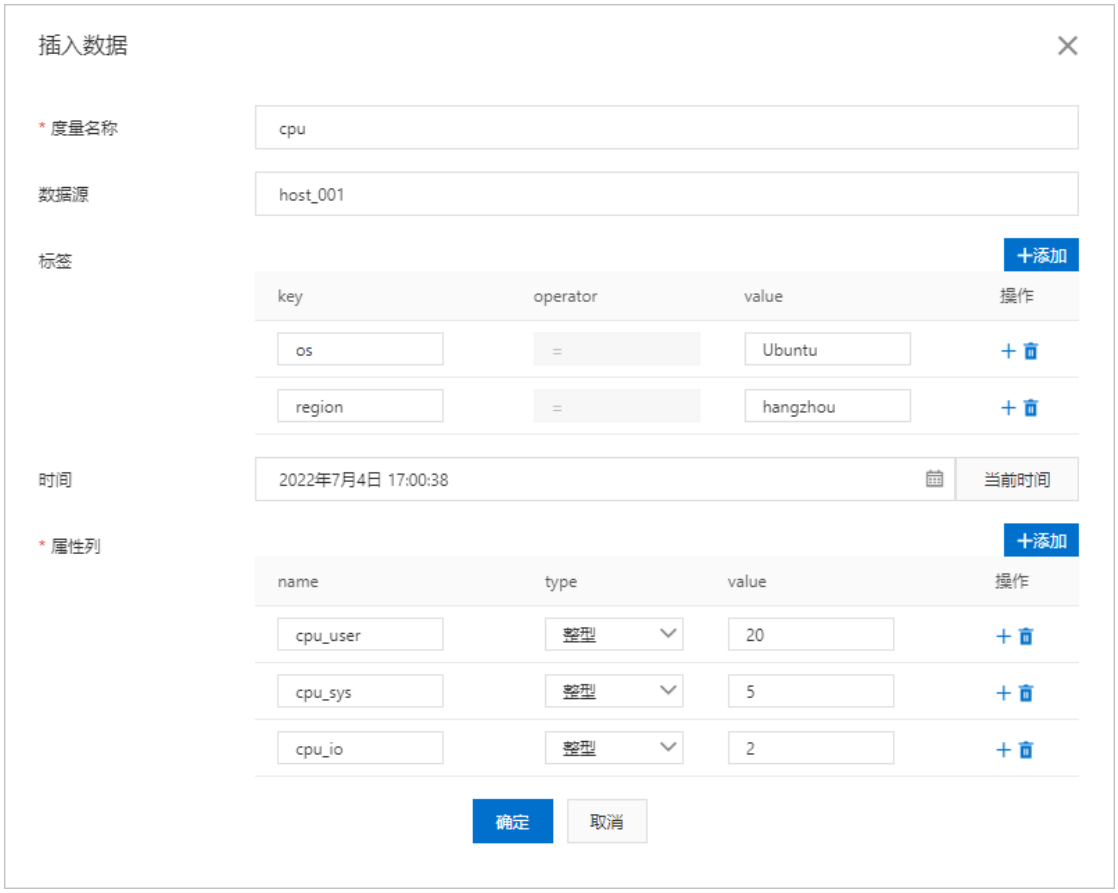
参数	描述
度量名称	时间线数据所度量的物理量或者监控指标的名称，例如cpu或net，用于表示该时间线记录的是cpu或者网络使用率等。
数据源	产生时间线的数据源标识，可以为空。
标签	时间线的标签信息。您可以自定义多个字符串类型的key-value对。
属性	时间线的属性列，用于记录该时间线的一些属性信息。

iii. 单击**确定**。

3. 插入数据。

i. 单击**插入数据**。

ii. 在**插入数据**对话框，设置时间和属性列。



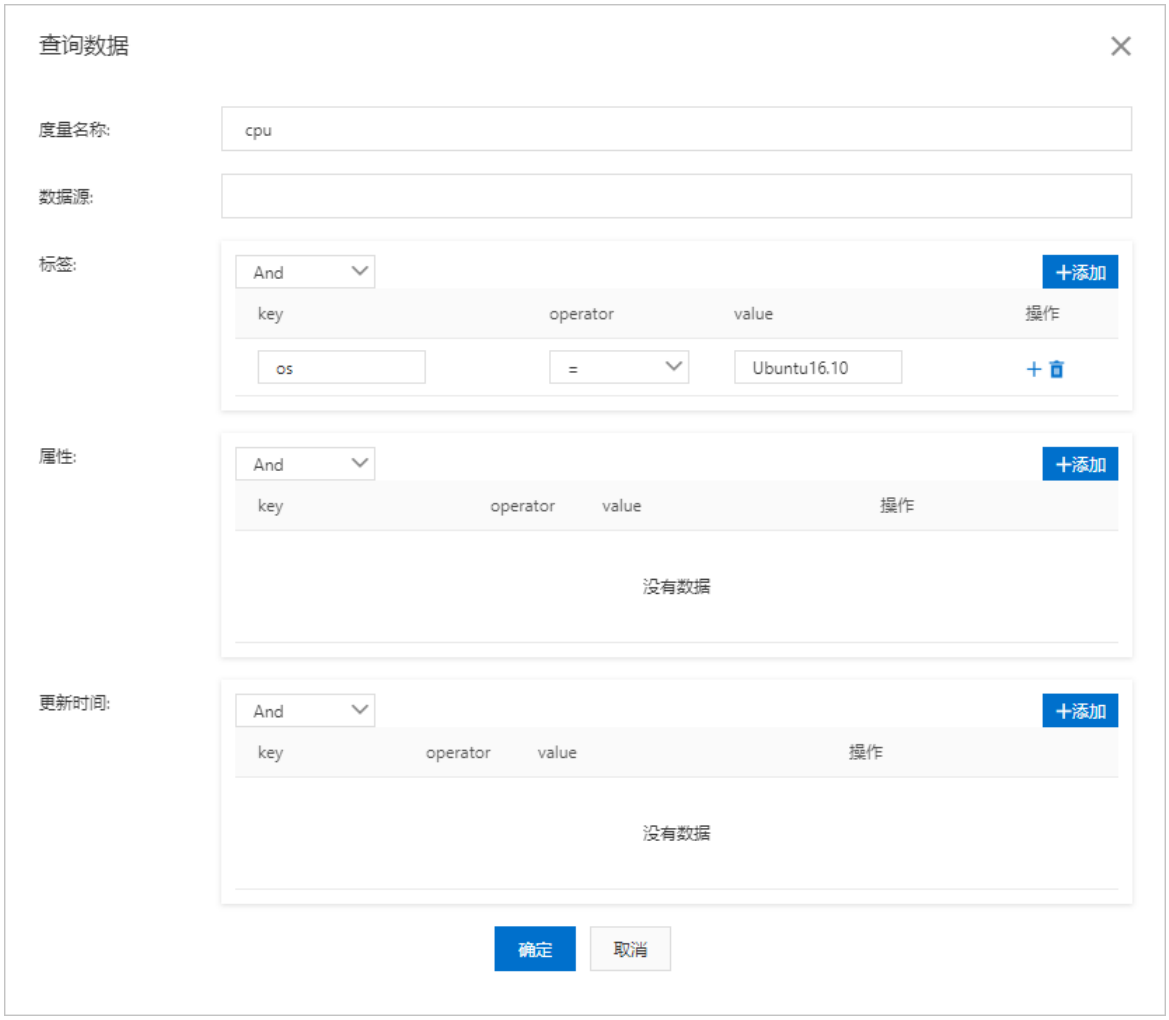
iii. 单击**确定**。

步骤三：检索时间线

检索符合指定条件的所有时间线。

1. 在**数据管理**页签，单击右上角的**查询数据**。
2. 在**查询数据**对话框，输入时间线的度量名称，根据实际需要输入数据源以及单击对应区域的**添加**设置标签、属性或者更新时间的匹配条件。

下图中条件为查询度量名称为cpu，标签中含有os=Ubuntu16.10的所有时间线。



3. 单击**确定**。

   符合查询条件的数据会显示在**数据管理**页签。

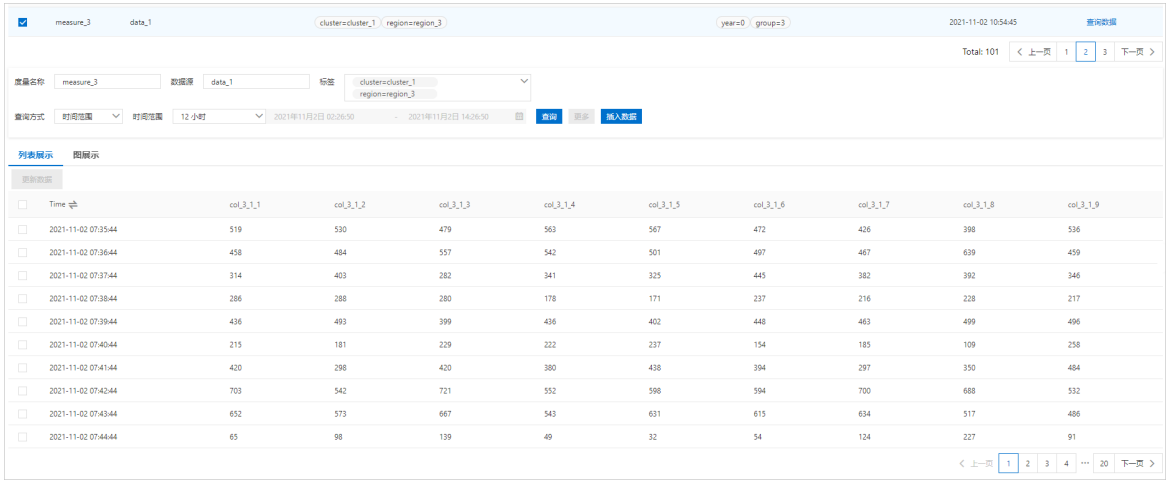
步骤四：查询时序数据

查询某一时间线在指定时间范围内的数据。

1. 在**数据管理**页签，在目标时间线的**操作**列单击**查询数据**。
2. 选择时间范围或者微秒时间戳的查询方式并设置时间，单击**查询**。

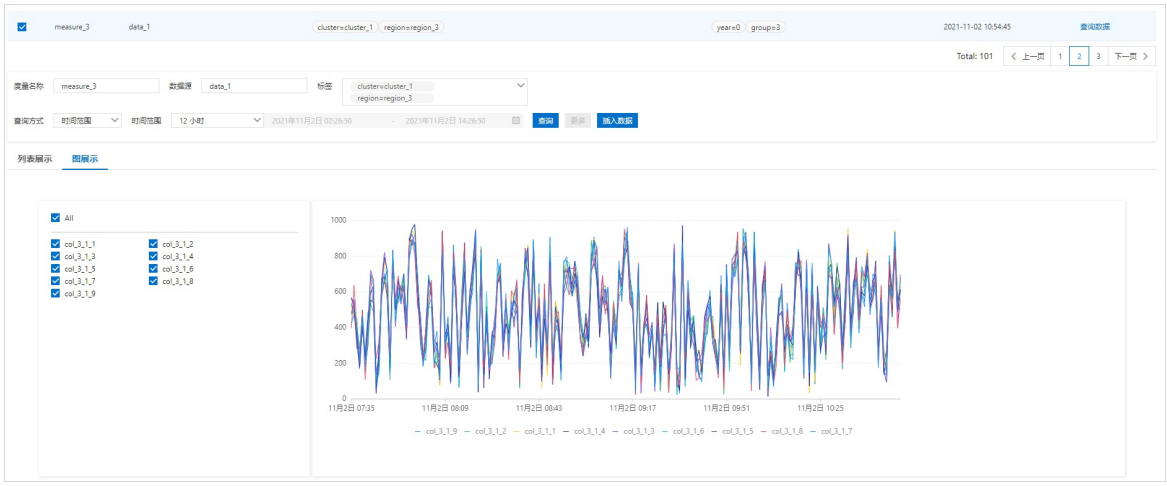
   符合查询条件的数据会显示在**数据管理**页签，查询结果支持按列表或者图的方式展示。

列表展示：



图展示：

> **说明** 图中不同颜色代表不同的数据列，将鼠标移动到数据趋势线上将显示该时间节点上不同数据列的值。您还可以通过取消选中或者选中指定数据列来展示所需数据列。



5.3.2. 使用命令行工具

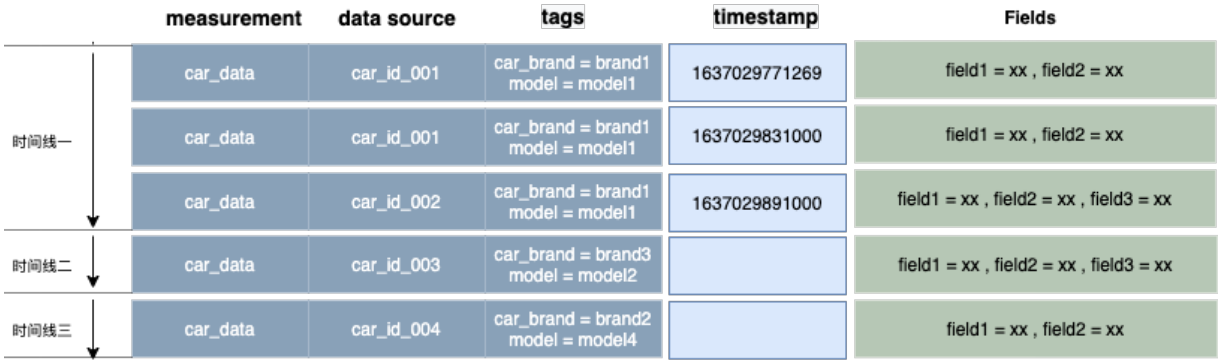
使用命令行工具（Tablestore CLI）创建时序表后，您可以通过CLI命令写入时序数据并对时间线进行检索，以及查询时序数据。您也可以通过SQL语句检索时间线以及查询时序数据。

前提条件

- 已创建时序模型实例。具体操作，请参见创建时序模型实例。
- 已下载命令行工具。具体操作，请参见下载。
- 已启动并配置实例。具体操作，请参见启动并配置。
- 已获取AccessKey。具体操作，请参见获取AccessKey。

样例场景

本文以车联网场景中的一张车辆状态表car_data为例，带您体验时序模型的基本使用方法。一张车辆状态表的时序模型包括了measurement（度量类型）、data source（数据源）、tags（时间线标签）、timestamp（时间戳）和fields（属性列），数据结构模型如下图所示。



时序表操作

1. 执行**create**命令创建一张时序表car_data。

   ```
   create -m timeseries -t car_data
   ```

2. 执行use --ts命令选择操作时序表car_data。

   ```
   use --ts -t car_data
   ```

3. 通过以下任意一种方式导入时序数据。

   - 写入单行时序数据

     执行putts命令写入单行时序数据。以下示例中写入了1条时序数据。

     ```
     putts --k '["car_data","car_0000010", ["brand=brand0","id=car_0000010","model=em3"]]'
     --field '[{"c":"duration","v":121,"isint":true},{"c":"mileage","v":6480,"isint":true}
     ,{"c":"power","v":69,"isint":true},{"c":"speed","v":24,"isint":true},{"c":"temperatur
     e","v":13,"isint":true}]' --time 1636460000000000
     ```

   - 批量导入时序数据

     下载样例数据，执行import_timeseries命令批量导入时序数据。样例数据中共包含了500万条时序数据，您还可以通过import_timeseries -l参数自定义导入行数（导入1000万行内免费使用）。

     以下示例中导入了5万条时序数据。其中yourFilePath表示样例数据压缩包解压后的路径，例如 `D:\\timeseries_demo_data_5000000` 。

     ```
     import_timeseries -i yourFilePath -l 50000
     ```

     导入数据时，日志输出示例如下：

     ```
     Current speed is: 11000 rows/s. Total succeed count 11000, failed count 0.
     Current speed is: 13000 rows/s. Total succeed count 24000, failed count 0.
     Current speed is: 16400 rows/s. Total succeed count 40400, failed count 0.
     Import finished, total count is 50000, failed 0 rows.
     ```

4. 执行qtm命令查询时间线，以下示例中查询所有时间线返回10条。

   ```
   qtm -l 10
   ```

返回结果示例如下：

```
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| measurement | data_source | tags                                            | attribute
s | update_time      |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000005 | ["brand=brand0","id=car_0000005","model=m0"]    | null
| 1637722788684102 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000009 | ["brand=brand2","id=car_0000009","model=em3"] | null
| 1637722790158982 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000000 | ["brand=brand0","id=car_0000000","model=m3"]    | null
| 1637722787172818 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000008 | ["brand=brand0","id=car_0000008","model=m3"]    | null
| 1637722789832880 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000002 | ["brand=brand1","id=car_0000002","model=nm1"] | null
| 1637722787915852 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | null
| 1637722789006974 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000001 | ["brand=brand2","id=car_0000001","model=em2"] | null
| 1637722787260034 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000004 | ["brand=brand0","id=car_0000004","model=m2"]    | null
| 1637722788529313 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000003 | ["brand=brand1","id=car_0000003","model=nm0"] | null
| 1637722788288273 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
| car_data    | car_0000007 | ["brand=brand2","id=car_0000007","model=em2"] | null
| 1637722789315575 |
+-------------+-------------+-------------------------------------------------+----------
--+------------------+
```

5. 执行getts命令查询一条时间线的前5个时间点。

```
getts --k '["car_data","car_0000006", ["brand=brand2","id=car_0000006","model=em2"]]' -
l 5
```

返回结果示例如下：

```
+-------------+-------------+-------------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| measurement | data_source | tags                                            | timestamp
| duration | mileage | power | speed | temperature |
+-------------+-------------+-------------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656000
0000000 | 190      | 1770    | 33    | 54    | 29          |
+-------------+-------------+-------------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656001
0000000 | 554      | 6670    | 42    | 24    | 12          |
+-------------+-------------+-------------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656002
0000000 | 564      | 9750    | 14    | 75    | 22          |
+-------------+-------------+-------------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656003
0000000 | 176      | 7950    | 90    | 24    | 22          |
+-------------+-------------+-------------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
| car_data    | car_0000006 | ["brand=brand2","id=car_0000006","model=em2"] | 163656004
0000000 | 441      | 6280    | 30    | 38    | 31          |
+-------------+-------------+-------------------------------------------------+----------
--------+----------+---------+-------+-------+-------------+
```

使用SQL查询时序数据

创建时序表后，系统会自动为时序表创建两个SQL映射关系，包括时序数据表和时序元数据表。

- 时序数据表：与时序表名称相同，用于查询时序数据。car_data的时序数据表名称为 `car_data` 。
- 时序元数据表：在时序表名称后拼接 `::meta` ，用于查询时间线元数据。car_data的时序元数据表名称为 `car_data::meta` 。

1. 执行sql命令进入SQL命令行模式。

   ```
   sql
   ```

2. 检索时间线。
   - 示例一：查询品牌为“brand0”并且型号为“m3”的车辆，返回前10条。

     ```
     SELECT * FROM `car_data::meta` WHERE _m_name = "car_data" AND tag_value_at(_tags,"bra
     nd") = "brand0" AND tag_value_at(_tags,"model") = "m3" LIMIT 10;
     ```

     返回结果示例如下：

```
+----------+--------------+-------------------------------------------------+-----------
--+-------------------+
| _m_name  | _data_source | _tags                                           | _attribute
s | _meta_update_time |
+----------+--------------+-------------------------------------------------+-----------
--+-------------------+
| car_data | car_0000000  | ["brand=brand0","id=car_0000000","model=m3"] | null
| 1637722787172818  |
+----------+--------------+-------------------------------------------------+-----------
--+-------------------+
| car_data | car_0000008  | ["brand=brand0","id=car_0000008","model=m3"] | null
| 1637722789832880  |
+----------+--------------+-------------------------------------------------+-----------
--+-------------------+
```

- 示例二：统计品牌为“brand2”的车辆总数。

```
SELECT count(*) FROM `car_data::meta` WHERE tag_value_at(_tags,"brand") = "brand2";
```

返回结果示例如下：

```
+----------+
| count(*) |
+----------+
| 4        |
+----------+
```

3. 查询时序数据。
   - 示例一：查询度量名称为“car_data”并且数据源为“car_0000175”车辆，返回power度量的前10个数据点。

```
SELECT _time, _field_name, _long_value as value FROM `car_data` WHERE _m_name = "car_
data" AND _data_source = "car_0000001" AND _field_name = "power" LIMIT 10;
```

返回结果示例如下：

```
+------------------+-------------+-------+
| _time            | _field_name | value |
+------------------+-------------+-------+
| 1636560000000000 | power       | 68    |
+------------------+-------------+-------+
| 1636560010000000 | power       | 41    |
+------------------+-------------+-------+
| 1636560020000000 | power       | 69    |
+------------------+-------------+-------+
| 1636560030000000 | power       | 95    |
+------------------+-------------+-------+
| 1636560040000000 | power       | 27    |
+------------------+-------------+-------+
| 1636560050000000 | power       | 26    |
+------------------+-------------+-------+
| 1636560060000000 | power       | 98    |
+------------------+-------------+-------+
| 1636560070000000 | power       | 82    |
+------------------+-------------+-------+
| 1636560080000000 | power       | 24    |
+------------------+-------------+-------+
| 1636560090000000 | power       | 2     |
+------------------+-------------+-------+
```

- 示例二：查询度量名称为“car_data”并且数据源为“car_000002”的车辆最大行驶速度。

```
SELECT max(_long_value) as speed FROM `car_data` WHERE _m_name = "car_data" AND _data
_source = "car_0000002" AND _field_name = "speed";
```

返回结果示例如下：

```
+-------+
| speed |
+-------+
| 100   |
+-------+
```

- 示例三：对度量名称为“car_data”并且数据源为“car_0000001”的车辆的室温数据按照时间窗口（60s聚合一次）进行聚合，统计每分钟最低室温。

```
SELECT _time DIV 60000000 * 60 as time_sec, min(_long_value) as temperature FROM `car
_data` WHERE _data_source = "car_0000001" AND _field_name = "temperature" GROUP BY ti
me_sec ORDER BY time_sec ASC LIMIT 10;
```

返回结果示例如下：

```
+------------+-------------+
| time_sec   | temperature |
+------------+-------------+
| 1636560000 | 11          |
+------------+-------------+
| 1636560060 | 10          |
+------------+-------------+
| 1636560120 | 11          |
+------------+-------------+
| 1636560180 | 10          |
+------------+-------------+
| 1636560240 | 11          |
+------------+-------------+
| 1636560300 | 12          |
+------------+-------------+
| 1636560360 | 14          |
+------------+-------------+
| 1636560420 | 10          |
+------------+-------------+
| 1636560480 | 15          |
+------------+-------------+
| 1636560540 | 11          |
+------------+-------------+
```

4. 退出SQL模式。

```
exit;
```

5. 退出命令行工具。

```
exit
```

5.4. 使用SDK

通过SDK创建时序表后，您可以写入时序数据并对时间线进行检索，以及查询时序数据。

前提条件

已创建时序模型实例。具体操作，请参见创建时序模型实例。

注意事项

- 时序模型功能将从2022年05月26日正式开始收费。
- 目前支持使用时序模型功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

如果使用过程中遇到问题，请通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

接口

接口	描述
CreateTimeseriesTable	创建一张时序表。

接口	描述
ListTimeseriesTable	获取当前实例下的时序表列表。
DescribeTimeseriesTable	获取一个时序表的信息。
UpdateTimeseriesTable	更新时序表的配置信息。
DeleteTimeseriesTable	删除一个时序表。
PutTimeseriesData	写入时序数据。
GetTimeseriesData	查询某个时间线的数据。
QueryTimeseriesMeta	检索时间线的元数据。
UpdateTimeseriesMeta	更新时间线的元数据。
DeleteTimeseriesMeta	删除时间线的元数据。

使用

您可以使用如下语言的SDK实现时序模型功能。

- Java SDK
- Go SDK

创建时序表

使用CreateTimeseriesTable创建时序表时，需要指定表的配置信息。

- 参数

  时序表的结构信息（timeseriesTableMeta）包括表名（timeseriesTableName）和配置信息（timeseriesTableOptions）的配置，详细参数说明请参见下表。

参数	说明
timeseriesTableName	时序表名。
timeseriesTableOptions	时序表的配置信息，包括如下内容： timeToLive：配置时序表的数据存活时间，单位为秒。如果希望数据永不过期，可以设置为-1。您可以通过UpdateTimeseriesTable接口修改。

- 示例

  创建test_timeseries_table时序表，且该表中数据永不过期。

```
private static void createTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    TimeseriesTableMeta timeseriesTableMeta = new TimeseriesTableMeta(tableName);
    int timeToLive = -1;
    timeseriesTableMeta.setTimeseriesTableOptions(new TimeseriesTableOptions(timeToLive))
;
    CreateTimeseriesTableRequest request = new CreateTimeseriesTableRequest(timeseriesTab
leMeta);
    client.createTimeseriesTable(request);
}
```

写入数据

使用PutTimeseriesData接口，您可以批量写入时序数据。一次PutTimeseriesData调用可以写入多行数据。

- 参数

  一行时序数据（timeseriesRow）包括时间线标识（timeseriesKey）和时间线数据的配置，其中时间线数据包括数据点的时间（timeInUs）和数据点（fields）。详细参数说明请参见下表。

参数	说明
timeseriesKey	时间线标识，包括如下内容： ○ measurementName：时间线的度量名称。 ○ dataSource：数据源信息，可以为空。 ○ tags：时间线的标签信息，为多个字符串的key-value对。
timeInUs	数据点的时间，单位为微秒。
fields	数据点，可以由多个名称（FieldKey）和数据值（FieldValue）对组成。

- 示例

  向test_timeseries_table时序表中批量写入时序数据。

```
private static void putTimeseriesData(TimeseriesClient client) {
    List<TimeseriesRow> rows = new ArrayList<TimeseriesRow>();
    for (int i = 0; i < 10; i++) {
        Map<String, String> tags = new HashMap<String, String>();
        tags.put("region", "hangzhou");
        tags.put("os", "Ubuntu16.04");
        // 通过measurementName、dataSource和tags构建TimeseriesKey。
        TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_" + i, tags);
        // 指定timeseriesKey和timeInUs创建timeseriesRow。
        TimeseriesRow row = new TimeseriesRow(timeseriesKey, System.currentTimeMillis() *
1000 + i);
        // 增加数据值（field）。
        row.addField("cpu_usage", ColumnValue.fromDouble(10.0));
        row.addField("cpu_sys", ColumnValue.fromDouble(5.0));
        rows.add(row);
    }
    String tableName = "test_timeseries_table";
    PutTimeseriesDataRequest putTimeseriesDataRequest = new PutTimeseriesDataRequest(tabl
eName);
    putTimeseriesDataRequest.setRows(rows);
    // 一次写入多行时序数据。
    PutTimeseriesDataResponse putTimeseriesDataResponse = client.putTimeseriesData(putTim
eseriesDataRequest);
    // 检查是否全部成功。
    if (!putTimeseriesDataResponse.isAllSuccess()) {
        for (PutTimeseriesDataResponse.FailedRowResult failedRowResult : putTimeseriesDat
aResponse.getFailedRows()) {
            System.out.println(failedRowResult.getIndex());
            System.out.println(failedRowResult.getError());
        }
    }
}
```

检索时间线

- 参数

  metaQueryCondition表示检索时间线的条件，包括compositeMetaQueryCondition（组合条件）、measurementMetaQueryCondition（度量名称条件）、dataSourceMetaQueryCondition（数据源条件）、tagMetaQueryCondition（标签条件）、attributeMetaQueryCondition（属性条件）和updateTimeMetaQueryCondition（更新时间条件）。详细参数说明请参见下表。

参数	说明
compositeMetaQueryCondition	组合条件，包括如下内容： ○ operator：逻辑运算符，可选AND、OR、NOT。 ○ subConditions：子条件列表，通过operator组成复杂查询条件。
measurementMetaQueryCondition	度量名称条件，包括如下内容： ○ operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 ○ value：要匹配的度量名称值，类型为字符串。

参数	说明
dataSourceMetaQueryCondition	数据源条件，包括如下内容： ○ operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 ○ value：要匹配的数据源值，类型为字符串。
tagMetaQueryCondition	标签条件，包括如下内容： ○ operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 ○ value：要匹配的标签值，类型为字符串。
attributeMetaQueryCondition	时间线元数据的属性条件，包括如下内容： ○ operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 ○ attributeName：属性名称，类型为字符串。 ○ value：属性值，类型为字符串。
updateTimeMetaQueryCondition	时间线元数据的更新时间条件，包括如下内容： ○ operator：关系运算符，可选=、!=、>、>=、<、<=。 ○ timeInUs：时间线元数据更新时间的时间戳，单位为微秒。

- 示例

  查询test_timeseries_table时序表中度量名称为cpu，标签中含有os标签且标签前缀为"Ubuntu"的所有时间线。

```
private static void queryTimeseriesMeta(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    QueryTimeseriesMetaRequest queryTimeseriesMetaRequest = new QueryTimeseriesMetaReques
t(tableName);
    // 查询度量名称为cpu，标签中含有os标签且前缀为"Ubuntu"的所有时间线。即measurement_name="cpu"
and have_prefix(os, "Ubuntu")
    CompositeMetaQueryCondition compositeMetaQueryCondition = new CompositeMetaQueryCondi
tion(MetaQueryCompositeOperator.OP_AND);
    compositeMetaQueryCondition.addSubCondition(new MeasurementMetaQueryCondition(MetaQue
rySingleOperator.OP_EQUAL, "cpu"));
    compositeMetaQueryCondition.addSubCondition(new TagMetaQueryCondition(MetaQuerySingle
Operator.OP_PREFIX, "os", "Ubuntu"));
    queryTimeseriesMetaRequest.setCondition(compositeMetaQueryCondition);
    queryTimeseriesMetaRequest.setGetTotalHits(true);
    QueryTimeseriesMetaResponse queryTimeseriesMetaResponse = client.queryTimeseriesMeta(
queryTimeseriesMetaRequest);
    System.out.println(queryTimeseriesMetaResponse.getTotalHits());
    for (TimeseriesMeta timeseriesMeta : queryTimeseriesMetaResponse.getTimeseriesMetas()
) {
        System.out.println(timeseriesMeta.getTimeseriesKey().getMeasurementName());
        System.out.println(timeseriesMeta.getTimeseriesKey().getDataSource());
        System.out.println(timeseriesMeta.getTimeseriesKey().getTags());
        System.out.println(timeseriesMeta.getAttributes());
        System.out.println(timeseriesMeta.getUpdateTimeInUs());
    }
}
```

查询时序数据

- 参数

参数	说明
timeseriesKey	要查询的时间线，包括如下内容： ○ measurementName：时间线的度量名称。 ○ dataSource：数据源信息，可以为空。 ○ tags：时间线的标签信息，为多个字符串的key-value对。
timeRange	要查询的时间范围，范围为左闭右开区间。包括如下内容： ○ beginTimeInUs：起始时间。 ○ endTimeInUs：结束时间。
backward	是否按照时间倒序读取数据，可用于获取某条时间线的最新数据。取值范围如下： ○ true：按照时间倒序读取。 ○ false（默认）：按照时间正序读取。

参数	说明
fieldsToGet	要获取的数据列列名。如果不指定，则默认获取所有列。 **注意** fieldsToGet中需要指定要获取的每一列的列名和类型。如果类型不匹配，则读取不到对应列的数据。
limit	本次最多返回的行数。 **说明** limit仅限制最多返回的行数，在满足条件行数大于limit时，也可能由于扫描数据量等限制导致返回行数少于limit条，此时可以通过nextToken继续获取后面的行。
nextToken	如果一次查询仅返回了部分符合条件的行，此时response中会包括nextToken，可在下一次请求中指定nextToken用来继续读取数据。

- 示例

  查询test_timeseries_table时序表中满足指定条件的时序数据。

```
private static void getTimeseriesData(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    GetTimeseriesDataRequest getTimeseriesDataRequest = new GetTimeseriesDataRequest(tabl
eName);
    Map<String, String> tags = new HashMap<String, String>();
    tags.put("region", "hangzhou");
    tags.put("os", "Ubuntu16.04");
    // 通过measurementName、dataSource和tags构建TimeseriesKey。
    TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_0", tags);
    getTimeseriesDataRequest.setTimeseriesKey(timeseriesKey);
    // 指定时间范围。
    getTimeseriesDataRequest.setTimeRange(0, (System.currentTimeMillis() + 60 * 1000) * 1
000);
    // 限制返回行数。
    getTimeseriesDataRequest.setLimit(10);
    // 设置是否倒序读取数据，可不设置，默认值为false。如果设置为true，则倒序读取数据。
    getTimeseriesDataRequest.setBackward(false);
    // 设置获取部分数据列，可不设置，默认获取全部数据列。
    getTimeseriesDataRequest.addFieldToGet("string_1", ColumnType.STRING);
    getTimeseriesDataRequest.addFieldToGet("long_1", ColumnType.INTEGER);
    GetTimeseriesDataResponse getTimeseriesDataResponse = client.getTimeseriesData(getTim
eseriesDataRequest);
    System.out.println(getTimeseriesDataResponse.getRows().size());
    if (getTimeseriesDataResponse.getNextToken() != null) {
        // 如果nextToken不为空，可以发起下一次请求。
        getTimeseriesDataRequest.setNextToken(getTimeseriesDataResponse.getNextToken());
        getTimeseriesDataResponse = client.getTimeseriesData(getTimeseriesDataRequest);
        System.out.println(getTimeseriesDataResponse.getRows().size());
    }
}
```

5.5. 使用SQL查询时序数据

创建时序表并建立SQL映射关系后，您可以通过控制台、SDK等不同方式使用SQL查询时序数据。

时序表的SQL映射关系

根据每个时间线每个时刻对应一个值或者多个值，时序模型分为“单值模型”和“多值模型”。对于同一个时序表，您可以建立三种SQL映射关系用于数据查询，详细说明请参见下表。

映射关系类型	描述	创建方式	在SQL中的表名
单值模型映射关系	以单值模型查询时序数据。	创建时序表后，系统自动建立SQL映射关系。	与时序表名相同
多值模型映射关系	以多值模型查询时序数据。	创建时序表后，由用户手动建立SQL映射关系。	在时序表名后拼接 `::后缀` ，即 `时序表名::后缀` ，其中 `后缀` 由用户在创建时自定义
时间线元数据映射关系	查询时间线元数据。	创建时序表后，系统自动建立SQL映射关系。	在时序表名称后拼接 `::meta` ，即 `时序表名::meta`

单值模型映射关系

时序表创建后，系统会自动创建单值模型映射关系。在SQL中表名与时序表名称相同，用于以单值模型查询时序数据。

表结构请参见下表。

字段名称	类型	描述
_m_name	VARCHAR	度量名称。
_data_source	VARCHAR	数据源。
_tags	VARCHAR	时间线标签，以数组表示。多个标签的格式为["tagKey1=tagValue1","tagKey2=tagValue2"] 。 您可以使用tag_value_at函数提取某个标签的值。
_time	BIGINT	数据点的时间戳，单位微秒。
_field_name	VARCHAR	数据列名。
_long_value	BIGINT	整型的数据值。如果该数据列为非整型，则值为NULL。
_double_value	DOUBLE	浮点数类型的数据值。如果该数据列为非浮点数，则值为NULL。
_bool_value	BOOL	布尔类型的数据值。如果该数据列为非布尔值，则值为NULL。

字段名称	类型	描述
_string_value	VARCHAR	字符串类型的数据值。如果该数据列为非字符串，则值为NULL。
_binary_value	MEDIUMBLOB	二进制类型的数据值。如果该数据列为非二进制，则值为NULL。
_attributes	VARCHAR	时间线属性，格式与标签相同。
_meta_update_time	BIGINT	时间线的元数据更新时间。 当用户更新时间线属性时，系统会自动更新时间线元数据更新时间。此外，如果该时间线的数据持续写入，系统也会定时更新该时间，可用于判断一条时间线是否活跃。

多值模型映射关系

当以多值模型查询时序数据时，您需要执行CREATE TABLE语句来创建多值模型映射关系。在SQL中表名为时序表名称后拼接`::后缀`，其中`后缀`由用户自定义。一个时序表支持创建多个多值模型映射关系。

创建多值模型映射关系时，您需要在SQL语句中指定多值模型的映射关系名称、包含的数据列的列名和类型等。具体操作，请参见创建多值模型映射关系。

表结构请参见下表。

> **说明** 如果要通过多值模型映射关系读取时间线元数据的属性列（_attributes）或者元数据最近更新时间列（_meta_update_time），您需要将这两列添加到多值模型映射关系中，系统会自动填充这两个元数据列的内容。

字段名称	类型	描述
_m_name	VARCHAR	度量名称。
_data_source	VARCHAR	数据源。
_tags	VARCHAR	时间线标签，以数组表示。多个标签的格式为["tagKey1=tagValue1","tagKey2=tagValue2"] 。您可以使用tag_value_at函数提取某个标签的值。
_time	BIGINT	数据点的时间戳，单位微秒。
自定义数据列名	SQL数据类型	自定义的数据列，支持添加多个。 如果指定的列名或者类型与表中实际写入的列名或者类型不符，则该列读取结果为null。
_attributes（可选）	MEDIUMTEXT	时间线属性，格式与标签相同。

字段名称	类型	描述
_meta_update_time（可选）	BIGINT	时间线的元数据更新时间。 当用户更新时间线属性时，系统会自动更新时间线元数据更新时间。此外，如果该时间线的数据持续写入，系统也会定时更新该时间，可用于判断一条时间线是否活跃。

时间线元数据映射关系

时序表创建后，系统会自动创建时间线元数据映射关系。在SQL中表名为时序表名称后拼接 `::meta` ，用于查询时间线元数据。假如时序表名称为timeseries_table，则时序元数据表的名称为 `timeseries_table::meta` 。

表结构请参见下表。

字段名称	类型	描述
_m_name	VARCHAR	度量名称。
_data_source	VARCHAR	数据源。
_tags	VARCHAR	时间线标签。
_attributes	VARCHAR	时间线属性。
_meta_update_time	BIGINT	时间线的元数据更新时间。 当用户更新时间线属性时，系统会自动更新时间线元数据更新时间。此外，如果该时间线的数据持续写入，系统也会定时更新该时间，可用于判断一条时间线是否活跃。

SQL语法

创建多值模型映射关系

通过CREATE TABLE语句创建多值模型映射关系。

- SQL语法

```
CREATE TABLE `timeseries_table::user_mapping_name` (
  `_m_name` VARCHAR(1024),
  `_data_source` VARCHAR(1024),
  `_tags` VARCHAR(1024),
  `_time` BIGINT(20),
  `user_column_name1 ` data_type,
  ......
  `user_column_namen ` data_type,
  PRIMARY KEY(`_m_name`,`_data_source`,`_tags`,`_time`)
);
```

详细参数说明请参见多值模型映射关系的表结构信息。

- SQL示例

假设测量的属性同时包含了cpu、memory、disktop三种度量，此处以创建多值类型映射表 `timeseries_table::muti_model` 为例介绍。SQL示例如下：

```
CREATE TABLE `timeseries_table::muti_model` (
  `_m_name` VARCHAR(1024),
  `_data_source` VARCHAR(1024),
  `_tags` VARCHAR(1024),
  `_time` BIGINT(20),
  `cpu` DOUBLE(10),
  `memory` DOUBLE(10),
  `disktop` DOUBLE(10),
  PRIMARY KEY(`_m_name`,`_data_source`,`_tags`,`_time`)
);
```

查询数据

通过SELECT语句执行时序数据查询。更多信息，请参见查询数据。

表格存储还提供tag_value_at扩展函数用于时间线标签（_tags）中的某个标签（tag）的值以及提取时间线属性（_attributes）中的某个属性值。

假设_tag为["host=abc","region=hangzhou"]，则您可以使用tag_value_at(_tags, "host")提取host标签的值，即"abc"。SQL语句示例如下：

```
SELECT tag_value_at(_tags, "host") as host FROM timeseries_table LIMIT 1;
```

SQL示例

查询时间线

创建时序表后，系统会自动创建时间线元数据映射表，您可以使用时间线元数据映射表查询时间线。

此处以时序表名称为timeseries_table，时间线元数据映射表名称为 `timeseries_table::meta` ，度量类型为basic_metric为例介绍。

- 查询时序元数据表中basic_metric度量类型下的时间线。

```
SELECT * FROM  `timeseries_table::meta` WHERE _m_name = "basic_metric" LIMIT 100;
```

- 查询时序元数据表中满足多个标签条件（host=host001, region=hangzhou）的时间线。

```
SELECT * FROM `timeseries_table::meta` WHERE _m_name = "basic_metric" AND tag_value_at(_t
ags, "host") = "host001"
 AND tag_value_at(_tags, "region") = "hangzhou" LIMIT 100;
```

使用单值模型映射表查询时序数据

创建时序表后，系统会自动创建同名的单值模型映射表，您可以使用单值模型映射表查询时序数据。

此处以时序表名称为timeseries_table，单值模型映射表名称为 `timeseries_table` ，度量类型为basic_metric为例介绍。

- 查询时序数据表中basic_metric度量类型的数据。

```
SELECT * FROM timeseries_table WHERE _m_name = "basic_metric" LIMIT 10;
```

- 查询时序数据表中满足单个标签条件（host=host001）的时间线的数据。

```
SELECT * FROM timeseries_table WHERE _m_name = "basic_metric" AND tag_value_at(_tags, "ho
st") = "host001"
 AND _time > (UNIX_TIMESTAMP() - 900) * 1000000 LIMIT 10;
```

- 查询时序数据表中满足多个标签条件（host=host001, region=hangzhou）的时间线的数据。

```
SELECT * FROM timeseries_table WHERE _m_name = "basic_metric" AND tag_value_at(_tags, "ho
st") = "host001" AND tag_value_at(_tags, "region") = "hangzhou"
 AND _time > (UNIX_TIMESTAMP() - 900) * 1000000 LIMIT 10;
```

使用多值模型映射表查询时序数据

创建时序表后，通过手动创建多值模型映射关系，您可以使用多值模型映射表查询时序数据。关于创建多值模型映射关系的具体操作，请参见创建多值模型映射关系。

此处以时序表名称为timeseries_table，多值模型映射表名称为 `timeseries_table::muti_model` ，测量的属性同时包含了cpu、memory、disktop三种度量为例介绍。

- 查询多值模型映射表中数据源标识为host_01的数据。（假设_data_source中保存了host_id）

```
SELECT * FROM `timeseries_table::muti_model` WHERE _data_source = "host_01" LIMIT 10;
```

- 查询多值模型映射表中cpu大于20.0的所有度量信息。

```
SELECT cpu,memory,disktop FROM `timeseries_table::muti_model` WHERE cpu > 20.0 LIMIT 10;
```

- 计算多值模型映射表中满足标签条件（region=hangzhou）的主机在日期为2022-01-01内的平均cpu和最大disktop。

```
SELECT avg(cpu) as avg_cpu,max(disktop) as max_disktop FROM `timeseries_table::muti_mode`
WHERE tag_value_at(_tags,"region") = "hangzhou"
 AND _time > 1640966400000000 AND _time < 1641052799000000 GROUP BY _data_source;
```

使用方式

您可以通过以下方式使用SQL查询时序数据。查询时序数据时，请根据实际需要对应映射表进行操作。

- 使用控制台。具体操作，请参见使用控制台。
- 使用SDK。具体操作，请参见使用SDK。
- 使用JDBC
  - 使用JDBC直连。具体操作，请参见JDBC连接表格存储。
  - 通过Hibernate使用。具体操作，请参见通过Hibernate使用。
  - 通过MyBatis使用。具体操作，请参见通过MyBatis使用。
- 使用Go语言驱动。具体操作，请参见使用Go语言驱动。
- 使用命令行工具。具体操作，请参见使用命令行工具。

6.Grid模型

6.1. 模型介绍

Grid模型（网格模型）是表格存储针对多维网格数据设计的模型。表格存储的Grid模型可以帮助您方便地实现多维网格数据的存储、查询和管理。

背景

什么是多维网格数据

多维网格数据是一种科学大数据，在地球科学领域（气象、海洋、地质、地形等）应用非常广泛，且数据规模也越来越大。多维网格数据一般包含以下五个维度。

- 物理量（或者称为要素，例如温度、湿度、风向、风速等）
- 时间（例如气象中的预报时效，未来3小时、6小时、9小时等）
- 高度
- 经度
- 纬度

多维网格数据的挑战

- 挑战一：数据规模大

  假设一个三维格点空间包含10个不同高度的平面，每个平面为一个2880 x 570的格点，每个格点保存一个4字节数据，那么这三维的数据量为2880 x 570 x 4 x 10，大约64MB。再加上物理量和时间维度，一个数据集的规模可以在几百MB到几GB的规模，而这样的数据集是每天不断产生，所以总数据量可以到百TB以上规模。

- 挑战二：查询种类丰富、延迟要求高

  相关的科学工作者会有快速浏览数据的需求，例如对于气象预报员会快速的浏览各种相关的数据来进行气象预报，于是对这些数据有着在线查询的需求。在对数据进行查询时，因为一个数据集数据较多，一般不会一次全部查出，而是会按照几种不同的方式来查看其中一部分数据，例如：

  - 查询某个经纬度平面。
  - 查询某个经纬度区域在不同时间范围内的值。
  - 查询某个经纬度区域在不同时间不同高度范围内的值。

简介

- 优势
  - 数据存储量无上限，解决了海量格点数据的规模问题。
  - 根据多维格点数据的特点，对数据进行了恰当的切分，大大提升了通过各种不同维度条件来查询数据的性能，解决了从海量格点数据进行快速检索的需求。
  - 利用了表格存储的多元索引，增加了数据集的元数据管理功能，可以通过多种组合条件筛选数据集，解决了海量格点数据集的管理问题。

- 元素

  在Grid模型设计中，一个五维网格数据为一个网格的数据集（GridDataSet）。按照维度顺序，五维分别为：

维度	描述
variable	变量，例如如各种物理量
time	时间维度
z	z轴，一般表示空间高度
x	x轴，一般表示经度或纬度
y	y轴，一般表示经度或纬度

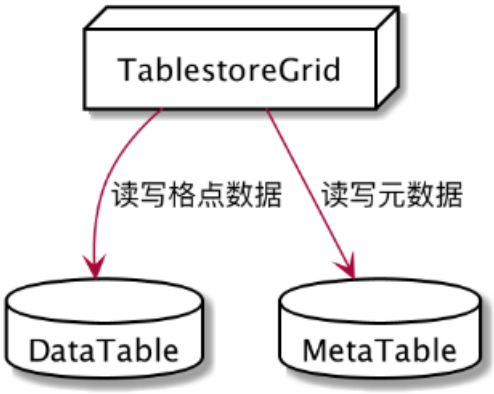
一个GridDataSet除了包含五维数据，还包含描述这些数据的元数据，例如各个维度的长度等，此外还包含GridDataSetId以及用户自定义的一些属性。

名称	说明
GridDataSetId	唯一标记这个GridDataSet的ID。
Attributes	自定义属性信息，例如该数据的产生时间、数据来源、预报类型等等。 您可以自定义属性，也可以给某些属性建立索引，建立索引后就可以通过各种组合条件来查询符合条件的数据集。

- 数据存储方案

  表格存储设计了两张表分别存储数据集的meta和data。

  - meta表示这个数据集的元数据，例如GridDataSetId、各维度长度、自定义属性等。
  - data表示这个数据集里实际的网格数据。data相比meta在数据大小上要大很多。



实现

具体参见基于TableStore的海量气象格点数据解决方案。

6.2. 快速入门

本文主要为您介绍如何通过Java示例代码快速使用Grid模型。

操作步骤

1. 登录表格存储控制台，并创建表格存储实例。详情参见创建实例。
2. 在项目中安装Grid模型SDK，Maven依赖如下：

```
<dependency>
 <groupId>com.aliyun.tablestore</groupId>
 <artifactId>tablestore-grid</artifactId>
 <version>1.0.0</version>
</dependency>
```

3. 通过示例代码快速使用Grid模型。

6.3. 基础操作

6.3.1. 概述

Grid（网格模型）是表格存储针对多维网格数据设计的模型，可以帮助您方便地实现多维网格数据的存储、查询和管理。

您可以通过以下操作对多维网格数据进行写入和查询。

- 初始化
- 写入数据
- 查询数据

6.3.2. 初始化

TablestoreGrid实例是Grid模型的客户端，使用TablestoreGrid实例可以进行创建表、删除表、创建索引表、读写数据等操作。

操作步骤

1. 安装Grid模型Java SDK。

   在Maven项目中使用Grid模型的Java SDK，只需在pom.xml中加入如下依赖。

```
<dependency>
 <groupId>com.aliyun.tablestore</groupId>
 <artifactId>tablestore-grid</artifactId>
 <version>1.0.0</version>
</dependency>
```

2. 创建TablestoreGrid实例。

   TableStoreGrid实例是Grid模型的客户端，用于执行Grid模型的各种操作。

   - 示例代码

```
private TableStoreGrid createTableStoreGrid() {
    TableStoreGridConfig config = new TableStoreGridConfig();
    config.setTableStoreEndpoint("TABLESTORE_ENDPOINT");
    config.setAccessId("ACCESS_ID");
    config.setAccessKey("ACCESS_KEY");
    config.setTableStoreInstance("TABLESTORE_INSTANCE_NAME");
    config.setDataTableName("GRID_DATA_TABLE");
    config.setMetaTableName("GRID_META_TABLE");
    TableStoreGrid tableStoreGrid = new TableStoreGrid(config);
    return tableStoreGrid;
}
```

- 参数说明

参数	说明
TABLESTORE_ENDPOINT	表格存储实例的服务地址（Endpoint）。更多信息，请参见服务地址。
ACCESS_ID和ACCESS_KEY	阿里云账号或者RAM用户的AccessKey ID和AccessKey Secret。具体操作，请参见如何获取AccessKey。
TABLESTORE_INSTANCE_NAME	表格存储实例名。
GRID_DATA_TABLE	存放网格数据的表名，表名建议具有实际业务意义。 说明 此步仅配置数据表的表名，不会创建该数据表。
GRID_META_TABLE	存放网格元数据的表名，表名建议具有实际业务意义。 说明 此步仅配置数据表的表名，不会创建该数据表。

3. 创建表。

   上个步骤中配置了数据表和元数据表的表名，实际表还未创建，您需要通过createStore接口创建表，示例代码如下：

```
private void createTable(TableStoreGrid grid) throws Exception {
    grid.createStore();
}
```

4. 创建元数据索引。

   元数据索引用于对网格元数据进行检索，索引信息中需要包含需要被检索的元数据列等信息。

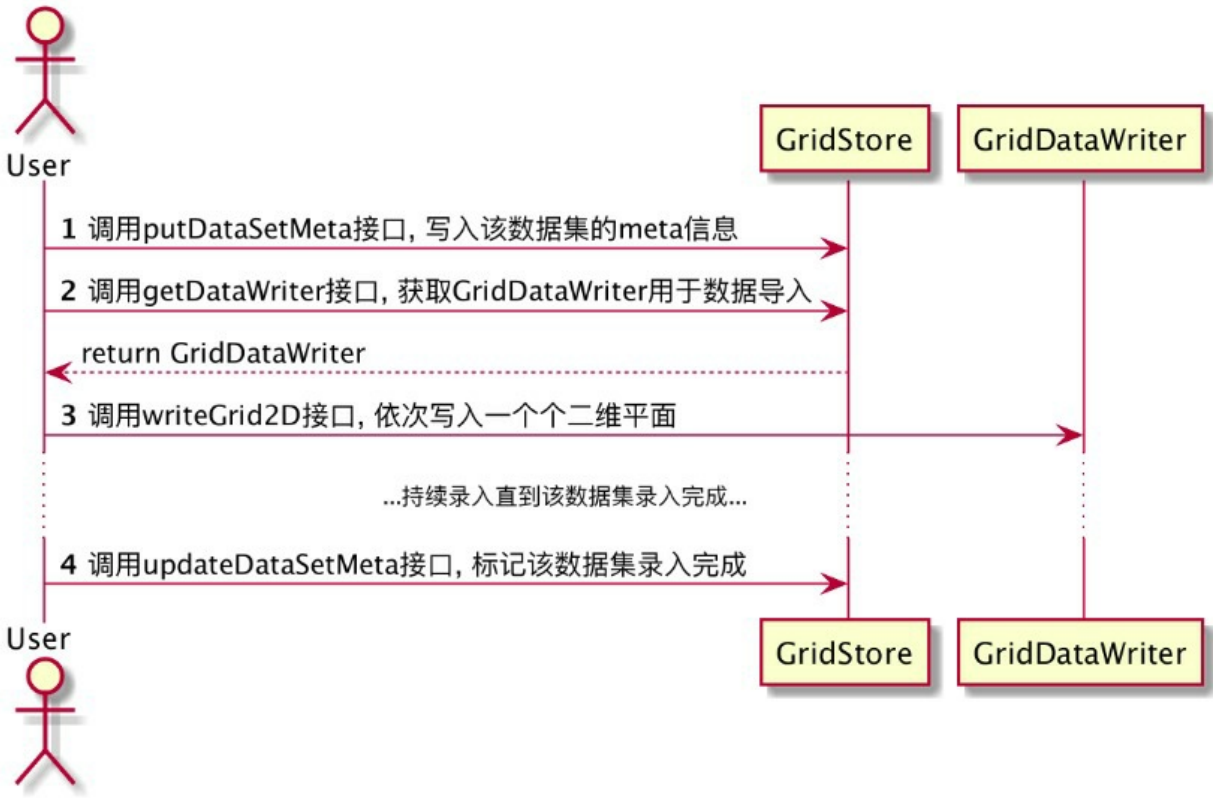
   以下示例中对status、tag1、tag2、create_time这4列创建了索引，您可以自行设计元数据中需要包含的列，以及需要创建索引的列。

```
private void createIndex(TableStoreGrid grid) throws Exception {
        IndexSchema indexSchema = new IndexSchema();
        indexSchema.setFieldSchemas(Arrays.asList(
                new FieldSchema("status", FieldType.KEYWORD).setIndex(true).setEnableSo
rtAndAgg(true),
                new FieldSchema("tag1", FieldType.KEYWORD).setIndex(true).setEnableSort
AndAgg(true),
                new FieldSchema("tag2", FieldType.KEYWORD).setIndex(true).setEnableSort
AndAgg(true),
                new FieldSchema("create_time", FieldType.LONG).setIndex(true).setEnable
SortAndAgg(true)
        ));
        grid.createMetaIndex("GRID_META_INDEX_NAME", indexSchema);
    }
```

6.3.3. 写入数据

本文主要为您介绍如何在Grid模型的客户端写入多维网格数据。

写入流程



如上图所示，网格数据录入流程可以分为三部分：

1. 通过putDataSetMeta接口写入数据集的元数据信息。
2. 通过GridDataWriter录入整个数据集的数据。
3. 通过updateDataSetMeta接口更新数据集的元数据信息，标记数据已经录入完成。

示例

读取一个NetCDF（气象格点数据常用的格式）文件，然后将其中的数据通过GridDataWriter录入到表格存储中。通过GridDataWriter每次写入时，只能写入一个二维平面，所以我们需要在外层进行3层循环，分别枚举变量维、时间维、高度维的值，然后读取对应的二维平面的数据进行录入。

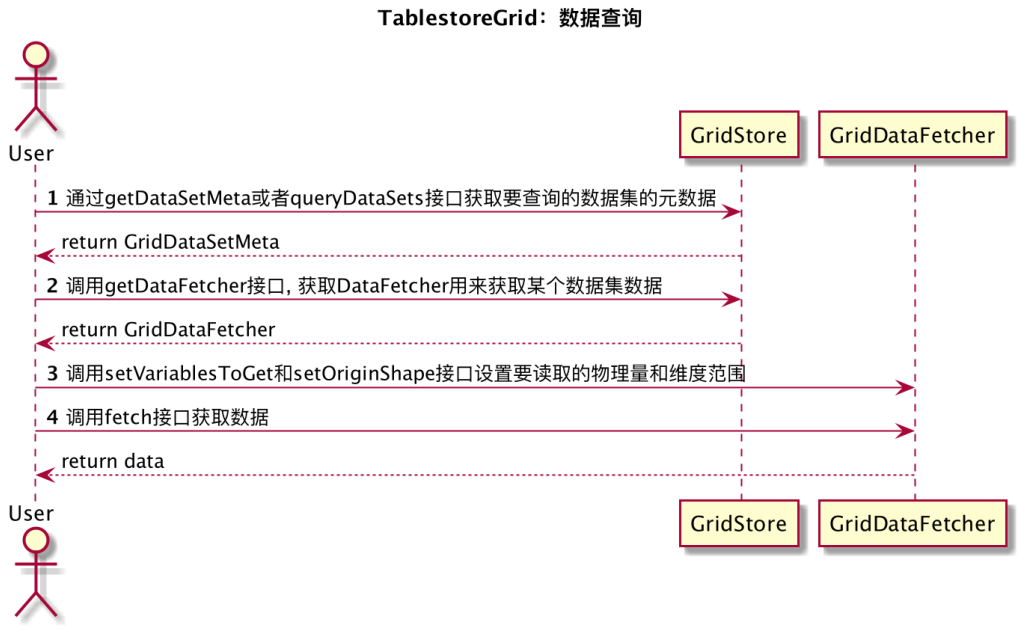
示例代码：

```
    public void importFromNcFile(TableStoreGrid grid, GridDataSetMeta meta, String ncFileNa
me) throws Exception {
        GridDataWriter writer = grid.getDataWriter(meta);
        NetcdfFile ncFile = NetcdfFile.open(ncFileName);
        List<Variable> variables = ncFile.getVariables();
        for (Variable variable : variables) {
            if (meta.getVariables().contains(variable.getShortName())) {
                for (int t = 0; t < meta.gettSize(); t++) {
                    for (int z = 0; z < meta.getzSize(); z++) {
                        Array array = variable.read(new int[]{t, z, 0, 0}, new int[]{1, 1,
meta.getxSize(), meta.getySize()});
                        Grid2D grid2D = new Grid2D(array.getDataAsByteBuffer(), variable.ge
tDataType(),
                                new int[] {0, 0}, new int[] {meta.getxSize(), meta.getySize
()});
                        writer.writeGrid2D(variable.getShortName(), t, z, grid2D);
                    }
                }
            }
        }
    }
```

6.3.4. 查询数据

本文主要为您介绍如何在Grid模型的客户端查询多维网格数据。

数据查询

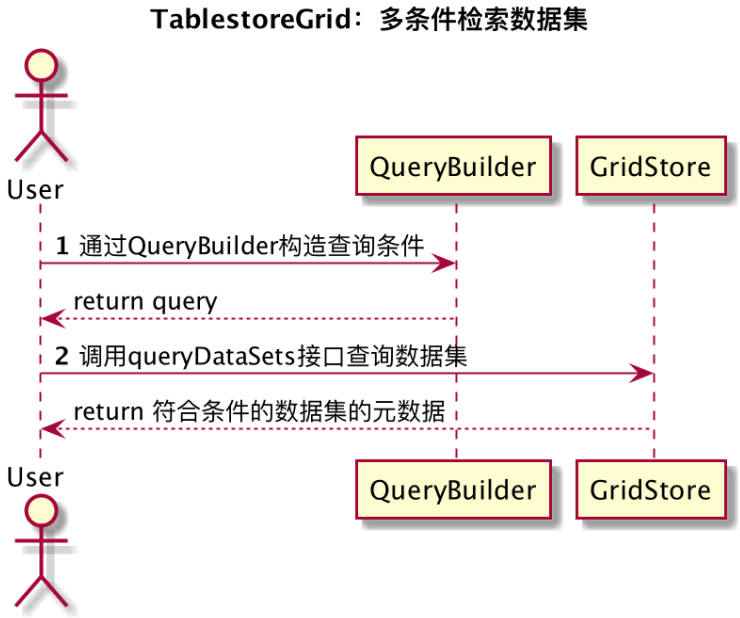


GridDataFetcher支持对五维数据进行任意维度的查询。第一维是变量维，通过setVariablesToGet接口设置要读取哪些变量，其余四维通过设置起始点（origin）和读取的大小（shape）就可以实现任意维度读取。

示例代码：

```
    public Array queryByTableStore(TableStoreGrid grid, String dataSetId, String variable,
int[] origin, int[] shape) throws Exception {
        GridDataFetcher fetcher = grid.getDataFetcher(grid.getDataSetMeta(dataSetId));
        fetcher.setVariablesToGet(Arrays.asList(variable));
        fetcher.setOriginShape(origin, shape);
        Grid4D grid4D = fetcher.fetch().getVariable(variable);
        return grid4D.toArray();
    }
```

多条件检索数据集



对元数据表建立多元索引后，您可以通过各种组合条件来进行数据集检索，查询出符合条件的数据集。多元索引是气象管理系统实现的主要组件。

假设我们要查询已经完成入库的、来源为ECMWF（欧洲中期天气预报中心）或者NMC（全国气象中心）的气象预报，并按照以下条件进行查询，查询结果按照创建时间从新到老排序。

- 创建时间为最近一天
- 精度为1km

示例代码

- 查询条件

```
(status == DONE)
and (create_time > System.currentTimeMillis - 86400000)
and (source == "ECMWF" or source == "NMC") and (accuracy == "1km")
```

- 执行代码

```
public QueryGridDataSetResult queryDataSet(TableStoreGrid grid) throws Exception {
    QueryGridDataSetResult result = grid.queryDataSets(
            ExampleConfig.GRID_META_INDEX_NAME,
            QueryBuilder.and()
                    .equal("status", "DONE")
                    .greaterThan("create_time", System.currentTimeMillis() - 86400000)
                    .equal("accuracy", "1km")
                    .query(QueryBuilder.or()
                            .equal("source", "ECMWF")
                            .equal("source", "NMC")
                            .build())
                    .build(),
            new QueryParams(0, 10, new Sort(Arrays.<Sort.Sorter>asList(new FieldSort("cre
ate_time", SortOrder.DESC)))));
    return result;
}
```

7.多元索引
7.1. 简介

通过使用多元索引（Search Index）的多种高效的索引结构，可以解决大数据的复杂查询难题。

解决什么问题

表格存储的数据表是一种典型的分布式NoSQL数据结构，可以高效地支持大规模数据的存储和读写场景，例如监控数据、日志数据等场景。

为了满足用户的按照非主键列查询，多列的自由组合查询等复杂查询需求，表格存储在支持单行读、范围读等主键查询的同时，创新性的推出了多元索引。

多元索引基于倒排索引和列式存储，可以解决大数据的复杂查询难题，包括非主键列查询、全文检索、前缀查询、模糊查询、多字段自由组合查询、嵌套查询、地理位置查询和统计聚合（max、min、count、sum、avg、distinct_count、group_by）等功能。

索引区别

表格存储在支持数据表的主键查询的基础上，还提供了二级索引（Secondary Index）和多元索引两种加速查询的索引结构。下表展示了三种索引的区别。

索引类型	原理	场景
数据表主键	数据表类似于一个巨大的Map，它的查询能力也就类似于Map，只能通过主键查询。	适用于可以确定完整主键（Key）或主键前缀（Key prefix）的场景。
二级索引	通过创建一张或多张索引表，使用索引表的主键列查询，相当于把数据表的主键查询能力扩展到了不同的列。	适用于能提前确定待查询的列，待查询列数量较少，且可以确定完整主键或主键前缀的场景。
多元索引	使用了倒排索引、BKD树、列存等结构，具备丰富的查询能力。	适用于除数据表主键和二级索引之外的其他所有查询和分析场景： • 非主键列的条件查询 • 任意列的自由组合查询 • And、Or、Not等关系查询 • 全文检索 • 地理位置查询 • 前缀查询 • 模糊查询 • 嵌套结构查询 • Null值查询 • 统计聚合（min、max、sum、avg、count、distinct_count和group_by）

多元索引的使用方式不同于MySQL等传统数据库的索引使用方式，无最左匹配原则的限制，使用时非常灵活。一般情况下一张表只需要创建一个多元索引即可。例如有一个学生表，包括的列有姓名、学号、性别、年级、班级、家庭住址等，创建多元索引时，将这些列添加到同一个多元索引中即可。使用多元索引时，可以指定任意条件的组合查询，例如“姓名等于张三且年级为三年级的学生”、“家庭住址在附近1公里内且性别为男的学生”、“找出三年级二班住在某小区的学生”等。

> **说明** 关于索引对比的更多信息，请参见TableStore索引功能详解。

接口

多元索引目前提供通用查询接口（Search）和数据导出接口（ParallelScan）。

两种接口的功能大部分相同，但是ParallelScan接口为了提高某些方面的性能和吞吐能力舍弃了部分功能，详细信息请参见下表。

接口	说明
Search	全功能查询接口，支持多元索引的所有功能点。 • 查询功能：非主键列查询、全文检索、前缀查询、模糊查询、多字段自由组合查询、嵌套查询、地理位置查询 • 折叠（去重） • 排序 • 统计聚合 • 数据总行数
ComputeSplits+ParallelScan	多并发数据导出接口，支持多元索引中的查询功能，不支持排序、统计聚合等分析功能。 相对于Search接口，ParallelScan可以提供更好的性能，单并发时性能（吞吐能力）是Search接口的5倍。 • 查询功能：非主键列查询、全文检索、前缀查询、模糊查询、多字段自由组合查询、嵌套查询、地理位置查询 • 单请求支持多并发查询

注意事项

> **注意** 使用多元索引时，无需为数据表设置预定义列。

- 索引同步

  用户为数据表创建了多元索引后，当在数据表中写入数据时，数据会先写入数据表中，数据写成功后会立即返回用户写成功，同时另一个异步线程会从数据表中读取写入的数据然后写入到多元索引，采用异步方式创建多元索引不会降低表格存储的写入能力。

  目前多元索引的延迟大部分在3秒以内，通过表格存储控制台可以实时查看多元索引创建的延迟情况。

- 数据生命周期（TTL）
  - 如果数据表无UpdateRow更新写入操作，则您可以使用多元索引TTL。更多信息，请参见多元索引生命周期。

- 当只需要保留一段时间内的数据且时间字段不需要更新时，可以通过按时间分表的方法实现数据生命周期功能。

  按时间分表的原理、原则和优点如下：

  - 原理：按照固定时间，例如“日”、“周”、“月”或者“年”分表，并为每个表建立一个多元索引，根据需要保留所需时间的数据表。

    例如当数据需要保留6个月时，可以将每个月的数据保存在一张数据表中，例如table_1、table_2、table_3、table_4、table_5、table_6，并为每个数据表创建一个多元索引，每个数据表和多元索引中只会保存一个月的数据，只需要每个月把6个月前的数据表删除即可。

    当使用多元索引查询数据时，如果时间范围在某个表中，只需要查询对应表；如果时间范围在多个表中，需要对涉及的数据表均查询一次，再将查询结果合并。
  - 原则：单表（单索引）大小不超过500亿行，当单表（单索引）大小不超过200亿行时，多元索引的查询性能最好。
  - 优点：
    - 通过保留数据表的个数调节数据存储时长。
    - 查询性能和数据量成正比，分表后每个表的数据大小有上限，查询性能更好，避免查询延迟太大或者超时。

- 数据多版本

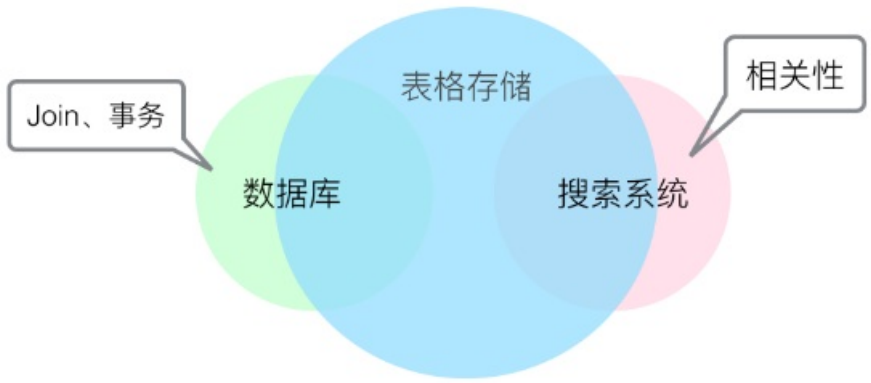
  多元索引不支持数据多版本，即不能对设置了数据多版本的数据表创建多元索引。

  当在单版本中每次写入数据时自定义了timestamp，且先写入版本号较大的数据，后写入版本号较小的数据，此时先写入的版本号较大的数据可能会被后写入的版本号较小的数据覆盖。

功能

多元索引可以解决大数据中复杂的查询问题，同时数据库、搜索引擎等其他系统也可以解决数据的查询问题。表格存储与数据库及搜索引擎等系统的主要区别如下。

除了Join、事务和相关性外，表格存储能覆盖数据库和搜索系统中的其他功能，同时具备数据库的数据高可靠性和搜索系统的高级查询能力，可以替换常见的 `数据库 + 搜索系统组合架构` 方式。如果您的使用场景中不需要Join、事务和相关性，您可以选择使用表格存储多元索引。



计费

更多信息，请参见多元索引计量计费。

7.2. 功能

本文介绍多元索引的核心功能以及部分SQL匹配功能。

核心功能

- 非主键列的查询

  仅使用主键列或主键列前缀查询，无法满足一些查询场景的需求。使用多元索引，您可以使用非主键列进行查询，您仅需要对要查询的列（Column）建立多元索引，即可通过该列的值查询数据。

- 多字段自由组合查询

  多元索引的多字段自由组合查询功能适用于订单场景。在订单场景中，表的字段数可能多达几十个，在创建表时很难完全确定需要查询字段的组合方式。即使在确定需要查询的字段的组合方式的情况下，组合方式会多达上百个，如果使用关系型数据库中可能需要创建上百个索引。同时，如果某种组合方式提前没预想到，没创建，则无法查询。

  使用表格存储，您只需要建立一个多元索引，在索引中包括可能需要查询的字段名，那么查询的时候可以随意自由组合这些字段进行查询。同时多元索引还支持多种关系，例如And、Or和Not等。

- 地理位置查询

  随着移动设备的普及，地理位置信息的价值越来越大，越来越多的应用中都加了地理位置信息，例如朋友圈、微博、外卖、运动和车联网等。这些应用中的信息中含有地理位置数据，那么就需要相匹配的查询能力。

  表格存储多元索引提供了以下地理位置查询功能。

  - Near：以一个点为原点，查询指定附近距离圈内的点，例如朋友圈中附近的人。
  - Within：指定一个矩形框或多边形框，查询出该框内的点。

  基于上述功能，如果应用中需要地理位置相关查询，使用表格存储可以一站式解决，不再需要借助其他数据库或搜索系统。

- 全文检索

  表格存储多元索引提供了分词能力。分词可以支持全文检索。但是表格存储和搜索引擎不一样的地方是，只提供最基础的BM25相关性，不提供自定义相关性能力，所以如果有相关性的搜索需求，还是最好使用搜索系统，否则可以使用表格存储满足需求。

  目前提供了5种分词类型：单字分词、分隔符分词、最小数量语义分词、最大数量语义分词和模糊分词，详情请参见分词。

- 模糊查询

  多元索引提供了通配符查询，等价于关系型数据库中的like功能，您可以指定字符和任意通配符：`?`或`*`，即可实现类似于like的功能。

- 前缀查询

  表格存储多元索引提供前缀查询功能。除了中英文外，其他语言也适用，例如前缀查询“apple”可能会查询出“apple6s”、“applexr”等。

- 嵌套查询

  在线数据中，除了扁平化的一层结构外，还存在一些更复杂的多层结构场景，例如图片标签：某个系统中存储了大量图片，每个图片都有多个实体，例如有房子、有轿车、有人。在每个图片中，这些实体占的位置和空间大小都不同，所以每个实体的权重（score）也不一样，这样相当于每个图片都有多个标签，每个标签有一个名字和一个权重分。如果要根据标签中的条件查询，这时候就需要使用到嵌套查询。

  JSON数据格式

```
{
 "tags": [
   {
      "name": "car",
      "score": 0.78
   },
   {
      "name": "tree",
      "score": 0.24
   }
 ]
}
```

嵌套查询可以优美的解决多层逻辑关系的数据存储和查询，为复杂数据的建模提供了极大的便利性。

- 去重

  多元索引为查询结果提供了去重功能。去重功能可以限制某个属性在一次结果中的最多个数，提供更好的结果多样性能力。例如电商搜索中，搜索一个“笔记本电脑”可能第一页全是某一个品牌的电脑，返回结果对用户并不友好，多元索引的去重功能可以避免这种情况。

- 排序

  表格存储提供主键的字母序排序功能，但如果需要按照其他字段排序，您可以使用多元索引的排序能力。目前表格存储提供了丰富的排序能力，包括正序、逆序；单条件、多条件等。默认返回结果是按照表中的主键顺序排序。多元索引的排序都是全局排序。

- 数据总行数

  使用多元索引查询时可以指定返回这次请求命中的数据行数。如果指定一个空查询条件，则所有创建了索引的数据都符合条件，此时返回的数据总行数就是表中已创建了索引的数据总行数。如果停止写入数据，且数据都已经创建了索引，则此时返回的数据总行数就是数据表中的总行数。此功能可以用于数据校验，运营等场景。

SQL

SQL中的大部分功能在多元索引中相匹配的功能请参见下表。

SQL	多元索引
Show	API：DescribeSearchIndex
Select	参数：ColumnsToGet
From	参数：index name **注意** 已经支持单索引，多索引还未支持。
Where	Query：TermQuery等各种Query
Order by	参数：sort
Limit	参数：limit
Delete	API：先Query再DeleteRow

SQL	多元索引
Like	Query：WildcardQuery
And	参数：operator = and
Or	参数：operator = or
Not	Query：BoolQuery(mustNotQueries)
Between	Query：RangeQuery
Null	Query：ExistsQuery
In	Query：TermsQuery
Min	Aggregation：min
Max	Aggregation：max
Avg	Aggregation：avg
Count	Aggregation：count
Count(distinct)	Aggregation：distinctCount
Sum	Aggregation：sum
Group By	GroupBy

7.3. 数据类型映射

本文介绍了数据表中和多元索引中字段数据类型的映射关系以及不同字段数据类型支持的附加属性信息。

多元索引的字段值来源于数据表中同名字段的值，两者的数据类型必须相匹配，多元索引字段数据类型与数据表中字段数据类型的匹配关系请参见下表。

> **注意** 表中的类型必须一一对应，否则数据会被当做脏数据丢弃，尤其是Geo-point和Nested拥有各自特定的格式。如果格式不匹配也会被当做脏数据丢弃，则会出现数据在表中能查询到，但是在多元索引中查询不到的情况。

多元索引中字段数据类型	数据表中字段数据类型	描述
Long	Integer	64位长整型。
Double	Double	64位双精度浮点数。
Boolean	Boolean	布尔值。
Keyword	String	不可分词字符串。

多元索引中字段数据类型	数据表中字段数据类型	描述
Text	String	可分词字符串或文本。关于分词的更多信息，请参见分词。
Date	Integer、String	日期数据类型，支持自定义各种格式日期数据。更多信息，请参见日期数据类型。
Geo-point	String	位置点坐标信息，格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围为[-180,+180]。例如 `35.8,-45.91` 。
Nested	String	嵌套类型，例如[{"a": 1}, {"a": 3}]。

多元索引字段还支持如下附加属性。

属性	类型	描述
Index	Boolean	是否开启索引。 • 如果为true，则会对该列构建倒排索引或者空间索引。 • 如果为false，则不会对该列构建索引。 如果未构建索引，则不能按照该列进行数据查询。
EnableSortAndAgg	Boolean	是否开启排序与统计聚合功能。 • 如果为true，则可以使用该列进行排序和统计聚合。 • 如果为false，则不能使用该列进行排序或统计聚合。
Store	Boolean	是否在多元索引中附加存储该列的值。 如果为true，则会在多元索引中附加存储该列的值。查询时如果需要读取该列的值，会优先从多元索引中直接读取，无须反查数据表。使用后查询性能更优和更稳定。
IsArray	Boolean	是否为数组。 如果为true，则该列是一个数组。在写入时，也必须按照JSON数组格式写入，例如["a","b","c"]。 Nested类型本身就是一个数组，所以无须设置Array。 Array类型不影响查询，所以Array类型的数据可以用于所有的Query查询。
isVirtualField	Boolean	是否为虚拟列。 • 如果为true，则该列为虚拟列。当该列为虚拟列时，必须设置此列映射的原始字段。关于虚拟列的更多信息，请参见虚拟列。 • 如果为false，则该列不为虚拟列。此时该列的数据类型必须和数据表的数据类型相匹配。

> **说明** Array和Nested的区别请参见Nested和Array对比。

数据类型和字段属性组合使用的情况请参见下表。

类型	Index	EnableSortAndAgg	Store	IsArray	isVirtualField
Long	支持	支持	支持	支持	支持
Double	支持	支持	支持	支持	支持
Boolean	支持	支持	支持	支持	不支持
Keyword	支持	支持	支持	支持	支持
Text	支持	不支持	支持	支持	支持
Date	支持	支持	支持	支持	支持
Geo-point	支持	支持	支持	支持	支持
Nested	只对子字段设置	只对子字段设置	只对子字段设置	Nested本身就是数组	不支持

7.4. 快速入门

7.4.1. 使用控制台

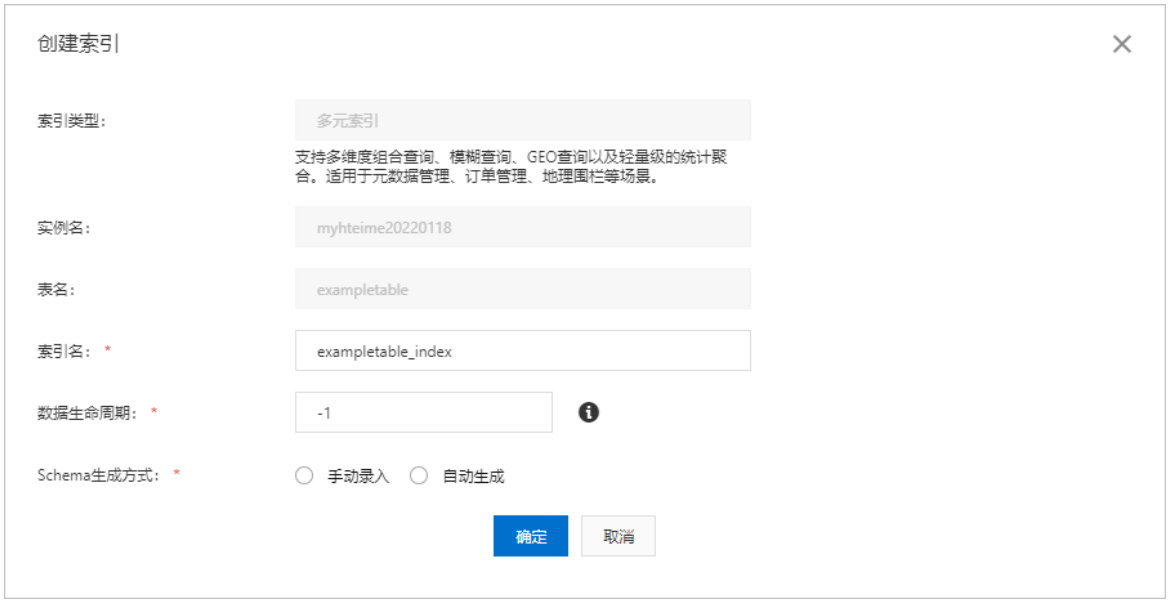
表格存储多元索引（Search Index）基于倒排索引和列式存储，可以解决多种大数据复杂的查询问题。创建多元索引后，您可以使用多元索引进行数据查询。

前提条件

已创建数据表，且数据表的最大版本数（max Versions）必须为1。具体操作，请参见创建数据表。

步骤一：创建多元索引

1. 登录表格存储控制台。
2. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
3. 在**实例详情**页签的**数据表列表**区域，单击数据表名称或在操作列单击**索引管理**。
4. 在**索引管理**页签，单击**创建多元索引**。
5. 在**创建索引**对话框，创建多元索引。



i. 系统默认会自动生成索引名，可根据需要设置索引名。

ii. 多元索引的数据生命周期默认为-1，可根据需要设置数据生命周期。

数据生命周期取值为大于等于86400秒（一天）或者-1（永不过期），单位为秒。同时多元索引的TTL值必须小于或等于数据表的TTL值。关于多元索引数据生命周期的更多信息，请参见多元索引生命周期。

> **注意** 使用多元索引生命周期功能，必须禁用数据表的UpdateRow更新写入功能。

iii. 选择Schema生成方式。

> **注意** 字段名和字段类型需与数据表匹配。数据表字段类型与多元索引字段类型的对应关系请参见数据类型映射。

- 当设置Schema生成方式为**手动录入**时，手动输入字段名，选择字段类型以及设置是否开启数组。
- 当设置Schema生成方式为**自动生成**时，系统会自动将数据表的主键列和属性列作为索引字段，可根据需要选择字段类型以及设置是否开启数组。

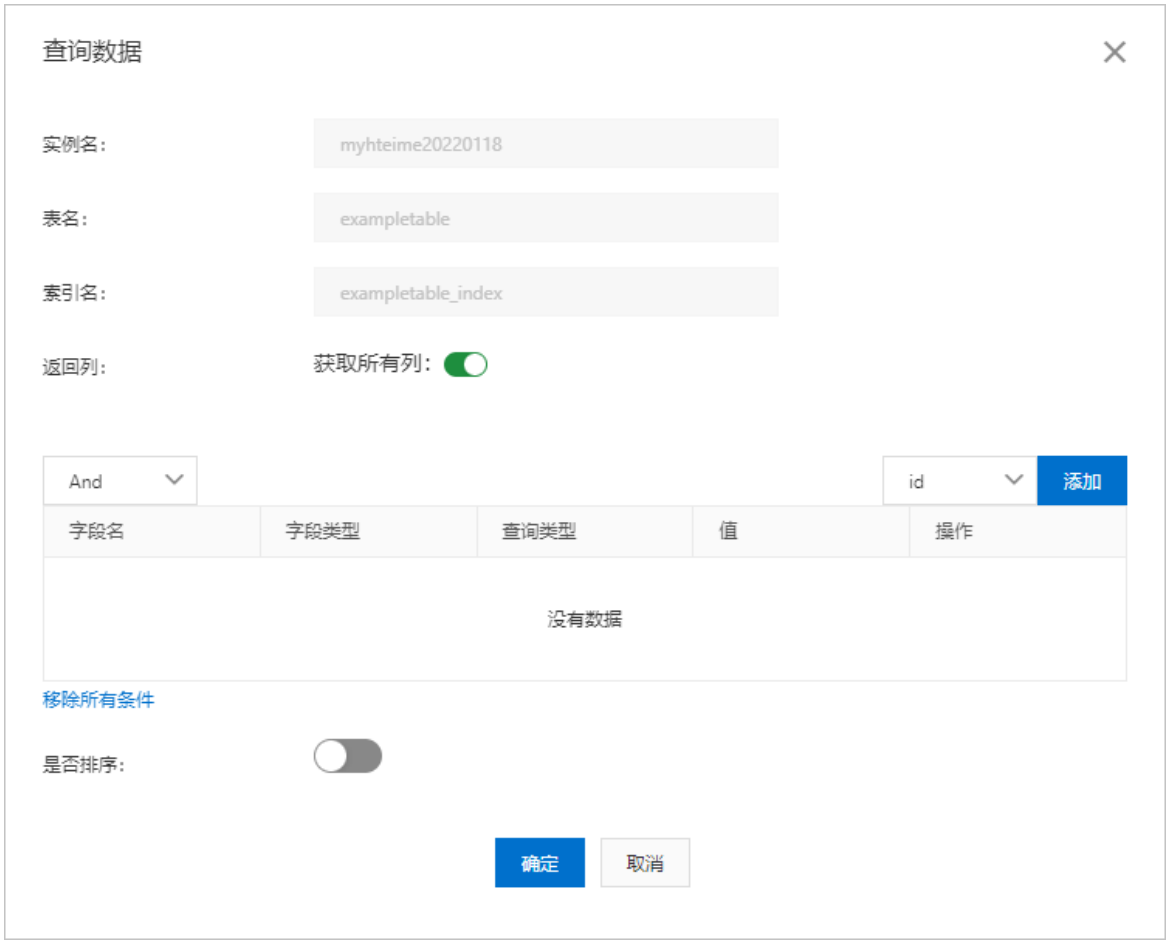
> **说明** 在部分情况下如果要优化性能，则可以使用虚拟列。关于虚拟列的更多信息，请参见虚拟列。

6. 单击**确定**。

多元索引创建完成后，在索引列表的操作列，单击**索引详情**，可查看索引表的索引计量信息、索引字段等信息。

步骤二：查询数据

1. 登录表格存储控制台。
2. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
3. 在**实例详情**页签的**数据表列表**区域，单击数据表名称或在操作列单击**索引管理**。
4. 在**索引管理**页签，单击目标多元索引操作列的**搜索**。



5. 在**查询数据**对话框，查询数据。
   i. 系统默认返回所有列，如需显示指定属性列，关闭**获取所有列**并输入需要返回的属性列，多个属性列之间用半角逗号（,）隔开。

      > **说明** 系统默认会返回数据表的主键列。

   ii. 选择索引字段，单击**添加**，并设置索引字段的查询类型和值。
   iii. 系统默认关闭排序功能，如需根据索引字段对返回结果进行排序，打开**是否排序**后，根据需要添加索引字段并配置排序方式。
6. 单击**确定**。

   符合查询条件的数据会显示在**索引管理**页签中。

相关操作

- 如果要在多元索引中新增、更新或者删除索引列，您可以使用动态修改schema功能实现。具体操作，请参见动态修改schema。
- 如果要在不修改数据表的存储结构和数据的情况下，对新字段新数据类型的查询，您可以使用虚拟列功能实现。具体操作，请参见虚拟列。

7.4.2. 使用命令行工具

表格存储多元索引（Search Index）基于倒排索引和列式存储，可以解决多种大数据复杂的查询问题。创建多元索引后，您可以使用多元索引进行数据查询。

前提条件

- 已下载命令行工具。具体操作，请参见下载。
- 已启动并配置实例。具体操作，请参见启动并配置。
- 已获取AccessKey。具体操作，请参见获取AccessKey。
- 已创建并使用数据表，且数据表的最大版本数（max Versions）必须为1。具体操作，请参见创建并使用数据表。

步骤一：创建多元索引

1. 执行**create_search_index**命令创建一个多元索引search_index。

```
create_search_index -n search_index
```

2. 根据系统提示输入索引Schema，示例如下：

   索引Schema包括IndexSetting（索引设置）、FieldSchemas（Index的所有字段的设置）和IndexSort（索引预排序设置）。关于索引Schema的更多信息，请参见创建多元索引。

```
{
   "IndexSetting": {
       "RoutingFields": null
   },
   "FieldSchemas": [
       {
           "FieldName": "gid",
           "FieldType": "LONG",
           "Index": true,
           "EnableSortAndAgg": true,
           "Store": true,
           "IsArray": false,
           "IsVirtualField": false
       },
       {
           "FieldName": "uid",
           "FieldType": "LONG",
           "Index": true,
           "EnableSortAndAgg": true,
           "Store": true,
           "IsArray": false,
           "IsVirtualField": false
       },
       {
           "FieldName": "col2",
           "FieldType": "LONG",
           "Index": true,
           "EnableSortAndAgg": true,
           "Store": true,
           "IsArray": false,
           "IsVirtualField": false
       },
       {
           "FieldName": "col3",
           "FieldType": "TEXT",
           "Index": true,
```

```
            "Analyzer": "single_word",
            "AnalyzerParameter": {
                "CaseSensitive": true,
                "DelimitWord": null
            },
            "EnableSortAndAgg": false,
            "Store": true,
            "IsArray": false,
            "IsVirtualField": false
        },
        {
            "FieldName": "col1",
            "FieldType": "KEYWORD",
            "Index": true,
            "EnableSortAndAgg": true,
            "Store": true,
            "IsArray": false,
            "IsVirtualField": false
        },
        {
            "FieldName": "col3V",
            "FieldType": "LONG",
            "Index": true,
            "EnableSortAndAgg": true,
            "Store": true,
            "IsArray": false,
            "IsVirtualField": true,
            "SourceFieldNames": [
                "col3"
            ]
        }
    ]
}
```

步骤二：查询数据

1. 执行search命令使用search_index多元索引查询数据，并返回所有建立索引的列。

```
search -n search_index --return_all_indexed
```

2. 根据系统提示输入查询条件，示例如下：

   多元索引支持全匹配查询（MatchAllQuery）、匹配查询（MatchQuery）、短语匹配查询（MatchPhraseQuery）、精确查询（TermQuery）、多词精确查询（TermsQuery）、前缀查询（PrefixQuery）等多种查询方式，此处以精确查询为例介绍。关于精确查询的更多信息，请参见精确查询。

```
{
    "Offset": -1,
    "Limit": 10,
    "Collapse": null,
    "Sort": null,
    "GetTotalCount": true,
    "Token": null,
    "Query": {
        "Name": "TermQuery",
        "Query": {
            "FieldName": "uid",
            "Term": 10001
        }
    },
    "Aggregations": [{
        "Name": "avg",
        "Aggregation": {
            "AggName": "agg1",
            "Field": "pid"
        }
    }]
}
```

7.5. 使用SDK

表格存储多元索引（Search Index）基于倒排索引和列式存储，可以解决多种大数据复杂的查询问题。创建多元索引后，您可以使用多元索引进行数据查询。

前提条件

- 已创建数据表，且数据表的最大版本数（max Versions）必须为1。
- 已初始化Client。具体操作，请参见初始化。

接口

分类	接口	描述
管控接口	CreateSearchIndex	创建一个多元索引。
	DescribeSearchIndex	获取多元索引的详细描述信息。
	ListSearchIndex	列出多元索引的列表。
	DeleteSearchIndex	删除某个多元索引。
查询接口	Search	查询接口包括所有的查询功能，以及排序、统计聚合等分析能力，其结果会按照指定的顺序返回。

查询接口 分类	接口	描述
	ParallelScan	数据导出接口只包括所有的查询功能，舍弃了排序、统计聚合等分析能力，能将命中的数据以更快的速度全部返回。 多并发导出数据时，您还需要通过ComputeSplits接口获取当前ParallelScan单个请求的最大并发数。

操作步骤

1. 创建多元索引。具体操作，请参见创建多元索引。
2. 使用Search接口或者ParallelScan接口查询数据，请根据下表说明选择合适的查询方式。

名称	Query	描述
全匹配查询	MatchAllQuery	常用于查询表中数据总行数，或者随机返回几条数据。
匹配查询	MatchQuery	采用近似匹配的方式查询表中的数据。会先对query内容按照配置好的分词器做切分，然后按照切分好后的词去查询。 不同分词之间的关系是Or，当分词后的多个词只要有部分匹配时，则表示行数据满足查询条件。
短语匹配查询	MatchPhraseQuery	类似于MatchQuery，但是只有当分词后的多个词必须在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。
精确查询	TermQuery	采用完整精确匹配的方式查询表中的数据，类似于字符串匹配。 对于分词字符串（Text）类型，当分词后的多个词只要有词可以精确匹配时，则表示行数据满足查询条件。
多词精确查询	TermsQuery	类似于TermQuery，但是可以一次指定多个词，当多个词中只要有一个词匹配，则表示行数据满足查询条件。
前缀查询	PrefixQuery	根据前缀条件查询表中的数据。 对于分词字符串（Text）类型，当分词后的多个词只要有词满足前缀条件时，则表示行数据满足查询条件。
范围查询	RangeQuery	根据范围条件查询表中的数据。 对于分词字符串（Text）类型，当分词后的多个词只要有词满足范围条件时，则表示行数据满足查询条件。

名称	Query	描述
通配符查询	WildcardQuery	要匹配的值可以是一个带有通配符的字符串。 要匹配的值中可以用星号（“*”）代表任意字符序列，或者用问号（“?”）代表任意单个字符。
多字段自由组合查询	BoolQuery	查询条件可以包含一个或者多个子查询条件，根据子查询条件是否满足来判断一行数据是否满足查询条件。 子查询条件的组合关系支持And、Or、Not等。
嵌套查询	NestedQuery	查询嵌套类型中子文档的数据。
地理距离查询	GeoDistanceQuery	根据一个中心点和距离条件查询表中的数据，当一个地理位置点到指定的中心点的距离不超过指定的值时，则表示行数据满足查询条件。
地理长方形范围查询	GeoBoundingBoxQuery	根据一个长方形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在给出的长方形范围内时，则表示行数据满足查询条件。
地理多边形范围查询	GeoPolygonQuery	根据一个多边形范围条件查询表中的数据，当一个地理位置点落在指定的多边形范围内时，则表示行数据满足查询条件。
列存在性查询	ExistQuery	也叫NULL查询或者空值查询。一般用于稀疏数据中，用于判断某一行的某一列是否存在。例如查询所有数据中，address列不为空的行有哪些。 只有当某一列在行数据中不存在或者为空数组（"[]"）时，则表示在行数据中该列不存在。

相关操作

- 如果要进行数据分析，例如求最值、求和、统计行数等，您可以使用Search接口的统计聚合功能来实现。具体操作，请参见统计聚合。
- 如果要快速导出数据，而不关心整个结果集的顺序时，您可以使用ParallelScan接口和ComputeSplits接口实现多并发导出数据。具体操作，请参见并发导出数据。
- 当通过Search接口查询数据时，如果要对结果集进行排序或者翻页，您可以使用排序和翻页功能来实现。具体操作，请参见排序和翻页。
- 当通过Search接口查询数据时，如果要按照某一列对结果集做折叠，使对应类型的数据在结果展示中只出现一次，您可以使用折叠（去重）功能来实现。具体操作，请参见折叠（去重）。
- 如果要进行全文检索，您可以对可分词类型的字段进行分词后再选择合适的查询方式来实现。具体操作，请参见分词。
- 如果要进行多层逻辑关系的数据存储和查询，您可以使用嵌套类型存储数据后再选择嵌套查询来实现。具体操作，请参见嵌套类型和嵌套类型查询。
- 如果要系统自动清理多元索引中超过保存时间的数据，您可以使用多元索引生命周期功能。具体操作，请参见多元索引生命周期。
- 如果要在多元索引中新增、更新或者删除索引列，您可以使用动态修改schema功能实现。具体操作，请参见动态修改schema。

- 如果要在不修改数据表的存储结构和数据的情况下，对新字段新数据类型的查询，您可以使用虚拟列功能实现。具体操作，请参见虚拟列。

7.6. 基础功能

7.6.1. 创建多元索引

在数据表上创建一个多元索引后，您可以根据多元索引中建立索引的字段来查询数据表中的数据。一张数据表可以创建多个多元索引。

最佳使用方式

请根据实际查询需求确定在一张数据表中创建多元索引的个数。

例如有一张数据表有5个字段，分别为id、name、age、city、sex，需要按照name、age或city查询数据时，有两种创建多元索引的方式。

- 方法一：一个字段建立一个多元索引

  按照name、age或city字段分别创建多元索引，名称分别为name_index、age_index、city_index。

  - 如果按照城市查询学生，则查询city_index，如果按照年龄查询学生，则查询age_index。
  - 如果查询年龄小于12岁且城市是成都的学生，此种方式就无法查询。

  此方式的实现类似于全局二级索引，使用此方式创建多元索引会带来更高的费用，因此不建议使用此方式创建多元索引。

- 方法二：多个字段创建在一个多元索引中

  将name、age和city字段创建在一个多元索引中，名称为student_index。

  - 如果按照城市查询学生，则查询student_index中city字段；如果按照年龄查询学生，则查询student_index中age字段。
  - 如果查询年龄小于12岁且城市是成都的学生，则查询student_index中age和city字段。

  此方式才能发挥多元索引最大优势，不仅功能更丰富，而且价格会更低。极力推荐使用此方式创建多元索引。

限制

- 创建索引的时效性

  创建多元索引后，需要等几秒钟才能使用，但是此过程中不影响数据写入，只影响索引元信息的查询和索引查询。

- 数量限制

  更多信息，请参见多元索引限制。

接口

创建多元索引的接口为CreateSearchIndex。

使用

您可以使用如下语言的SDK创建多元索引。

- Java SDK：创建多元索引
- Go SDK：创建多元索引

- Python SDK：创建多元索引
- Node.js SDK：创建多元索引
- .NET SDK：创建多元索引
- PHP SDK：创建多元索引

参数

创建多元索引时，需要指定数据表名称（tableName）、多元索引名称（indexName）和索引的结构信息（indexSchema），其中indexSchema包含fieldSchemas（Index的所有字段的设置）、indexSetting（索引设置）和indexSort（索引预排序设置）。详细参数说明请参见下表。

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

参数	说明
fieldSchemas	fieldSchema的列表，每个fieldSchema包含如下内容： • fieldName（必选）：创建多元索引的字段名，即列名，类型为String。 多元索引中的字段可以是主键列或者属性列。 • fieldType（必选）：字段类型，类型为FieldType.XXX。更多信息，请参见数据类型映射。 • array（可选）：是否为数组，类型为Boolean。 如果设置为true，则表示该列是一个数组，在写入时，必须按照JSON数组格式写入，例如["a","b","c"]。 由于Nested类型是一个数组，当fieldType为Nested类型时，无需设置此参数。 • index（可选）：是否开启索引，类型为Boolean。 默认为true，表示对该列构建倒排索引或者空间索引；如果设置为false，则不会对该列构建索引。 • analyzer（可选）：分词器类型。当字段类型为Text时，可以设置此参数；如果不设置此参数，则默认分词器类型为单字分词。关于分词的更多信息，请参见分词。 • enableSortAndAgg（可选）：是否开启排序与统计聚合功能，类型为Boolean。 只有设置enableSortAndAgg为true的字段才能进行排序。关于排序的更多信息，请参见排序和翻页。 注意 Nested类型的字段不支持开启排序与统计聚合功能，但是Nested类型内部的子列支持开启排序与统计聚合功能。 • store（可选）：是否在多元索引中附加存储该字段的值，类型为Boolean。 开启后，可以直接从多元索引中读取该字段的值，而不必反查数据表，可用于查询性能优化。 • isVirtualField（可选）：该字段是否为虚拟列，类型为Boolean类型，默认值为false。只有使用虚拟列时，才需要设置此参数。关于虚拟列的更多信息，请参见虚拟列。 • sourceFieldName（可选）：数据表中的字段名称，类型为String。当设置isVirtualField为true时，必须设置此参数。 • dateFormats（可选）：日期的格式，类型为String。当字段类型为Date时，必须设置此参数。更多信息，请参见日期数据类型。
indexSetting	索引设置，包含routingFields设置。 routingFields（可选）：自定义路由字段。可以选择部分主键列作为路由字段，在进行索引数据写入时，会根据路由字段的值计算索引数据的分布位置，路由字段的值相同的记录会被索引到相同的数据分区中。

参数	说明
indexSort	索引预排序设置，包含sorters设置。如果不设置，则默认按照主键排序。 **说明** 含有Nested类型的索引不支持indexSort，没有预排序。 sorters（必选）：索引的预排序方式，支持按照主键排序和字段值排序。更多信息，请参见排序和翻页。 • PrimaryKeySort表示按照主键排序，包含如下设置： order：排序的顺序，可按升序或者降序排序，默认为升序（SortOrder.ASC）。 • FieldSort表示按照字段值排序， 包含如下设置： 只有建立索引且开启排序与统计聚合功能的字段才能进行预排序。 ○ fieldName：排序的字段名。 ○ order：排序的顺序，可按照升序或者降序排序，默认为升序（SortOrder.ASC）。 ○ mode：当字段存在多个值时的排序方式。
timeToLive	可选参数，默认值为-1。数据生命周期（TTL），即数据的保存时间。 当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。 多元索引生命周期的使用方式，请参见多元索引生命周期。

示例

- 创建多元索引。

  创建一个多元索引，包含Col_Keyword和Col_Long两列，类型分别设置为字符串（String）和整型（Long）。

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName); //设置数据表名称。
    request.setIndexName(indexName); //设置多元索引名称。
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD) //设置字段名和类型。
                    .setIndex(true) //设置开启索引。
                    .setEnableSortAndAgg(true) //设置开启排序与统计聚合功能。
                    .setStore(true), //设置在多元索引中附加存储该列的值。
            new FieldSchema("Col_Long", FieldType.LONG)
                    .setIndex(true)
                    .setEnableSortAndAgg(true)
                    .setStore(true)));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); //调用client创建多元索引。
}
```

- 创建多元索引时指定IndexSort。

```
private static void createSearchIndexWithIndexSort(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName);
    request.setIndexName(indexName);
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD).setIndex(true).setEnableSor
tAndAgg(true).setStore(true),
            new FieldSchema("Col_Long", FieldType.LONG).setIndex(true).setEnableSortAndAg
g(true).setStore(true),
            new FieldSchema("Col_Text", FieldType.TEXT).setIndex(true).setStore(true),
            new FieldSchema("Timestamp", FieldType.LONG).setIndex(true).setEnableSortAndA
gg(true).setStore(true)));
    //设置按照Timestamp列进行预排序，Timestamp列必须建立索引且开启enableSortAndAgg。
    indexSchema.setIndexSort(new Sort(
            Arrays.<Sort.Sorter>asList(new FieldSort("Timestamp", SortOrder.ASC))));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request);
}
```

- 创建多元索引时设置生命周期

> **注意** 请确保数据表的更新状态为禁止。

```
// 请使用5.12.0及以上版本的Java SDK。
public void createIndexWithTTL(SyncClient client) {
    int days = 7;
    CreateSearchIndexRequest createRequest = new CreateSearchIndexRequest();
    createRequest.setTableName(tableName);
    createRequest.setIndexName(indexName);
    createRequest.setIndexSchema(indexSchema);
    // 设置多元索引TTL。
    createRequest.setTimeToLiveInDays(days);
    client.createSearchIndex(createRequest);
}
```

- 创建多元索引时指定虚拟列。

  创建一个多元索引，多元索引包含Col_Keyword和Col_Long两列，同时创建虚拟列Col_Keyword_Virtual_Long和Col_Long_Virtual_Keyword。Col_Keyword_Virtual_Long映射为数据表中Col_Keyword列，虚拟列Col_Long_Virtual_Keyword映射为数据表中Col_Long列。

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName); //设置数据表名称。
    request.setIndexName(indexName); //设置多元索引名称。
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
        new FieldSchema("Col_Keyword", FieldType.KEYWORD) //设置字段名和类型。
            .setIndex(true) //设置开启索引。
            .setEnableSortAndAgg(true) //设置开启排序和统计功能。
            .setStore(true),
        new FieldSchema("Col_Keyword_Virtual_Long", FieldType.LONG) //设置字段名和类型。
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true) //设置字段是否为虚拟列。
            .setSourceFieldName("Col_Keyword"), //虚拟列对应的数据表中字段。
        new FieldSchema("Col_Long", FieldType.LONG)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true),
        new FieldSchema("Col_Long_Virtual_Keyword", FieldType.KEYWORD)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true)
            .setSourceFieldName("Col_Long")));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); //调用client创建多元索引。
}
```

7.6.2. 多元索引生命周期

数据生命周期（Time To Live，简称TTL）是多元索引的一个属性，即数据的保存时间。多元索引会自动清理超过保存时间的数据，减少用户的数据存储空间，降低存储成本。

注意事项

- 使用多元索引生命周期功能，必须禁用数据表的UpdateRow更新写入功能，避免一些语义上的问题：

  由于数据表TTL是属性列级别生效的，而多元索引TTL是整行生效的，如果存在UpdateRow写入操作，当系统清理数据表中数据时，数据表中部分字段值已删除而部分字段值未删除，但是多元索引中整行数据均未删除，则会造成数据表和多元索引中的数据不一致。

  如果业务有UpdateRow更新写入操作，请查看是否能改为PutRow覆盖写入操作。
- 多元索引的TTL取值范围为-1或者int32的正整数（单位为秒），其中-1表示永久存储，int32最大值换算为年大约为68年。
- 多元索引的TTL和数据表的TTL是独立的，多元索引的TTL值必须小于或等于数据表的TTL值。当需要同时调小多元索引TTL和数据表TTL时，请先调整多元索引TTL，再调整数据表TTL。
- 多元索引每天会自动清理已过期的数据，过期数据的清理粒度为“天”，因此您仍然可以查询到某一时刻已过期但是还未及时清理的数据，多元索引会在下一次清理过期数据时自动清理这些过期数据。
- 数据表和多元索引的TTL更新后，系统会在下一次清理过期数据时自动清理数据表和多元索引中的存量过期数据。

使用流程

通过控制台或者SDK设置多元索引生命周期。使用多元索引生命周期，您必须一直保持数据表UpdateRow更新写入操作为禁止状态。

使用控制台

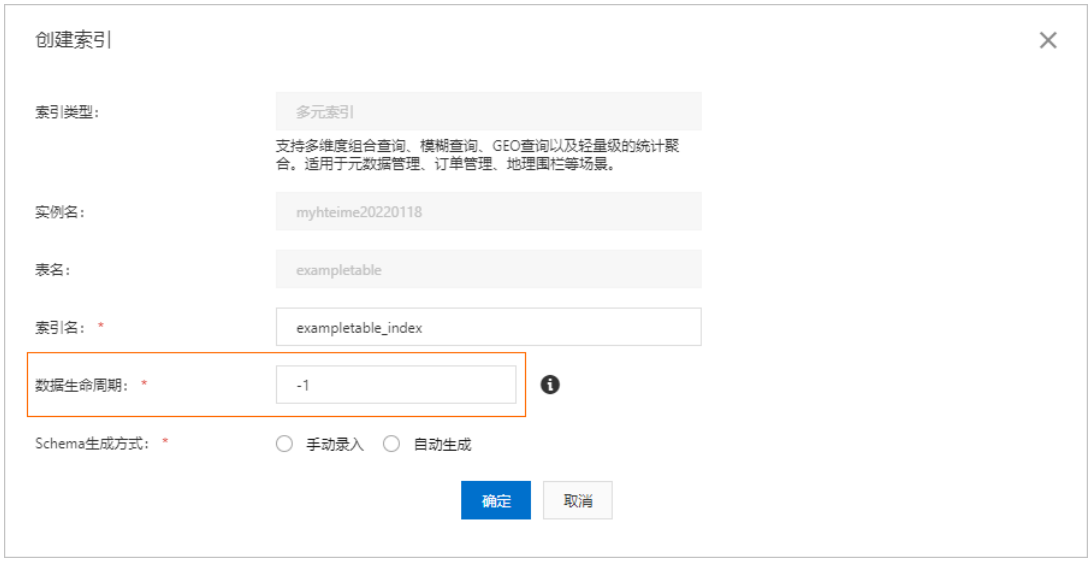
1. 禁用数据表UpdateRow更新写入操作。
   i. 在数据表的**基本详情**页签，单击**修改表属性**。
   ii. 在**修改表属性**对话框，设置是否允许更新为**否**，然后单击**确定**。

2. 设置多元索引生命周期。

   禁用数据表UpdateRow更新写入操作后，您可以在创建多元索引时指定TTL或者为已有多元索引指定TTL。

   - 创建多元索引时指定TTL
     a. 在数据表的**索引管理**页签，单击**创建多元索引**。

b. 在**创建索引**对话框，设置索引名、数据生命周期和Schema生成方式，然后单击**确定**。



- 为已有多元索引指定TTL

a. 在数据表的**索引管理**页签，单击目标多元索引操作列的**索引详情**。

b. 在**索引详情**对话框，单击图标，修改数据生命周期，然后单击**修改ttl**。

c. 单击**确定**。

3. 多元索引的TTL和数据表的TTL是独立的。如果需要使用数据表TTL，请为数据表设置TTL。
   i. 在数据表的**基本详情**页签中**基本信息**区域，单击**修改表属性**。
   ii. 在**修改表属性**对话框，根据需要设置数据生命周期，然后单击**确定**。

使用SDK

1. 禁用数据表UpdateRow更新写入操作。

```
public void disableTableUpdate(SyncClient client) {
    UpdateTableRequest updateTableRequest = new UpdateTableRequest(tableName);
    TableOptions options = new TableOptions();
    // 禁用数据表Update更新写入操作，请确保数据表无Update写入操作，避免影响业务。
    options.setAllowUpdate(false);
    updateTableRequest.setTableOptionsForUpdate(options);
    client.updateTable(updateTableRequest);
}
```

2. 设置多元索引生命周期。

禁用数据表UpdateRow更新写入操作后，您可以在创建多元索引时指定TTL或者为已有多元索引指定TTL。

- 创建多元索引时指定TTL

```
// 请使用5.12.0及以上版本的Java SDK。
public void createIndexWithTTL(SyncClient client) {
    int days = 7;
    CreateSearchIndexRequest createRequest = new CreateSearchIndexRequest();
    createRequest.setTableName(tableName);
    createRequest.setIndexName(indexName);
    createRequest.setIndexSchema(indexSchema);
    // 设置多元索引TTL。
    createRequest.setTimeToLiveInDays(days);
    client.createSearchIndex(createRequest);
}
```

- 为已有多元索引指定TTL

```
// 请使用5.12.0及以上版本的Java SDK。
public void updateIndexWithTTL(SyncClient client) {
    int days = 7;
    UpdateSearchIndexRequest updateSearchIndexRequest = new UpdateSearchIndexRequest(
tableName, indexName);
    // 更新多元索引TTL。
    updateSearchIndexRequest.setTimeToLiveInDays(days);
    client.updateSearchIndex(updateSearchIndexRequest);
}
```

3. 多元索引的TTL和数据表的TTL是独立的。如果需要使用数据表TTL，请为数据表设置TTL。

```
public void updateTableTTL(SyncClient client) {
    int days = 7;
    UpdateTableRequest updateTableRequest = new UpdateTableRequest(tableName);
    TableOptions options = new TableOptions();
    options.setTimeToLiveInDays(days);
    updateTableRequest.setTableOptionsForUpdate(options);
    client.updateTable(updateTableRequest);
}
```

7.6.3. 查询多元索引描述信息

创建多元索引后，可以查询多元索引的描述信息，包括多元索引的字段信息和索引配置等。

接口

查询多元索引描述信息的接口为DescribeSearchIndex。

使用

您可以使用如下语言的SDK查询多元索引描述信息。

- Java SDK：查询多元索引描述信息
- Go SDK：查询多元索引描述信息
- Python SDK：查询多元索引描述信息

- Node.js SDK：查询多元索引描述信息
- .NET SDK：查询多元索引描述信息
- PHP SDK：查询多元索引描述信息

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

示例

```
private static DescribeSearchIndexResponse describeSearchIndex(SyncClient client) {
    DescribeSearchIndexRequest request = new DescribeSearchIndexRequest();
    request.setTableName(TABLE_NAME); //设置数据表名称。
    request.setIndexName(INDEX_NAME); //设置多元索引名称。
    DescribeSearchIndexResponse response = client.describeSearchIndex(request);
    System.out.println(response.jsonize()); //打印response的详细信息。
    System.out.println(response.getSyncStat().getSyncPhase().name());//打印多元索引数据同步状
态。
    return response;
}
```

7.6.4. 日期数据类型

多元索引支持丰富的日期数据类型，您可以将数据表中整型（Integer）或者字符串（String）类型的数据在多元索引中映射为日期数据类型。当通过多元索引进行范围查询时，使用日期数据类型查询会比使用字符串类型查询更快。

精度和范围

日期类型支持的最大精度是纳秒，日期类型数据的取值范围为 `["1970-01-01 00:00:00.000000000", "2262-04-11 23:47:16.854775807"]` 。

日期格式

数据表中的字段类型映射到多元索引的日期类型时，日期类型的格式支持设置，详细说明请参见下表。

数据表中字段类型	多元索引日期类型的格式（Format）

数据表中字段类型	多元索引日期类型的格式（Format）
整型（Integer）	支持选择预定义格式。预定义格式包括如下选项： • "epoch_second"：表示秒时间戳，例如秒时间戳"1218197720"对应的时间为"2008-08-08 20:15:20"。 • "epoch_millis"：表示毫秒时间戳，例如毫秒时间戳"1218197720123"对应的时间为"2008-08-08 20:15:20.123"。 • "epoch_micros"：表示微秒时间戳，例如微秒时间戳"1218197720123456"对应的时间为"2008-08-08 20:15:20.123456"。 • "epoch_nanos"：表示纳秒时间戳，例如纳秒时间戳"1218197720123456789"对应的时间为"2008-08-08 20:15:20.123456789"。
字符串（String）	支持自定义格式。常用日期格式如下： • yyyy-MM-dd HH:mm:ss.SSS • yyyyMMdd HHmmss • yyyy-MM-dd'T'HH:mm:ss.SSSX 其中yyyy表示4位的年份，MM表示月份，dd表示天，HH表示24小时制，mm表示分钟，ss表示秒，SSS表示秒的精度，X表示时区偏移量。 关于自定义日期格式规则的更多信息，请参见自定义日期格式规则。

自定义日期格式规则

符号	含义	示例
y	年份	• yyyy：2008 • yy：08
M	月份	• M：7 • MM：07
d	月分中的第几天	• d：8 • dd：08
a	上午（AM）或者下午（PM）标记	• a：AM • a：PM
K	AM或者PM中的小时，取值范围为0~11	• K：0 • KK：00
H	一天中的小时，取值范围为0~23	• H：0 • HH：00

符号	含义	示例
m	分钟	• m：1 • mm：01
s	秒	• s：1 • ss：01
S	秒的精度，支持1~9个数字	• S：3 • SSS：234 • SSSSSSSSS：123456789
X	时区偏移	• X：+01;Z • XX：+0130;Z • XXX：+01:30;Z • XXXX：+013015;Z • XXXXX：+01:30:15;Z
x	时区偏移	• x：+01;+00 • xx：+0130;+0000, • xxx：+01:30;+00:00 • xxxx：+013015;+0000 • xxxxx：+01:30:15
'	自定义符号限定符	A~Z和a~z字符为特殊符号，当添加自定义字符串时，需要使用单引号包括。 **注意** 空格和短划线（-）不需要使用单引号包括。
''	转义单引号	''

日期格式合法性验证

使用日期格式查询数据前，建议您通过如下方式验证日期格式是否合法。

- 通过创建包含指定日期格式的多元索引后，使用精确查询（TermQuery）进行验证

```
public void testDateFormat(SyncClient client, String tableName, String indexName) {
    // 创建多元索引。
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName);
    request.setIndexName(indexName);
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
        new FieldSchema("col_date", FieldType.DATE)
        .setIndex(true)
        .setEnableSortAndAgg(true)
        .setDateFormats(Arrays.asList("yyyy-MM-dd HH:mm:ss.SSS"))
    ));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request);
    // 验证日期格式。如果查询不报错，则日期格式正确。
    client.search(SearchRequest.newBuilder()
                  .tableName(tableName)
                  .indexName(indexName)
                  .searchQuery(SearchQuery
                               .newBuilder()
                               .query(QueryBuilders.term("col_date", "2012-12-12 12:10:03
.123")).build())
                  .build());
}
```

- 通过JDK8及以上版本的DateTimeFormatter进行验证

> **说明** 此方式在测试某些时区时可能存在误差。

```
import java.time.format.DateTimeFormatter;
public void testFormatByJdk8() {
    DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss.SSS").parse("2012-12-12 12:10:03.123
");
}
```

7.6.5. 数组和嵌套类型

多元索引除了提供Long、Double、Boolean、Keyword、Text和GeoPoint等基本类型外，还提供了数组类型和嵌套类型两种特殊类型。

数组类型

数组类型属于附加类型，可以附加在Long、Double、Boolean、Keyword、Text和GeoPoint等基本类型之上。例如Long类型+数组后，即为数组长整型，该字段中可以包括多个长整型数字，查询数据时其中任何一个匹配都可以返回该行数据。

多元索引的基本类型数组格式请参见下表。

数组类型	说明
Long Array	长整型的数组形式，格式为“[1000, 4, 5555]”。

数组类型	说明
Boolean Array	布尔值的数组形式，格式为“[true, false]”。
Double Array	浮点数的数组形式，格式为“[3.1415926, 0.99]”。
Keyword Array	字符串的数组形式，格式为JSON Array，例如 “[\"杭州\", \"西安\"]” 。
Text Array	文本的数组形式，格式为JSON Array，例如“[\"杭州\", \"西安\"]” 。 对于Text类型，一般无需使用数组形式。
GeoPoint Array	地理位置点的数组形式，格式为"[\"34.2, 43.0\", \"21.4, 45.2\"]"。

数组类型仅是多元索引中的概念，数据表中还未支持数组。对于多元索引中数组类型的字段，在数据表中必须为String类型，且对应的多元索引中的类型必须为相应的类型，例如Long、Double等。如果字段price是Double Array数组类型，则在数据表中price必须为String类型，在对应的多元索引中的类型必须为Double类型，且附加isArray=true属性。

嵌套类型

嵌套类型（Nested）代表嵌套文档类型。嵌套文档是指对于一行数据（文档）可以包含多个子行（子文档），多个子行保存在一个嵌套类型字段中。对于嵌套类型字段，需要指定其子行的结构，即子行中包含哪些字段以及每个字段的属性。嵌套类型类似数组，但是功能更丰富。

嵌套类型字段在写入数据表时为字符串类型，对应到多元索引嵌套字段类型的格式为JSON对象的数组，例如[{"tagName":"tag1", "score":0.8}, {"tagName":"tag2", "score":0.2}]。

> **注意** 即使只有一个子行，也必须按照JSON数组的格式构造字符串。

- 单层级嵌套类型示例

  单层级嵌套类型可以通过控制台或者SDK创建。

  以Java代码为例介绍创建单层级嵌套类型。示例中嵌套类型字段的名称为tags，子行中包含两个字段，如下图所示。

字段名	字段类型	数组	索引	统计排序	存储	分词	分词参数
id	字符串	否	是	是	是		
tags	嵌套文档	否	否	否	否		
tagName	字符串	否	是	是	是		
score	浮点数	否	是	是	是		

  - 一个字段名称为tagName，类型为字符串类型（Keyword）。
  - 一个字段名称为score，类型为浮点数（Double）。

  写入数据表时的数据样例为 `[{"tagName":"tag1", "score":0.8}, {"tagName":"tag2", "score":0.2}]` 。

```
//构造子行的FieldSchema。
List<FieldSchema> subFieldSchemas = new ArrayList<FieldSchema>();
subFieldSchemas.add(new FieldSchema("tagName", FieldType.KEYWORD)
    .setIndex(true).setEnableSortAndAgg(true));
subFieldSchemas.add(new FieldSchema("score", FieldType.DOUBLE)
    .setIndex(true).setEnableSortAndAgg(true));
//将子行的FieldSchema设置到嵌套类型字段的subfieldSchemas中。
FieldSchema nestedFieldSchema = new FieldSchema("tags", FieldType.NESTED)
    .setSubFieldSchemas(subFieldSchemas);
```

- 多层级嵌套类型示例

  多层级嵌套类型只能通过SDK创建。

  以Java示例代码为例介绍创建多层级嵌套类型。示例中嵌套类型字段的名称为user，子行中包含三个基础类型字段和一个嵌套类型字段。

  - 一个字段名称为name，类型为字符串类型（Keyword）。
  - 一个字段名称为age，类型为长整型（Long）。
  - 一个字段名称为phone，类型为字符串类型（Keyword）。
  - 一个嵌套类型字段的名称为address，子行中包含的三个字段名称分别为province、city和street，类型均为字符串类型（Keyword）。

  写入数据表时的数据样例为 `[ {"name":"张三","age":20,"phone":"13900006666","address":[{"province":"浙江省","city":"杭州市","street":"阳光大道幸福小区1201号"}]}]` 。

```
//构造嵌套类型字段address的子行FieldSchema，子行中包含三个字段。查询子行中字段的数据时，子行中字段的路径为user.address。
List<FieldSchema> addressSubFiledSchemas = new ArrayList<>();
addressSubFiledSchemas.add(new FieldSchema("province",FieldType.KEYWORD));
addressSubFiledSchemas.add(new FieldSchema("city",FieldType.KEYWORD));
addressSubFiledSchemas.add(new FieldSchema("street",FieldType.KEYWORD));
//构造嵌套类型字段user的子行FieldSchema，子行中包含三个基础类型字段和一个嵌套类型字段address。查询子行中字段的数据时，子行中字段的路径为user。
List<FieldSchema> subFieldSchemas = new ArrayList<>();
subFieldSchemas.add(new FieldSchema("name",FieldType.KEYWORD));
subFieldSchemas.add(new FieldSchema("age",FieldType.LONG));
subFieldSchemas.add(new FieldSchema("phone",FieldType.KEYWORD));
subFieldSchemas.add(new FieldSchema("address",FieldType.NESTED).setSubFieldSchemas(addressSubFiledSchemas));
//将嵌套类型字段user的子行FieldSchema设置到嵌套类型字段的subfieldSchemas中。
List<FieldSchema> fieldSchemas = new ArrayList<>();
fieldSchemas.add(new FieldSchema("user",FieldType.NESTED).setSubFieldSchemas(subFieldSchemas));
```

嵌套类型的局限性如下：

- 由于含有嵌套类型字段的多元索引不支持索引预排序（IndexSort），而索引预排序功能在很多场景下可以带来很大性能提升。
- 如果使用含有嵌套类型字段的多元索引查询数据且需要翻页，则必须在查询条件中指定数据返回的排序方式，否则当符合查询条件的数据未读取完时，服务端不会返回nextToken。
- 嵌套类型的查询性能相比其他类型的查询性能更低一些。

嵌套类型除了上述局限性外，和非嵌套类型支持的功能相同，支持所有的查询类型、排序和统计聚合。

如果想了解更多Array和Nested的对比，请参见Array和Nested对比。

7.6.6. 列出多元索引列表

创建多元索引后，可以查询当前实例下或某个数据表关联的所有多元索引的列表信息。

接口

列出多元索引列表的接口为ListSearchIndex。

使用

您可以使用如下语言的SDK列出多元索引列表。

- Java SDK：列出多元索引列表
- Go SDK：列出多元索引列表
- Python SDK：列出多元索引列表
- Node.js SDK：列出多元索引列表
- .NET SDK：列出多元索引列表
- PHP SDK：列出多元索引列表

参数

参数	说明
tableName	数据表名称，可以为空。 • 如果设置了数据表名称，则返回该数据表关联的所有多元索引的列表。 • 如果未设置数据表名称，则返回当前实例下所有多元索引的列表。

示例

```
private static List<SearchIndexInfo> listSearchIndex(SyncClient client) {
    ListSearchIndexRequest request = new ListSearchIndexRequest();
    request.setTableName(TABLE_NAME); //设置数据表名称。
    return client.listSearchIndex(request).getIndexInfos(); //获取数据表关联的所有多元索引。
}
```

7.6.7. 删除多元索引

删除指定数据表的一个多元索引。

接口

删除多元索引的接口为DeleteSearchIndex。

使用

您可以使用如下语言的SDK删除多元索引。

- Java SDK：删除多元索引
- Go SDK：删除多元索引

- Python SDK：删除多元索引
- Node.js SDK：删除多元索引
- .NET SDK：删除多元索引
- PHP SDK：删除多元索引

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

示例

```
private static void deleteSearchIndex(SyncClient client) {
    DeleteSearchIndexRequest request = new DeleteSearchIndexRequest();
    request.setTableName(TABLE_NAME); //设置数据表名称。
    request.setIndexName(INDEX_NAME); //设置多元索引名称。
    client.deleteSearchIndex(request); //调用client删除多元索引。
}
```

7.6.8. 排序和翻页

您可以在创建多元索引时指定索引预排序和在使用多元索引查询数据时指定排序方式，以及在获取返回结果时使用limit和offset或者使用token进行翻页。

使用

您可以使用如下语言的SDK实现排序和翻页。

- Java SDK：排序和翻页
- Go SDK：排序和翻页
- Python SDK：排序和翻页
- Node.js SDK：排序和翻页
- .NET SDK：排序和翻页
- PHP SDK：排序和翻页

索引预排序

多元索引默认按照设置的索引预排序（IndexSort）方式进行排序，使用多元索引查询数据时，IndexSort决定了数据的默认返回顺序。

在创建多元索引时，您可以自定义IndexSort，如果未自定义IndexSort，则IndexSort默认为主键排序。

> 注意 含有Nested类型字段的多元索引不支持索引预排序。

查询时指定排序方式

只有enableSortAndAgg设置为true的字段才能进行排序。

在每次查询时，可以指定排序方式，多元索引支持如下四种排序方式（Sorter）。您也可以使用多个Sorter，实现先按照某种方式排序，再按照另一种方式排序的需求。

- ScoreSort

  按照查询结果的相关性（BM25算法）分数进行排序，适用于有相关性的场景，例如全文检索等。

  > **注意** 如果需要按照相关性打分进行排序，必须手动设置ScoreSort，否则会按照索引设置的IndexSort进行排序。

```
SearchQuery searchQuery = new SearchQuery();
searchQuery.setSort(new Sort(Arrays.asList(new ScoreSort())));
```

- PrimaryKeySort

  按照主键进行排序。

```
SearchQuery searchQuery = new SearchQuery();
searchQuery.setSort(new Sort(Arrays.asList(new PrimaryKeySort()))); //正序。
//searchQuery.setSort(new Sort(Arrays.asList(new PrimaryKeySort(SortOrder.DESC)))); //逆序
。
```

- FieldSort

  按照某列的值进行排序。

```
SearchQuery searchQuery = new SearchQuery();
searchQuery.setSort(new Sort(Arrays.asList(new FieldSort("col", SortOrder.ASC))));
```

  先按照某列的值进行排序，再按照另一列的值进行排序。

```
SearchQuery searchQuery = new SearchQuery();
searchQuery.setSort(new Sort(Arrays.asList(
    new FieldSort("col1", SortOrder.ASC), new FieldSort("col2", SortOrder.ASC))));
```

- GeoDistanceSort

  根据地理点距离进行排序。

```
SearchQuery searchQuery = new SearchQuery();
//geo列为Geopoint类型，按照此列的值距离"0,0"点的距离进行排序。
Sort.Sorter sorter = new GeoDistanceSort("geo", Arrays.asList("0, 0"));
searchQuery.setSort(new Sort(Arrays.asList(sorter)));
```

翻页方式

在获取返回结果时，可以使用limit和offset或者使用token进行翻页。

- 使用limit和offset进行翻页

  当需要获取的返回结果行数小于50000行时，可以使用limit和offset进行翻页，即limit+offset<=50000，其中limit的最大值为100。

  > **说明** 如果需要提高limit的上限，请参见如何将多元索引Search接口查询数据的limit提高到1000。

  如果使用此方式进行翻页时未设置limit和offset，则limit的默认值为10，offset的默认值为0。

```
SearchQuery searchQuery = new SearchQuery();
searchQuery.setQuery(new MatchAllQuery());
searchQuery.setLimit(100);
searchQuery.setOffset(100);
```

- 使用token进行翻页

  由于使用token进行翻页时翻页深度无限制，当需要进行深度翻页时，推荐使用token进行翻页。

  当符合查询条件的数据未读取完时，服务端会返回nextToken，此时可以使用nextToken继续读取后面的数据。

  使用token进行翻页时默认只能向后翻页。由于在一次查询的翻页过程中token长期有效，您可以通过缓存并使用之前的token实现向前翻页。

  > **注意** 如果需要持久化nextToken或者传输nextToken给前端页面，您可以使用Base64编码将nextToken编码为String进行保存和传输。token本身不是字符串，直接使用 `new String(nextToken)` 将token编码为String会造成token信息丢失。

  使用token翻页后的排序方式和上一次请求的一致，无论是系统默认使用IndexSort还是自定义排序，因此设置了token不能再设置Sort。另外使用token后不能设置offset，只能依次往后读取，即无法跳页。

  > **注意** 由于含有Nested类型字段的多元索引不支持索引预排序，如果使用含有Nested类型字段的多元索引查询数据且需要翻页，则必须在查询条件中指定数据返回的排序方式，否则当符合查询条件的数据未读取完时，服务端不会返回nextToken。

```
private static void readMoreRowsWithToken(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    searchQuery.setQuery(new MatchAllQuery());
    searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    SearchResponse resp = client.search(searchRequest);
    if (!resp.isAllSuccess()) {
        throw new RuntimeException("not all success");
    }
    List<Row> rows = resp.getRows();
    while (resp.getNextToken()!=null) { // 当读取到nextToken为null时，表示读出全部数据。
        // 获取nextToken。
        byte[] nextToken = resp.getNextToken();
        {
            // 如果需要持久化nextToken或者传输nextToken给前端页面，您可以使用Base64编码将nextTo
ken编码为String进行保存和传输。
            // token本身不是字符串，直接使用new String(nextToken)将token编码为String会造成toke
n信息丢失。
            String tokenAsString = Base64.toBase64String(nextToken);
            // 将String解码为byte。
            byte[] tokenAsByte = Base64.fromBase64String(tokenAsString);
        }
        // 将token设置到下一次请求中。
        searchRequest.getSearchQuery().setToken(nextToken);
        resp = client.search(searchRequest);
        if (!resp.isAllSuccess()) {
            throw new RuntimeException("not all success");
        }
        rows.addAll(resp.getRows());
    }
    System.out.println("RowSize: " + rows.size());
    System.out.println("TotalCount: " + resp.getTotalCount());// 打印匹配到的总行数，非返回行
数。
}
```

7.6.9. 分词

为Text类型的字段设置分词类型后，系统会将可分词类型的内容根据设定的分词类型分成多个词。非Text类型的字段不能设置分词类型。

对于Text类型字段，常用于匹配查询（MatchQuery）和短语匹配查询（MatchPhraseQuery），少部分场景也会用到精确查询（TermQuery）、多词精确查询（TermsQuery）、前缀查询（PrefixQuery）、通配符查询（WildcardQuery）等。

分词类型

目前支持单字分词、分隔符分词、最小数量语义分词、最大数量语义分词和模糊分词5种分词类型。

所有分词类型都可以用于模糊查询场景，具体如何选择请参见详解TableStore模糊查询。

单字分词（SingleWord）

单字分词适用于汉语、英语、日语等所有语言文字。Text类型字段的默认分词类型为单字分词。

设置分词类型为单字分词后，具体的分词方式如下：

- 汉语会按照“字”切分。例如“杭州”会切分成“杭”和“州”，通过MatchQuery或MatchPhraseQuery查询“杭”可以查询到含有“杭州”内容的数据。
- 英文字母或数字会按照空格或标点符号切分，然后转换为小写。例如“Hang Zhou”会切分成“hang”和“zhou”，通过MatchQuery或MatchPhraseQuery查询“hang”或“HANG”或“Hang”都能查询到该行数据。
- 对于数字和英文字母连接在一起的词，例如商品型号等，也会按照空格或标点符号切分，但是数字和英文不会拆分开。例如“IPhone6”会拆分成“IPhone6”，通过MatchQuery或MatchPhraseQuery查询时，只能指定完整“iphone6”才能查询到，使用“iphone”查询不到。

单字分词的参数说明请参见下表。

参数	说明
caseSensitive	是否大小写敏感。默认是false，此时所有英文字母会转换为小写。 如果不需要系统自动将英文字母转换为小写字母，需要保持大小写敏感，可以设置caseSensitive为true。
delimitWord	对于英文和数字连接在一起的单词，是否分割英文和数字。默认是false。 如果需要将数字和英文拆分开，可以设置delimitWord参数为true，此时“iphone6”会被拆分成“iphone”和“6”。

分隔符分词（Split）

表格存储提供了基于通用词典的分词，但是有些特殊行业需要一些自定义的辞典做分词，为了解决此问题，表格存储提供了分隔符分词，也叫自定义分词，用户先按照自己的方式分词，再按照特定分隔符分隔后写入表格存储。

分隔符分词适用于汉语、英语、日语等所有语言文字。

设置分词类型为分隔符分词后，会按照分隔符delimiter指定值切分字段值。例如字段值为“羽毛球,乒乓球,说唱”，分隔符delimeter为英文逗号（,），则会切分为“羽毛球”、“乒乓球”和“说唱”并建立索引，通过MatchQuery或MatchPhraseQuery查询“羽毛球”、“乒乓球”、“说唱”或“羽毛球,乒乓球”均可以查询到该行数据。

分隔符分词的参数说明请参见下表。

参数	说明
delimiter	分隔符，默认是空白字符，可以自定义分隔符。 - 创建多元索引时，字段分词配置中的分隔符必须和写入数据时的分隔符保持一致，否则可能会查询不到数据。 - 当自定义的分隔符为特殊字符井号（#）时，字段分词配置中的分隔符请使用转义字符 `\` 表示，例如 `\#`。

最小数量语义分词（MinWord）

最小数量语义分词适用于汉语，一般应用于全文检索场景。

设置分词类型为最小数量语义分词后，系统按照语义对Text字段内容分词时，会将Text字段内容切分成最小数量的语义词。例如“梨花茶”会切分成“梨”和“花茶”，切分后的结果没有重合。再例如“中华人民共和国”会被切分成“中华人民共和国”。

最大数量语义分词（MaxWord）

最大数量语义分词适用于汉语，一般应用于全文检索场景。

设置分词类型为最大数量语义分词后，系统会尽量多的分出语义词，不同语义词之间会有重叠，总长度累加后会大于原文长度，索引大小也会膨胀。例如“梨花茶”会切分成“梨花”和“花茶”，切分后的结果没有重合。再例如“中华人民共和国”会被切分成“中华人民共和国”、“中华人民”、“中华”、“华人”、“人民共和国”、“人民”、“共和国”、“共和”和“国”。

此种分词类型的分词后结果更多，查询时命中的概率更大，但是索引大小会有比较大的膨胀。适合使用MatchQuery而非MatchPhraseQuery查询，如果使用MatchPhraseQuery查询，由于查询关键词中也会按照同样的方式分词，那么位置信息会重叠，就有可能导致搜索不到。

模糊分词（Fuzzy）

模糊分词适用于汉语、英语、日语等所有语言文字，一般应用于文本内容较短场景，例如标题、电影名称、书名、文件名、目录名等。

模糊分词可用于以很低的延迟返回结果，比WildcardQuery性能更好，但是索引大小会有一定膨胀。

设置分词类型为模糊分词后，系统对文本内容进行Ngram分词，结果介于minChars和maxChars之间。例如用在下拉提示等功能中。

模糊分词会将内容全部转换为小写存储，因此查询时大小写不敏感，类似于SQL中的like。

要实现模糊查询时，您必须在进行模糊分词的列上使用MatchPhraseQuery，而不能使用其他Query。如果对该列有多种查询需求，请使用虚拟列功能。关于虚拟列的具体操作，请参见虚拟列。

- 限制
  - 当Text字段的分词类型为模糊分词时，长度不能超过1024字符。如果超过长度限制，系统会将超过的字符截断丢弃，只保留前1024个字符。
  - 为了防止索引数据量过度膨胀，最大和最小字符切分单元的差值（maxChars-minChars）不能超过6。
- 参数

参数	说明
minChars	最小字符切分单元，即切分的字符组合中字符数量必须大于等于此值，默认值为1。
maxChars	最大字符切分单元，即切分的字符组合中字符数量必须小于等于此值，默认值为7。

分词类型对比

从几个关键维度对不同分词的比较信息请参见下表。

对比项	单字分词	分隔符分词	最小数量语义分词	最大数量语义分词	模糊分词
索引膨胀	小	小	小	中	大
相关性影响	弱	弱	中	较强	较强
适用语言	所有	所有	汉语	汉语	所有
长度限制	无	无	无	无	1024字符

对比项	单字分词	分隔符分词	最小数量语义分词	最大数量语义分词	模糊分词
召回率	高	低	低	中	中

示例

通过如下示例查看各种分词器在索引时和查询时的分词效果。

索引时，分词字符串的值会被切分并建立索引；查询时，查询关键词也被切分，并尝试匹配之前建立的索引。

- 字段值为“中华人民共和国-国歌”，使用各类型分词器后索引的切词效果。

类型	参数	索引时分词
单字分词（SingleWord）	默认	“中”，“华”，“人”，“民”，“共”，“和”，“国”，“歌”
分隔符分词（Split）	delimiter: "-"	“中华人民共和国”，“国歌”
最小数量语义分词（MinWord）	默认	“中华人民共和国”，“国歌”
最大数据量语义分词（MaxWord）	默认	“中华人民共和国”，“中华人民”，“中华”，“华人”，“人民共和国”，“人民”，“共和国”，“共和”，“国”，“国歌”
模糊分词（Fuzzy）	minChars:1, maxChars:3	“中”，“中华”，“中华人”，“华”，“华人”，“华人民”，“人”，“人民”，“人民共”，“民”，“民共”，“民共和”，“共”，“共和”，“共和国”，“和”，“和国”，“国”，“国歌”，“歌”

- 字段值为“中华人民共和国-国歌”，使用各类型分词器后查询的切词效果。

类型	参数	查询时分词
单字分词（SingleWord）	默认	“中”，“华”，“人”，“民”，“共”，“和”，“国”，“歌”
分隔符分词（Split）	delimiter: "-"	“中华人民共和国”，“国歌”
最小数量语义分词（MinWord）	默认	“中华人民共和国”，“国歌”
最大数据量语义分词（MaxWord）	默认	“中华人民共和国”，“中华人民”，“中华”，“华人”，“人民共和国”，“人民”，“共和国”，“共和”，“国”，“国歌”

类型	参数	查询时分词
模糊分词（Fuzzy）	默认（minChars:1, maxChars:7）	“中华人民共和国”，“国歌”

7.6.10. 全匹配查询

MatchAllQuery可以匹配所有行，常用于查询表中数据总行数，或者随机返回几条数据。

接口

全匹配查询的接口为Search或者ParallelScan，具体的Query类型为MatchAllQuery。

使用

您可以使用如下语言的SDK实现全匹配查询。

- Java SDK：全匹配查询
- Go SDK：全匹配查询
- Python SDK：全匹配查询
- Node.js SDK：全匹配查询
- .NET SDK：全匹配查询
- PHP SDK：全匹配查询

参数

参数	说明
query	设置查询类型为MatchAllQuery。
tableName	数据表名称。
indexName	多元索引名称。
limit	本次查询需要返回的最大数量。 如果要随机获取几行数据，请设置limit为正整数。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。
columnsToGet	是否返回所有列。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。

示例

```
/**
 * 通过MatchAllQuery查询表中数据的总行数。
 * @param client
 */
private static void matchAllQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    /**
    * 设置查询类型为MatchAllQuery。
    */
    searchQuery.setQuery(new MatchAllQuery());
    /**
     * MatchAllQuery结果中的TotalCount可以表示表中数据的总行数。
     * 如果要随机获取几行数据，请设置limit为正整数。
     * 如果只为了获取行数，但不需要具体数据，可以设置limit=0，即不返回任意一行数据。
     */
    searchQuery.setLimit(0);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME, searchQuery);
    /**
    * 设置返回匹配的总行数。
    */
    searchQuery.setGetTotalCount(true);
    SearchResponse resp = client.search(searchRequest);
    /**
     * 判断返回的结果是否完整，当isAllSuccess为false时，表示可能存在部分节点查询失败，返回的是部分数
据。
     */
    if (!resp.isAllSuccess()) {
        System.out.println("NotAllSuccess!");
    }
    System.out.println("IsAllSuccess: " + resp.isAllSuccess());
    System.out.println("TotalCount: " + resp.getTotalCount()); //打印总行数。
    System.out.println(resp.getRequestId());
}
```

7.6.11. 匹配查询

MatchQuery采用近似匹配的方式查询表中的数据。对Text类型的列值和查询关键词会先按照设置好的分词器做切分，然后按照切分好后的词去查询。对于进行模糊分词的列，建议使用MatchPhraseQuery实现高性能的模糊查询。

场景

匹配查询一般应用于全文检索场景，可应用于Text类型。例如某一行数据的title列的值是“杭州西湖风景区”，使用单字分词，如果MatchQuery中的查询词是“湖风”，则可以匹配到该行数据。

接口

匹配查询的接口为Search或者ParallelScan，具体的Query类型为MatchQuery。

使用

您可以使用如下语言的SDK实现匹配查询。

- Java SDK：匹配查询
- Go SDK：匹配查询
- Python SDK：匹配查询
- Node.js SDK：匹配查询
- .NET SDK：匹配查询
- PHP SDK：匹配查询

参数

参数	说明
fieldName	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如当要匹配的列为Text类型时，分词类型为单字分词，则查询词为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
query	设置查询类型为matchQuery。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。
minimumShouldMatch	最小匹配个数。 只有当某一行数据的fieldName列的值中至少包括最小匹配个数的词时，才会返回该行数据。 **说明** minimumShouldMatch需要与逻辑运算符OR配合使用。
operator	逻辑运算符。默认为OR，表示当分词后的多个词只要有部分匹配时，则行数据满足查询条件。 如果设置operator为AND，则只有分词后的所有词都在列值中时，才表示行数据满足查询条件。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。

参数	说明
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列的值能够匹配"hangzhou"的数据，返回匹配到的总行数和一些匹配成功的行。
 * @param client
 */
private static void matchQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    MatchQuery matchQuery = new MatchQuery(); //设置查询类型为MatchQuery。
    matchQuery.setFieldName("Col_Keyword"); //设置要匹配的列。
    matchQuery.setText("hangzhou"); //设置要匹配的值。
    searchQuery.setQuery(matchQuery);
    searchQuery.setOffset(0); //设置offset为0。
    searchQuery.setLimit(20); //设置limit为20，表示最多返回20行数据。
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.12. 短语匹配查询

类似于MatchQuery，但是分词后多个词的位置关系会被考虑，只有分词后的多个词在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。如果查询列的分词类型为模糊分词，则使用MatchPhraseQuery可以实现比WildcardQuery更快的模糊查询。

例如字段值是“杭州西湖风景区”，Query中是“杭州风景区”，如果是MatchQuery，则可以匹配到该行数据，但是如果是MatchPhraseQuery，则不能匹配到该行数据，因为“杭州”和“风景区”在Query中的距离是0，但是在行数据中的距离是2（西湖两个字导致间隔距离是2）。

接口

短语匹配查询的接口为Search或者ParallelScan，具体的Query类型为MatchPhraseQuery。

使用

您可以使用如下语言的SDK实现短语匹配查询。

- Java SDK：短语匹配查询
- Go SDK：短语匹配查询
- Python SDK：短语匹配查询
- Node.js SDK：短语匹配查询
- .NET SDK：短语匹配查询
- PHP SDK：短语匹配查询

参数

参数	说明
fieldName	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如查询的值为“this is”，可以匹配到“..., this is tablestore”、“this is a table”，但是无法匹配到“this table is ...”以及“is this a table”。
query	设置查询类型为matchPhraseQuery。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Text列的值能够匹配"hangzhou shanghai"的数据，匹配条件为短语匹配（要求短语完整的按照顺序匹配），返回匹配到的总行数和一些匹配成功的行。
 * @param client
 */
private static void matchPhraseQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    MatchPhraseQuery matchPhraseQuery = new MatchPhraseQuery(); //设置查询类型为MatchPhraseQuery。
    matchPhraseQuery.setFieldName("Col_Text"); //设置要匹配的列。
    matchPhraseQuery.setText("hangzhou shanghai"); //设置要匹配的值。
    searchQuery.setQuery(matchPhraseQuery);
    searchQuery.setOffset(0); //设置offset为0。
    searchQuery.setLimit(20); //设置limit为20，表示最多返回20行数据。
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.13. 精确查询

TermQuery采用完整精确匹配的方式查询表中的数据，类似于字符串匹配。对于Text类型字段，只要分词后有词条可以精确匹配即可。

接口

精确查询的接口为Search或者ParallelScan，具体的Query类型为TermQuery。

使用

您可以使用如下语言的SDK实现精确查询。

- Java SDK：精确查询
- Go SDK：精确查询
- Python SDK：精确查询
- Node.js SDK：精确查询
- .NET SDK：精确查询
- PHP SDK：精确查询

参数

参数	说明
query	设置查询类型为TermQuery。
fieldName	要匹配的字段。
term	查询关键词，即要匹配的值。 该词不会被分词，会被当做完整词去匹配。 对于Text类型字段，只要分词后有词条可以精确匹配即可。例如某个Text类型的字段，值为“tablestore is cool”，如果分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列精确匹配"hangzhou"的数据。
 * @param client
 */
private static void termQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermQuery termQuery = new TermQuery(); //设置查询类型为TermQuery。
    termQuery.setFieldName("Col_Keyword"); //设置要匹配的字段。
    termQuery.setTerm(ColumnValue.fromString("hangzhou")); //设置要匹配的值。
    searchQuery.setQuery(termQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.14. 多词精确查询

类似于TermQuery，但是TermsQuery可以指定多个查询关键词，查询匹配这些词的数据。多个查询关键词中只要有一个词精确匹配，该行数据就会被返回，等价于SQL中的In。

接口

多词精确查询的接口为Search或者ParallelScan，具体的Query类型为TermsQuery。

使用

您可以使用如下语言的SDK实现多词精确查询。

- Java SDK：多词精确查询
- Go SDK：多词精确查询
- Python SDK：多词精确查询
- Node.js SDK：多词精确查询
- PHP SDK：多词精确查询

参数

参数	说明
query	设置查询类型为TermsQuery。
fieldName	要匹配的字段。

参数	说明
terms	多个查询关键词，即要匹配的值。最多支持设置1024个查询关键字。 多个查询关键词中只要有一个词精确匹配，该行数据就会被返回。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列精确匹配"hangzhou"或"xi'an"的数据。
 * @param client
 */
private static void termQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermsQuery termsQuery = new TermsQuery(); //设置查询类型为TermsQuery。
    termsQuery.setFieldName("Col_Keyword"); //设置要匹配的字段。
    termsQuery.addTerm(ColumnValue.fromString("hangzhou")); //设置要匹配的值。
    termsQuery.addTerm(ColumnValue.fromString("xi'an")); //设置要匹配的值。
    searchQuery.setQuery(termsQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.15. 前缀查询

PrefixQuery根据前缀条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。

接口

前缀查询的接口为Search或者ParallelScan，具体的Query类型为PrefixQuery。

使用

您可以使用如下语言的SDK实现前缀查询。

- Java SDK：前缀查询
- Go SDK：前缀查询
- Python SDK：前缀查询
- Node.js SDK：前缀查询
- .NET SDK：前缀查询
- PHP SDK：前缀查询

参数

参数	说明
query	设置查询类型为PrefixQuery。
fieldName	要匹配的字段。
prefix	前缀值。 对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列中前缀为"hangzhou"的数据。
 * @param client
 */
private static void prefixQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    PrefixQuery prefixQuery = new PrefixQuery(); //设置查询类型为PrefixQuery。
    searchQuery.setGetTotalCount(true);
    prefixQuery.setFieldName("Col_Keyword");
    prefixQuery.setPrefix("hangzhou");
    searchQuery.setQuery(prefixQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
```

7.6.16. 范围查询

RangeQuery根据范围条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足范围条件即可。

接口

范围查询的接口为Search或者ParallelScan，具体的Query类型为RangeQuery。

使用

您可以使用如下语言的SDK实现范围查询。

- Java SDK：范围查询
- Go SDK：范围查询
- Python SDK：范围查询
- Node.js SDK：范围查询
- .NET SDK：范围查询
- PHP SDK：范围查询

参数

参数	说明
fieldName	要匹配的字段。

参数	说明
from	起始位置的值。 设置范围条件时，大于（>）可以使用greaterThan表示，大于等于（>=）可以使用greaterThanOrEqual表示。
to	结束位置的值。 设置范围条件时，小于（<）可以使用lessThan表示；小于等于（<=）可以使用lessThanOrEqual表示。
includeLower	结果中是否需要包括from值，类型为Boolean。
includeUpper	结果中是否需要包括to值，类型为Boolean。
query	设置查询类型为RangeQuery。
sort	按照指定方式排序，详情请参见排序和翻页。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Long列大于3的数据，结果按照Col_Long列的值逆序排序。
 * @param client
 */
private static void rangeQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    RangeQuery rangeQuery = new RangeQuery(); //设置查询类型为RangeQuery。
    rangeQuery.setFieldName("Col_Long");  //设置要匹配的字段。
    rangeQuery.greaterThan(ColumnValue.fromLong(3));  //设置该字段的范围条件为大于3。
    searchQuery.setGetTotalCount(true);
    searchQuery.setQuery(rangeQuery);
    //设置按照Col_Long列逆序排序。
    FieldSort fieldSort = new FieldSort("Col_Long");
    fieldSort.setOrder(SortOrder.DESC);
    searchQuery.setSort(new Sort(Arrays.asList((Sort.Sorter)fieldSort)));
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.17. 通配符查询

通配符查询中，要匹配的值可以是一个带有通配符的字符串，目前支持星号（*）和问号（?）两种通配符。要匹配的值中可以用星号（*）代表任意字符序列，或者用问号（?）代表任意单个字符，且支持以星号（*）或问号（?）开头。例如查询“table*e”，可以匹配到“tablestore”。

如果查询的模式为 `*word*` （等价于SQL中的 `WHERE field_a LIKE '%word%'` ），则您可以使用性能更好的模糊查询，具体实现方法请参见模糊查询。该方案不会随数据量变大而导致性能下降。

限制

带有通配符的字符串长度不能超过32个字符。

接口

通配符查询的接口为Search或者ParallelScan，具体的Query类型为WildcardQuery。

使用

您可以使用如下语言的SDK实现通配符查询。

- Java SDK：通配符查询
- Go SDK：通配符查询
- Python SDK：通配符查询

- Node.js SDK：通配符查询
- .NET SDK：通配符查询
- PHP SDK：通配符查询

参数

参数	描述
fieldName	列名称。
value	带有通配符的字符串，字符串长度不能超过32个字符。
query	设置查询类型为WildcardQuery。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 使用通配符查询，查询表中Col_Keyword列的值中匹配"hang*u"的数据。
 * @param client
 */
private static void wildcardQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    WildcardQuery wildcardQuery = new WildcardQuery(); //设置查询类型为WildcardQuery。
    wildcardQuery.setFieldName("Col_Keyword");
    wildcardQuery.setValue("hang*u"); //wildcardQuery支持通配符。
    searchQuery.setQuery(wildcardQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.18. 多条件组合查询

BoolQuery查询条件包含一个或者多个子查询条件，根据子查询条件来判断一行数据是否满足查询条件。每个子查询条件可以是任意一种Query类型，包括BoolQuery。

接口

多条件组合查询的接口为Search或者ParallelScan，具体的Query类型为BoolQuery。

使用

您可以使用如下语言的SDK实现多条件组合查询。

- Java SDK：多条件组合查询
- Go SDK：多条件组合查询
- Python SDK：多条件组合查询
- Node.js SDK：多条件组合查询
- .NET SDK：多条件组合查询
- PHP SDK：多条件组合查询

参数

参数	说明
mustQueries	多个Query列表，行数据必须满足所有的子查询条件才算匹配，等价于And操作符。

参数	说明
mustNotQueries	多个Query列表，行数据必须不能满足任何的子查询条件才算匹配，等价于Not操作符。
filterQueries	多个Query列表，行数据必须满足所有的子filter才算匹配，filter类似于query，区别是filter不会根据满足的filterQueries个数进行相关性算分。
shouldQueries	多个Query列表，可以满足，也可以不满足，等价于Or操作符。 一行数据应该至少满足shouldQueries子查询条件的最小匹配个数才算匹配。 如果满足的shouldQueries子查询条件个数越多，则整体的相关性分数更高。
minimumShouldMatch	shouldQueries子查询条件的最小匹配个数。当同级没有其他Query，只有shouldQueries时，默认值为1；当同级已有其他Query，例如mustQueries、mustNotQueries和filterQueries时，默认值为0。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

- 示例1

  通过BoolQuery进行And条件查询。

```
/**
 * 通过BoolQuery进行And条件查询。
 * @param client
 */
public static void andQuery(SyncClient client){
    /**
     * 查询条件一：RangeQuery，Col_Long的列值大于3。
     */
    RangeQuery rangeQuery = new RangeQuery();
    rangeQuery.setFieldName("Col_Long");
    rangeQuery.greaterThan(ColumnValue.fromLong(3));
    /**
     * 查询条件二：MatchQuery，Col_Keyword的列值要匹配"hangzhou"。
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * 构造一个BoolQuery，设置查询条件为必须同时满足"查询条件一"和"查询条件二"。
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustQueries(Arrays.asList(rangeQuery, matchQuery));
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键
列。
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); //设置为返回所有列。
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列
。
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse resp = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，
非返回行数。
        System.out.println("Row: " + resp.getRows());
    }
}
```

- 示例2

  通过BoolQuery进行Or条件查询。

```
/**
 * 通过BoolQuery进行Or条件查询。
 * @param client
 */
public static void orQuery(SyncClient client) {
    /**
     * 查询条件一：RangeQuery，Col_Long的列值大于3。
     */
    RangeQuery rangeQuery = new RangeQuery();
    rangeQuery.setFieldName("Col_Long");
    rangeQuery.greaterThan(ColumnValue.fromLong(3));
    /**
     * 查询条件二：MatchQuery，Col_Keyword的列值要匹配"hangzhou"。
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
    /**
     * 构造一个BoolQuery，设置查询条件为至少满足"条件一"和"条件二"中的一个条件。
     */
    BoolQuery boolQuery = new BoolQuery();
    boolQuery.setShouldQueries(Arrays.asList(rangeQuery, matchQuery));
    boolQuery.setMinimumShouldMatch(1); //设置至少满足一个条件。
    searchQuery.setQuery(boolQuery);
    //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返
回行数
    System.out.println("Row: " + resp.getRows());
    }
}
```

- 示例3

  通过BoolQuery进行Not条件查询。

```
/**
 * 通过BoolQuery进行Not条件查询。
 * @param client
 */
public static void notQuery(SyncClient client) {
    /**
     * 查询条件一：MatchQuery，Col_Keyword的列值要匹配"hangzhou"。
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * 构造一个BoolQuery，设置查询条件为不满足"查询条件一"。
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustNotQueries(Arrays.asList(matchQuery));
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键
列。
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); //设置为返回所有列。
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列
。
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse resp = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，
非返回行数。
        System.out.println("Row: " + resp.getRows());
    }
}
```

- 示例4

  含有子BoolQuery的组合查询。使用多条件组合查询实现（col2<4 or col3<5） or （col2 = 4 and （col3 = 5 or col3 =6））， 每个and或or相当于一个BoolQuery，多个表达式的and或or就是多个BoolQuery的组合。

```
/**
 * (col2<4 or col3<5) or (col2 = 4 and (col3 = 5 or col3 =6))
 * 使用多条件组合查询上述表达式，每个and或or相当于一个BoolQuery，多个表达式的and或or就是多个BoolQu
ery的组合。
 * @param client
*/
private static void boolQuery2(SyncClient client){
        //条件1为col2<4。
        RangeQuery rangeQuery1 = new RangeQuery();
        rangeQuery1.setFieldName("col2");
        rangeQuery1.lessThan(ColumnValue.fromLong(4));
        //条件2为col3<5。
        RangeQuery rangeQuery2 = new RangeQuery();
```

```
        RangeQuery rangeQuery2 = new RangeQuery();
        rangeQuery2.setFieldName("col3");
        rangeQuery2.lessThan(ColumnValue.fromLong(5));
        //条件3为col2=4。
        TermQuery termQuery = new TermQuery();
        termQuery.setFieldName("col2");
        termQuery.setTerm(ColumnValue.fromLong(4));
        //条件4为col3 = 5 or col3 = 6。
        TermsQuery termsQuery = new TermsQuery();
        termsQuery.setFieldName("col3");
        termsQuery.addTerm(ColumnValue.fromLong(5));
        termsQuery.addTerm(ColumnValue.fromLong(6));
        SearchQuery searchQuery = new SearchQuery();
        List<Query> queryList1 = new ArrayList<>();
        queryList1.add(rangeQuery1);
        queryList1.add(rangeQuery2);
        //组合条件1为col2<4 OR col3<5。
        BoolQuery boolQuery1 = new BoolQuery();
        boolQuery1.setShouldQueries(queryList1);
        //组合条件2为col2=4 and (col3=5 or col3=6)。
        List<Query> queryList2 = new ArrayList<>();
        queryList2.add(termQuery);
        queryList2.add(termsQuery);
        BoolQuery boolQuery2 = new BoolQuery();
        boolQuery2.setMustQueries(queryList2);
        //总组合条件为(col2<4 OR col3<5) or (col2=4 and (col3=5 or col3=6))。
        List<Query> queryList3 = new ArrayList<>();
        queryList3.add(boolQuery1);
        queryList3.add(boolQuery2);
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setShouldQueries(queryList3);
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键
列。
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); //设置为返回所有列。
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列
。
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse response = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，
非返回行数。
        System.out.println(response.getRows());
    }
```

7.6.19. 嵌套类型查询

NestedQuery用于查询嵌套类型字段中子行的数据。嵌套类型不能直接查询，需要通过NestedQuery包装，NestedQuery中需要指定嵌套类型字段的路径和一个子查询，其中子查询可以是任意Query类型。

> 说明
> - 嵌套类型查询只能查询嵌套类型的列。
> - 在一个请求中可以同时查询普通列和嵌套类型的列。嵌套类型的更多信息请参见嵌套类型。

接口

嵌套类型查询的接口为Search或者ParallelScan，具体的Query类型为NestedQuery。

使用

您可以使用如下语言的SDK实现嵌套类型查询。

- Java SDK：嵌套类型查询
- Go SDK：嵌套类型查询
- Python SDK：嵌套类型查询
- Node.js SDK：嵌套类型查询
- .NET SDK：嵌套类型查询
- PHP SDK：嵌套类型查询

参数

参数	说明
path	路径名，嵌套类型字段的树状路径。例如news.title表示嵌套类型的news字段中的title子列。
query	嵌套类型字段的子列上的查询，子列上的查询可以是任意Query类型。
scoreMode	当字段存在多个值时基于哪个值计算分数。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

使用嵌套类型查询的示例如下：

- 示例1

查询col_nested.nested_1为tablestore的数据。其中col_nested为嵌套类型字段，子行中包含nested_1和nested_2两列。

```
private static void nestedQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    NestedQuery nestedQuery = new NestedQuery(); //设置查询类型为NestedQuery。
    nestedQuery.setPath("col_nested"); //设置嵌套类型列的路径。
    TermQuery termQuery = new TermQuery(); //构造NestedQuery的子查询。
    termQuery.setFieldName("col_nested.nested_1"); //设置列名，注意带有嵌套类型列的路径。
    termQuery.setTerm(ColumnValue.fromString("tablestore")); //设置要查询的值。
    nestedQuery.setQuery(termQuery);
    nestedQuery.setScoreMode(ScoreMode.None);
    searchQuery.setQuery(nestedQuery);
    //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返
回行数。
    System.out.println("Row: " + resp.getRows());
}
```

- 示例2

  查询col_nested.nested_2.nested_2_2为tablestore的数据。其中col_nested为嵌套类型字段，col_nested的子行中包含nested_1和nested_2两列，nested_2的子行中又包含nested_2_1和nested_2_2两列。

```
private static void nestedQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    NestedQuery nestedQuery = new NestedQuery(); //设置查询类型为NestedQuery。
    nestedQuery.setPath("col_nested.nested_2"); //设置嵌套类型列的路径，即要查询字段的父路径。
    TermQuery termQuery = new TermQuery(); //构造NestedQuery的子查询。
    termQuery.setFieldName("col_nested.nested_2.nested_2_2"); //设置列名，即要查询字段的完整路径。
    termQuery.setTerm(ColumnValue.fromString("tablestore")); //设置要查询的值。
    nestedQuery.setQuery(termQuery);
    nestedQuery.setScoreMode(ScoreMode.None);
    searchQuery.setQuery(nestedQuery);
    //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.20. 地理距离查询

GeoDistanceQuery根据一个中心点和距离条件查询表中的数据，当一个地理位置点到指定的中心点的距离不超过指定的值时，满足查询条件。

参数

参数	说明
fieldName	列名，类型为Geopoint。
centerPoint	中心地理坐标点，是一个经纬度值。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围为[-180,+180]。例如 `35.8,-45.91` 。
distanceInMeter	距离中心点的距离。类型为Double。单位为米。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
query	多元索引的查询语句。设置查询类型为GeoDistanceQuery。
tableName	数据表名称。

参数	说明
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

查询表中Col_GeoPoint列的值距离中心点不超过一定距离的数据。

```
public static void geoDistanceQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoDistanceQuery geoDistanceQuery = new GeoDistanceQuery();  //设置查询类型为GeoDistanceQuery。
    geoDistanceQuery.setFieldName("Col_GeoPoint");
    geoDistanceQuery.setCenterPoint("5,5"); //设置中心点。
    geoDistanceQuery.setDistanceInMeter(10000); //设置条件为到中心点的距离不超过10000米。
    searchQuery.setQuery(geoDistanceQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.21. 地理长方形范围查询

GeoBoundingBoxQuery根据一个长方形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在给出的长方形范围内时满足查询条件。

参数

参数	说明
fieldName	列名，类型为Geopoint。
topLeft	长方形框的左上角的坐标。

参数	说明
bottomRight	长方形框的右下角的坐标，通过左上角和右下角就可以确定一个唯一的长方形。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围[-180,+180]。例如 `35.8,-45.91` 。
query	多元索引的查询语句。设置查询类型为GeoBoundingBoxQuery。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

Col_GeoPoint是GeoPoint类型，查询表中Col_GeoPoint列的值在左上角为"10,0"，右下角为"0,10"的长方形范围内的数据。

```
public static void geoBoundingBoxQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoBoundingBoxQuery geoBoundingBoxQuery = new GeoBoundingBoxQuery(); //设置查询类型为GeoBoundingBoxQuery。
    geoBoundingBoxQuery.setFieldName("Col_GeoPoint"); //设置列名。
    geoBoundingBoxQuery.setTopLeft("10,0"); //设置长方形左上角。
    geoBoundingBoxQuery.setBottomRight("0,10"); //设置长方形右下角。
    searchQuery.setQuery(geoBoundingBoxQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.22. 地理多边形范围查询

GeoPolygonQuery根据一个多边形范围条件查询表中的数据，当一个地理位置点落在指定的多边形范围内时满足查询条件。

参数

参数	说明
fieldName	列名，类型为Geopoint。
points	组成多边形的距离坐标点。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围为[-180,+180]。例如 `35.8,-45.91` 。
query	多元索引的查询语句。设置查询类型为GeoPolygonQuery。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

查询表中Col_GeoPoint列的值在一个给定多边形范围内的数据。

```
public static void geoPolygonQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoPolygonQuery geoPolygonQuery = new GeoPolygonQuery();  //设置查询类型为GeoPolygonQuery
。
    geoPolygonQuery.setFieldName("Col_GeoPoint");
    geoPolygonQuery.setPoints(Arrays.asList("0,0","5,5","5,0")); //设置多边形的顶点。
    searchQuery.setQuery(geoPolygonQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.23. 列存在性查询

ExistsQuery也叫NULL查询或者空值查询，一般用于判断稀疏数据中某一行的某一列是否存在。例如查询所有数据中address列不为空的行。

> **说明**
>
> - 如果需要查询某一列为空，则ExistsQuery需要和BoolQuery中的mustNotQueries结合使用。
> - 以下情况会认为某一列不存在，以city列为例说明。
>   - city列在多元索引中的数据类型为keyword（或其他基础类型），如果数据表中某行数据不存在city列，则多元索引认为该行数据的city列不存在。
>   - city列在多元索引中的数据类型为keyword（或其他基础类型）数组，如果数据表中某行数据的city列为空数组，即"city" = "[]"，则多元索引认为该行数据的city列不存在。

接口

列存在性查询的接口为Search或者ParallelScan，具体的Query类型为ExistsQuery。

使用

您可以使用如下语言的SDK实现列存在性查询。

- Java SDK：列存在性查询
- Go SDK：列存在性查询
- Python SDK：列存在性查询
- PHP SDK：列存在性查询

参数

参数	说明
fieldName	列名。
query	设置查询类型为ExistsQuery。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 使用列存在查询，查询表中address列的值不为空的数据。
 * @param client
 */
private static void existQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    ExistsQuery existQuery = new ExistsQuery(); //设置查询类型为ExistsQuery。
    existQuery.setFieldName("address");
    searchQuery.setQuery(existQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

7.6.24. 折叠（去重）

当数据查询的结果中含有某种类型的数据较多时，可以使用折叠（Collapse）功能按照某一列对结果集做折叠，使对应类型的数据在结果展示中只出现一次，保证结果展示中类型的多样性。

折叠功能可以在大部分场景下实现去重（Distinct）功能，相当于按照折叠列做去重，但是只支持应用于整型、浮点数和Keyword类型的列，不支持数组类型的列，且只能返回排序后的前10000个结果。

注意事项

- 折叠功能只能使用offset+limit方式翻页，不能使用token方式。
- 对结果集同时使用统计聚合与折叠功能时，统计聚合功能只作用于使用折叠功能前的结果集。
- 使用折叠功能后，返回的总分组数取决于offset+limit的最大值，目前支持返回的总分组数最大为10000。
- 执行结果中返回的总行数是使用折叠功能前的匹配行数，使用折叠功能后的总分组数无法获取。

接口

折叠（去重）功能的接口为Search，通过collapse参数实现。

使用

您可以使用如下语言的SDK实现折叠（去重）功能。

- Java SDK：折叠（去重）
- Go SDK：折叠（去重）
- PHP SDK：折叠（去重）

参数

参数	说明
query	可以是任意Query类型。
collapse	折叠参数设置，包含fieldName设置。 fieldName：列名，按该列对结果集做折叠，只支持应用于整型、浮点数和Keyword类型的列，不支持数组类型的列。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
private static void UseCollapse(SyncClient client){
    SearchQuery searchQuery = new SearchQuery(); //构造查询条件。
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("user_id");
    matchQuery.setText("00002");
    searchQuery.setQuery(matchQuery);
    Collapse collapse = new Collapse("product_name"); //根据"product_name"列对结果集做折叠。
    searchQuery.setCollapse(collapse);
    searchQuery.setOffset(1000);
    searchQuery.setLimit(20);
    //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);//设置数据表名称和多元索引名称。      //通过设置columnsToGet参数可以指定返回的列或返回所有列
，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse response = client.search(searchRequest);
    //System.out.println(response.getTotalCount());
    //System.out.println(response.getRows().size()); //根据"product_name"列的产品种类返回个数
。
    System.out.println(response.getRows()); //根据"product_name"列的产品种类返回相应产品名称。
}
```

7.6.25. 统计聚合

使用统计聚合功能可以实现求最小值、求最大值、求和、求平均值、统计行数、去重统计行数、百分位统计、按字段值分组、按范围分组、按地理位置分组、按过滤条件分组、直方图统计、获取统计聚合分组内的行、嵌套查询等；同时多个统计聚合功能可以组合使用，满足复杂的查询需求。

最小值

返回一个字段中的最小值，类似于SQL中的min。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 *  商品库中有每一种商品的价格，求产地为浙江省的商品中，价格最低的商品价格是多少。
 *  等效的SQL语句是SELECT min(column_price) FROM product where place_of_production="浙江省"
。
 */
public void min(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production","浙江省"))
                .limit(0) //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性
能。
                .addAggregation(AggregationBuilders.min("min_agg_1", "column_price").miss
ing(100))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("min_agg_1"
).getValue());
}
```

最大值

返回一个字段中的最大值，类似于SQL中的max。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的价格，求产地为浙江省的商品中，价格最高的商品价格是多少。
 * 等效的SQL语句是SELECT max(column_price) FROM product where place_of_production="浙江省"。
 */
public void max(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "浙江省"))
                .limit(0) //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
                .addAggregation(AggregationBuilders.max("max_agg_1", "column_price").missing(0))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsMaxAggregationResult("max_agg_1").getValue());
}
```

和

返回数值字段的总数，类似于SQL中的sum。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的售出数量，求产地为浙江省的商品中，一共售出了多少件商品。如果某一件商品没有该值，默认售出了10件。
 * 等效的SQL语句是SELECT sum(column_price) FROM product where place_of_production="浙江省"。
 */
public void sum(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "浙江省"))
                .limit(0) //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
                .addAggregation(AggregationBuilders.sum("sum_agg_1", "column_number").missing(10))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("sum_agg_1").getValue());
}
```

平均值

返回数值字段的平均值，类似于SQL中的avg。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的售出数量，求产地为浙江省的商品中，平均价格是多少。
 * 等效的SQL语句是SELECT avg(column_price) FROM product where place_of_production="浙江省"。
 */
public void avg(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "浙江省"))
                .limit(0) //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
                .addAggregation(AggregationBuilders.avg("avg_agg_1", "column_number"))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsAvgAggregationResult("avg_agg_1"
).getValue());
}
```

统计行数

返回指定字段值的数量或者表中数据总行数，类似于SQL中的count。

> **说明** 通过如下方式可以统计表中数据总行数或者某个query匹配的行数。
>
> - 使用统计聚合的count功能，在请求中设置count(*)。
> - 使用query功能的匹配行数，在query中设置setGetTotalCount(true)；如果需要统计表中数据总行数，则使用MatchAllQuery。
>
> 如果需要获取表中数据某列出现的次数，则使用count（列名），可应用于稀疏列的场景。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long、Double、Boolean、Keyword和Geo_point类型。

- 示例

```
/**
 * 商家库中有每一种商家的惩罚记录，求浙江省的商家中，有惩罚记录的一共有多少个商家。如果商家没有惩罚记录，则商家信息中不存在该字段。
 * 等效的SQL语句是SELECT count(column_history) FROM product where place_of_production="浙江省"。
 */
public void count(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place", "浙江省"))
                .limit(0)
                .addAggregation(AggregationBuilders.count("count_agg_1", "column_history"))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsCountAggregationResult("count_agg_1").getValue());
}
```

去重统计行数

返回指定字段不同值的数量，类似于SQL中的count（distinct）。

> **说明** 去重统计行数的计算结果是个近似值。
> - 当去重统计行数小于1万时，计算结果是一个精确值。
> - 当去重统计行数达到1亿时，计算结果的误差为2%左右。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long、Double、Boolean、Keyword和Geo_point类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 求所有的商品，产地一共来自多少个省份。
 * 等效的SQL语句是SELECT count(distinct column_place) FROM product。
 */
public void distinctCount(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.distinctCount("dis_count_agg_1", "col
umn_place"))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
dis_count_agg_1").getValue());
}
```

百分位统计

百分位统计常用来统计一组数据的百分位分布情况，例如在日常系统运维中统计每次请求访问的耗时情况时，需要关注系统请求耗时的P25、P50、P90、P99值等分布情况。

> **说明** 百分位统计为非精确统计，对不同百分位数值的计算精确度不同，较为极端的百分位数值更加准确，例如1%或99%的百分位数值会比50%的百分位数值准确。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
percentiles	百分位分布例如50、90、99，可根据需要设置一个或者多个百分位。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 分析系统请求耗时分位数分布情况。
 */
public  void percentilesAgg(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .addAggregation(AggregationBuilders.percentiles("percentilesAgg", "latenc
y")
                    .percentiles(Arrays.asList(25.0d, 50.0d, 99.0d))
                    .missing(1.0))
                .build())
        .build();
    //执行查询
    SearchResponse resp = client.search(searchRequest);
    //获取结果
    PercentilesAggregationResult percentilesAggregationResult = resp.getAggregationResult
s().getAsPercentilesAggregationResult(
        "percentilesAgg");
    for (PercentilesAggregationItem item : percentilesAggregationResult.getPercentilesAgg
regationItems()) {
        System.out.println("key: " + item.getKey() + " value:" + item.getValue().asDouble
());
    }
}
```

字段值分组

根据一个字段的值对查询结果进行分组，相同的字段值放到同一分组内，返回每个分组的值和该值对应的个数。

> **说明** 当分组较大时，按字段值分组可能会存在误差。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long、Double、Boolean和Keyword类型。

参数	说明
groupBySorter	分组中的item排序规则，默认按照分组中item的数量降序排序，多个排序则按照添加的顺序进行排列。支持的参数如下。 ○ 按照值的字典序升序排列 ○ 按照值的字典序降序排列 ○ 按照行数升序排列 ○ 按照行数降序排列 ○ 按照子统计聚合结果中值升序排列 ○ 按照子统计聚合结果中值降序排列
size	返回的分组数量，最大值为2000。当分组数量超过2000时，只会返回前2000个分组。
subAggregation和subGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。 ○ 场景 统计每个类别的商品数量，且统计每个类别价格的最大值和最小值。 ○ 方法 最外层的统计聚合是根据类别进行分组，再添加两个子统计聚合求价格的最大值和最小值。 ○ 结果示例 ■ 水果：5个（其中价格的最大值为15，最小值为3） ■ 洗漱用品：10个（其中价格的最大值为98，最小值为1） ■ 电子设备：3个（其中价格的最大值为8699，最小值为2300） ■ 其它：15个（其中价格的最大值为1000，最小值为80）

- 示例1

```
/**
 * 所有商品中每一个类别各有多少个，且统计每一个类别的价格最大值和最小值。
 * 返回结果举例："水果：5个（其中价格的最大值为15，最小值为3），洗漱用品：10个（其中价格的最大值为98，最小值为1），电子设备：3个（其中价格的最大值为8699，最小值为2300），
 * 其它：15个（其中价格的最大值为1000，最小值为80）"。
 */
public void groupByField(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByField("name1", "column_type")
                    .addSubAggregation(AggregationBuilders.min("subName1", "column_price"))
                    .addSubAggregation(AggregationBuilders.max("subName2", "column_price"))
                )
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    for (GroupByFieldResultItem item : resp.getGroupByResults().getAsGroupByFieldResult("name1").getGroupByFieldResultItems()) {
        //打印值。
        System.out.println(item.getKey());
        //打印个数。
        System.out.println(item.getRowCount());
        //打印价格的最小值。
        System.out.println(item.getSubAggregationResults().getAsMinAggregationResult("subName1").getValue());
        //打印价格的最大值。
        System.out.println(item.getSubAggregationResults().getAsMaxAggregationResult("subName2").getValue());
    }
}
```

- 示例2

```
    /**
     * 按照多字段分组的示例。
     * 多元索引目前不能原生支持SQL中的groupBy多字段，但是可以通过嵌套使用两个groupBy完成相似功能。
     * 等效的SQL语句是select a,d, sum(b),sum(c) from user group by a,d。
     */
    public void GroupByMultiField() {
        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .returnAllColumns(true)    //设置为false时，指定addColumnsToGet，性能会高。
```

```
            //.addColumnsToGet("col_1","col_2")
            .searchQuery(SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())    //此处相当于SQL中的where条件，可以通过Quer
yBuilders.bool()嵌套查询实现复杂的查询。
                .addGroupBy(
                    GroupByBuilders
                        .groupByField("任意唯一名字标识_1", "field_a")
                        .size(20)
                        .addSubGroupBy(
                            GroupByBuilders
                                .groupByField("任意唯一名字标识_2", "field_d")
                                .size(20)
                                .addSubAggregation(AggregationBuilders.sum("任意唯一名字标
识_3", "field_b"))
                                .addSubAggregation(AggregationBuilders.sum("任意唯一名字标
识_4", "field_c"))
                        )
                )
                .build())
            .build();
        SearchResponse response = client.search(searchRequest);
        //查询符合条件的行。
        List<Row> rows = response.getRows();
        //获取统计聚合结果。
        GroupByFieldResult groupByFieldResult1 = response.getGroupByResults().getAsGroupB
yFieldResult("任意唯一名字标识_1");
        for (GroupByFieldResultItem resultItem : groupByFieldResult1.getGroupByFieldResul
tItems()) {
            System.out.println("field_a key:" + resultItem.getKey() + " Count:" + resultI
tem.getRowCount());
            //获取子统计聚合结果。
            GroupByFieldResult subGroupByResult = resultItem.getSubGroupByResults().getAs
GroupByFieldResult("任意唯一名字标识_2");
            for (GroupByFieldResultItem item : subGroupByResult.getGroupByFieldResultItem
s()) {
                System.out.println("field_a " + resultItem.getKey() + " field_d key:" + i
tem.getKey() + " Count: " + item.getRowCount());
                double sumOf_field_b = item.getSubAggregationResults().getAsSumAggregatio
nResult("任意唯一名字标识_3").getValue();
                double sumOf_field_c = item.getSubAggregationResults().getAsSumAggregatio
nResult("任意唯一名字标识_4").getValue();
                System.out.println("sumOf_field_b:" + sumOf_field_b);
                System.out.println("sumOf_field_c:" + sumOf_field_c);
            }
        }
    }
```

- 示例3

```
    /**
     * 使用统计聚合排序的示例。
     * 使用方法：按顺序添加GroupBySorter即可，添加多个GroupBySorter时排序结果按照添加顺序生效。GroupBySorter支持升序和降序两种方式。
     * 默认排序是按照行数降序排列即GroupBySorter.rowCountSortInDesc()。
     */
    public void groupByFieldWithSort(SyncClient client) {
        //构建查询语句。
        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .searchQuery(
                SearchQuery.newBuilder()
                    .query(QueryBuilders.matchAll())
                    .limit(0)
                    .addGroupBy(GroupByBuilders
                        .groupByField("name1", "column_type")
                        //.addGroupBySorter(GroupBySorter.subAggSortInAsc("subName1")) //按照子统计聚合结果中的值升序排序。
                        .addGroupBySorter(GroupBySorter.groupKeySortInAsc())           //按照统计聚合结果中的值升序排序。
                        //.addGroupBySorter(GroupBySorter.rowCountSortInDesc())        //按照统计聚合结果中的行数降序排序。
                        .size(20)
                        .addSubAggregation(AggregationBuilders.min("subName1", "column_price"))
                        .addSubAggregation(AggregationBuilders.max("subName2", "column_price"))
                    )
                    .build())
            .build();
        //执行查询。
        SearchResponse resp = client.search(searchRequest);
    }
```

范围分组

根据一个字段的范围对查询结果进行分组，字段值在某范围内放到同一分组内，返回每个范围中相应的item个数。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
range[double_from, double_to)	分组的范围。 起始值double_from可以使用最小值Double.MIN_VALUE，结束值double_to可以使用最大值Double.MAX_VALUE。

参数	说明
subAggregation和subGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。 例如按销量分组后再按省份分组，即可获得某个销量范围内哪个省比重比较大，实现方法是GroupByRange下添加一个GroupByField。

- 示例

```
/**
 * 求商品销量时按[0，1000）、[1000，5000）、[5000，Double.MAX_VALUE）这些分组计算每个范围的销量。
 */
public void groupByRange(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByRange("name1", "column_number")
                    .addRange(0, 1000)
                    .addRange(1000, 5000)
                    .addRange(5000, Double.MAX_VALUE)
                )
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    for (GroupByRangeResultItem item : resp.getGroupByResults().getAsGroupByRangeResult("name1").getGroupByRangeResultItems()) {
        //打印个数。
        System.out.println(item.getRowCount());
    }
}
```

地理位置分组

根据距离某一个中心点的范围对查询结果进行分组，距离差值在某范围内放到同一分组内，返回每个范围中相应的item个数。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Geo_point类型。

参数	说明
origin(double lat, double lon)	起始中心点的经纬度。 double lat是起始中心点纬度，double lon是起始中心点经度。
range[double_from, double_to)	分组的范围，单位为米。 起始值double_from可以使用最小值Double.MIN_VALUE，结束值double_to可以使用最大值Double.MAX_VALUE。
subAggregation和subGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。

- 示例

```
/**
 * 求距离万达广场[0，1000）、[1000，5000）、[5000，Double.MAX_VALUE）这些范围内的人数，距离的单位为米。
 */
public void groupByGeoDistance(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByGeoDistance("name1", "column_geo_point")
                    .origin(3.1, 6.5)
                    .addRange(0, 1000)
                    .addRange(1000, 5000)
                    .addRange(5000, Double.MAX_VALUE)
                )
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    for (GroupByGeoDistanceResultItem item : resp.getGroupByResults().getAsGroupByGeoDistanceResult("name1").getGroupByGeoDistanceResultItems()) {
        //打印个数。
        System.out.println(item.getRowCount());
    }
}
```

过滤条件分组

按照过滤条件对查询结果进行分组，获取每个过滤条件匹配到的数量，返回结果的顺序和添加过滤条件的顺序一致。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
filter	过滤条件，返回结果的顺序和添加过滤条件的顺序一致。
subAggregation和subGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。

- 示例

```
/**
 * 按照过滤条件进行分组，例如添加三个过滤条件（销量大于100、产地是浙江省、描述中包含杭州关键词），然后获取每个过滤条件匹配到的数量。
 */
public void groupByFilter(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByFilter("name1")
                    .addFilter(QueryBuilders.range("number").greaterThanOrEqual(100))
                    .addFilter(QueryBuilders.term("place","浙江省"))
                    .addFilter(QueryBuilders.match("text","杭州"))
                )
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //按照过滤条件的顺序获取的统计聚合结果。
    for (GroupByFilterResultItem item : resp.getGroupByResults().getAsGroupByFilterResult
("name1").getGroupByFilterResultItems()) {
        //打印个数。
        System.out.println(item.getRowCount());
    }
}
```

直方图统计

按照指定数据间隔对查询结果进行分组，字段值在相同范围内放到同一分组内，返回每个分组的值和该值对应的个数。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
interval	统计间隔。
fieldRange[min,max]	统计范围，与interval参数配合使用限制分组的数量。`(fieldRange.max-fieldRange.min)/interval` 的值不能超过2000。
minDocCount	最小行数。当分组中的行数小于最小行数时，不会返回此分组的统计结果。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 统计不同年龄段用户数量分布情况。
 */
public static void groupByHistogram(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .addGroupBy(GroupByBuilders
                    .groupByHistogram("groupByHistogram", "age")
                    .interval(10)
                    .minDocCount(0L)
                    .addFieldRange(0, 99))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = ots.search(searchRequest);
    //获取直方图的统计聚合结果。
    GroupByHistogramResult results = resp.getGroupByResults().getAsGroupByHistogramResult
("groupByHistogram");
    for (GroupByHistogramItem item : results.getGroupByHistogramItems()) {
        System.out.println("key: " + item.getKey().asLong() + " value:" + item.getValue()
);
    }
}
```

获取统计聚合分组中的行

对查询结果进行分组后，获取每个分组内的一些行数据，可实现和MySQL中ANY_VALUE(field)类似的功能。

> **说明** 获取统计聚合分组中的行时，如果多元索引中包含嵌套类型、地理位置类型或者数组类型的字段，则返回结果中只会包含主键信息，请手动反查数据表获取所需字段。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
limit	每个分组内最多返回的数据行数，默认返回1行数据。
sort	分组内数据的排序方式。
columnsToGet	指定返回的字段，仅支持多元索引中的字段，且不支持数组、Geopoint和Nested类型的字段。

- 示例

```
/**
 * 某学校有一个活动报名表，活动报名表中包含学生姓名、班级、班主任、班长等信息，如果希望按班级进行分组，以查看每个班级的报名情况，同时获取班级的属性信息。
 * 等效的SQL语句是select className, teacher, monitor, COUNT(*) as number from table GROUP BY className。
 */
public void testTopRows(SyncClient client) {
    SearchRequest searchRequest = SearchRequest.newBuilder()
            .indexName("indexName")
            .tableName("tableName")
            .searchQuery(
                    SearchQuery.newBuilder()
                            .query(QueryBuilders.matchAll())
                            .limit(0)
                            .addGroupBy(GroupByBuilders.groupByField("groupName", "className")
                                    .size(5)
                                    .addSubAggregation(AggregationBuilders.topRows("topRowsName")
                                            .limit(1)
                                            .sort(new Sort(Arrays.asList(new FieldSort("teacher", SortOrder.DESC)))) // topRows的排序
                                    )
                            )
                            .build())
            .addColumnsToGet(Arrays.asList("teacher", "monitor"))
            .build();
    SearchResponse resp = client.search(searchRequest);
    List<GroupByFieldResultItem> items = resp.getGroupByResults().getAsGroupByFieldResult("groupName").getGroupByFieldResultItems();
    for (GroupByFieldResultItem item : items) {
        String className = item.getKey();
        long number = item.getRowCount();
        List<Row> topRows = item.getSubAggregationResults().getAsTopRowsAggregationResult("topRowsName").getRows();
        Row row = topRows.get(0);
        String teacher = row.getLatestColumn("teacher").getValue().asString();
        String monitor = row.getLatestColumn("monitor").getValue().asString();
    }
}
```

嵌套

分组类型的统计聚合功能支持嵌套，其内部可以添加子统计聚合。

主要用于在分组内再次进行统计聚合，以两层的嵌套为例：

- GroupBy+SubGroupBy：按省份分组后再按照城市分组，获取每个省份下每个城市的数据。
- GroupBy+SubAggregation：按照省份分组后再求某个指标的最大值，获取每个省的某个指标最大值。

> **说明** 为了性能、复杂度等综合考虑，嵌套的层级只开放了一定的层数。更多信息，请参见多元索引限制。

示例

```
/**
 * 嵌套的统计聚合示例。
 * 外层2个Aggregation和1个GroupByField，GroupByField中又添加了2个Aggregation和1个GroupByRange。
 */
public void subGroupBy(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .indexName("index_name")
        .tableName("table_name")
        .returnAllColumns(true)
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.match("textField", "hello"))
                .limit(10)
                .addAggregation(AggregationBuilders.min("name1", "fieldName1"))
                .addAggregation(AggregationBuilders.max("name2", "fieldName2"))
                .addGroupBy(GroupByBuilders
                    .groupByField("name3", "fieldName3")
                    .addSubAggregation(AggregationBuilders.max("subName1", "fieldName4"))
                    .addSubAggregation(AggregationBuilders.sum("subName2", "fieldName5"))
                    .addSubGroupBy(GroupByBuilders
                        .groupByRange("subName3", "fieldName6")
                        .addRange(12, 90)
                        .addRange(100, 900)
                    ))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取第一层求最小值和求最大值的统计聚合结果。
    AggregationResults aggResults = resp.getAggregationResults();
    System.out.println(aggResults.getAsMinAggregationResult("name1").getValue());
    System.out.println(aggResults.getAsMaxAggregationResult("name2").getValue());
    //获取第一层按字段值分组的统计聚合结果，并同时获取其嵌套的子统计聚合结果。
    GroupByFieldResult results = resp.getGroupByResults().getAsGroupByFieldResult("someName
1");
    for (GroupByFieldResultItem item : results.getGroupByFieldResultItems()) {
        System.out.println("数量：" + item.getRowCount());
        System.out.println("key：" + item.getKey());
        //获取子统计聚合结果。
        //打印求最大值的子统计聚合结果。
        System.out.println(item.getSubAggregationResults().getAsMaxAggregationResult("subNa
me1"));
        //打印求和的子统计聚合结果。
        System.out.println(item.getSubAggregationResults().getAsSumAggregationResult("subNa
me2"));
        //打印按范围分组的子统计聚合结果。
        GroupByRangeResult subResults = resp.getGroupByResults().getAsGroupByRangeResult("s
ubName3");
        for (GroupByRangeResultItem subItem : subResults.getGroupByRangeResultItems()) {
            System.out.println("数量：" + subItem.getRowCount());
            System.out.println("key：" + subItem.getKey());
```

```
        }
    }
}
```

多个统计聚合

多个统计聚合功能可以组合使用。

> **说明** 当多个统计聚合的复杂度较高时可能会影响响应速度。

- 示例1

```
public void multipleAggregation(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.min("name1", "long"))
                .addAggregation(AggregationBuilders.sum("name2", "long"))
                .addAggregation(AggregationBuilders.distinctCount("name3", "long"))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取求最小值的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").ge
tValue());
    //获取求和的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").ge
tValue());
    //获取去重统计行数的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
name3").getValue());
}
```

- 示例2

```
public void multipleGroupBy(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.min("name1", "long"))
                .addAggregation(AggregationBuilders.sum("name2", "long"))
                .addAggregation(AggregationBuilders.distinctCount("name3", "long"))
                .addGroupBy(GroupByBuilders.groupByField("name4", "type"))
                .addGroupBy(GroupByBuilders.groupByRange("name5", "long").addRange(1, 15)
)
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取求最小值的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").ge
tValue());
    //获取求和的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").ge
tValue());
    //获取去重统计行数的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
name3").getValue());
    //获取按字段值分组的统计聚合结果。
    for (GroupByFieldResultItem item : resp.getGroupByResults().getAsGroupByFieldResult("
name4").getGroupByFieldResultItems()) {
        //打印key。
        System.out.println(item.getKey());
        //打印个数。
        System.out.println(item.getRowCount());
    }
    //获取按范围分组的统计聚合结果。
    for (GroupByRangeResultItem item : resp.getGroupByResults().getAsGroupByRangeResult("
name5").getGroupByRangeResultItems()) {
        //打印个数。
        System.out.println(item.getRowCount());
    }
}
```

7.6.26. 并发导出数据

当使用场景中不关心整个结果集的顺序时，可以使用并发导出数据功能以更快的速度将命中的数据全部返回。

背景

多元索引中提供了Search接口，Search接口支持全功能集，包括所有的查询功能，以及排序、统计聚合等分析能力，其结果会按照指定的顺序返回。

但是在有些场景中，例如对接计算系统Spark、Presto等或者一些圈选场景，只需要使用完整的查询能力，能将命中的数据以更快的速度全部返回，不关心整个结果集的顺序。为了支持此类需求，多元索引中提供了ParallelScan接口。

> **说明** 5.6.0及其以上版本的SDK开始支持ParallelScan功能。

ParallelScan接口相对于Search接口，保留了所有的查询功能，但是舍弃了排序、统计聚合等分析功能，带来了5倍以上的性能提升，因此可以实现1分钟内上亿级别数据行的导出能力，导出能力可以水平扩展，不存在上限。

在接口的实现中，单次请求的limit限制更宽松。目前Search接口的limit最大允许100行，但是ParallelScan接口的limit最大允许2000行。同时支持多个线程一起并发请求，因此导出速度会极快。

场景

- 如果请求关心排序、统计聚合，或者是终端客户的直接查询请求，需要用Search接口。
- 如果请求不关心排序，只关心把所有符合条件的数据尽快返回，或者是计算系统（Spark、Presto等）拉取数据，可以考虑ParallelScan接口。

特点

ParallelScan接口相对于Search接口的典型特征如下：

- 结果稳定性

  ParallelScan任务是有状态的。在一个Session请求中，获取到的结果集是确定的，由发起第一次请求时的数据状态决定。如果发起第一次请求后插入了新的数据或者修改了原有的数据不会对结果集造成影响。

- 新增会话（Session）概念

  在ParallelScan系列接口中新增了Session概念。使用sessionId能够保证获取到的结果集是稳定的，具体流程如下：

  i. 通过ComputeSplits接口获取最大并发数和当前sessionId。

  ii. 通过发起多个并发ParallelScan请求读取数据，请求中需要指定当前的sessionId和当前并发ID。

  在某些不易获取sessionId的场景中，ParallelScan也支持不携带sessionId发起请求，但是不使用sessionId可能会有极小的概率导致获取到的结果集中有重复数据。

  如果在ParallelScan过程中发生网络异常、线程异常、动态Schema修改、索引切换等情况，导致ParallelScan不能继续扫描数据，服务端会返回“OTSSessionExpired”错误码，此时需要重新开始一个新的ParallelScan任务，从最开始重新拉取数据。

- 最大并发数

  ParallelScan支持的单请求的最大并发数由ComputeSplits的返回值确定。数据越多，支持的并发数越大。

  单请求指同一个查询语句，例如查询city=“杭州”的结果，如果使用Search接口，那么Search请求的返回值中会包括所有city=“杭州”的结果；如果使用ParallelScan接口且并发数是2，那么每个ParallelScan请求返回50%的结果，然后将两个并发的结果集合并在一起才是完整的结果集。

- 每次返回行数

  limit默认为2000，最大可以到2000。超过2000后，limit的变化对性能基本无影响。

- 性能

  ParallelScan接口单并发扫描数据的性能是Search接口的5倍。当增大并发数时，性能可以继续线性提高，例如8并发时仍可以继续提高4倍性能。

- 成本

  由于ParallelScan请求对资源的消耗更少，价格会更便宜，所以对于大数据量的导出类需求，强烈建议使用ParallelScan接口。

- 返回列

  只支持返回多元索引中已建立索引的列，不能返回多元索引中没有的列，可以支持的ReturnType是RETURN_ALL_INDEX或者RETURN_SPECIFIED，不支持RETURN_ALL。

  目前已支持返回数组和地理位置，但是返回的字段值会被格式化，可能和写入数据表的值不一致。例如对于数组类型，写入数据表的值为"[1,2, 3, 4]"，则通过ParallelScan接口导出的值为"[1,2,3,4]"；对于地理位置类型，写入数据表的值为 `10,50` ，则通过ParallelScan接口导出的值为 `10.0,50.0` 。

- 限制项

  同时存在的ParallelScan任务数量有一定的限制，当前为10个，后续会根据客户需求继续调整。同一个sessionId且ScanQuery相同的多个并发任务视为一个任务。一个ParallelScan任务的生命周期定义为：开始时间是第一次发出ParallelScan请求，结束时间是翻页扫描完所有数据或者请求的token失效。

接口

多并发数据导出功能涉及如下两个接口。

- ComputeSplits：获取当前ParallelScan单个请求的最大并发数。
- ParallelScan：执行具体的数据导出功能。

参数

参数		说明
tableName		数据表名称。
indexName		多元索引名称。
scanQuery	query	多元索引的查询语句。支持精确查询、模糊查询、范围查询、地理位置查询、嵌套查询等，功能和Search接口一致。
	limit	扫描数据时一次能返回的数据行数。
	maxParallel	最大并发数。请求支持的最大并发数由用户数据量决定。数据量越大，支持的并发数越多，每次任务前可以通过ComputeSplits API进行获取。
	currentParallelId	当前并发ID。取值范围为[0, maxParallel)。
	token	用于翻页功能。ParallelScan请求结果中有下一次进行翻页的token，使用该token可以接着上一次的结果继续读取数据。

参数		说明
	aliveTime	ParallelScan的当前任务有效时间，也是token的有效时间。默认值为60，建议使用默认值，单位为秒。如果在有效时间内没有发起下一次请求，则不能继续读取数据。持续发起请求会刷新token有效时间。 **说明** 由于服务端采用异步方式清理过期任务，因此当前任务只保证在设置的有效时间内不会过期，但不能保证有效时间之后一定过期。
columnsToGet		ParallelScan目前仅可以扫描多元索引中的数据，需要在创建多元索引时设置附加存储（即store=true）。
sessionId		本次并发扫描数据任务的sessionId。创建Session可以通过ComputeSplits API来创建，同时获得本次任务支持的最大并发数。

示例

单并发扫描数据和多线程并发扫描数据的代码示例如下：

- 单并发扫描数据

  相对于多并发扫描数据，单并发扫描数据的代码更简单，单并发代码无需关心currentParallelId和maxParallel参数。单并发使用方式的整体吞吐比Search接口方式高，但是比多线程多并发使用方式的吞吐低，多线程多并发方式请参见最下方的“多线程并发扫描数据”示例代码。

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.ComputeSplitsRequest;
import com.alicloud.openservices.tablestore.model.ComputeSplitsResponse;
import com.alicloud.openservices.tablestore.model.Row;
import com.alicloud.openservices.tablestore.model.SearchIndexSplitsOptions;
import com.alicloud.openservices.tablestore.model.iterator.RowIterator;
import com.alicloud.openservices.tablestore.model.search.ParallelScanRequest;
import com.alicloud.openservices.tablestore.model.search.ParallelScanResponse;
import com.alicloud.openservices.tablestore.model.search.ScanQuery;
import com.alicloud.openservices.tablestore.model.search.SearchRequest.ColumnsToGet;
import com.alicloud.openservices.tablestore.model.search.query.MatchAllQuery;
import com.alicloud.openservices.tablestore.model.search.query.Query;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
public class Test {
    public static List<Row> scanQuery(final SyncClient client) {
        String tableName = "<TableName>";
        String indexName = "<IndexName>";
        //获取sessionId和本次请求支持的最大并发数。
        ComputeSplitsRequest computeSplitsRequest = new ComputeSplitsRequest();
        computeSplitsRequest.setTableName(tableName);
        computeSplitsRequest.setSplitsOptions(new SearchIndexSplitsOptions(indexName));
        ComputeSplitsResponse computeSplitsResponse = client.computeSplits(computeSplitsR
equest);
```

```
        byte[] sessionId = computeSplitsResponse.getSessionId();
        int splitsSize = computeSplitsResponse.getSplitsSize();
        /*
         *  创建并发扫描数据请求。
         */
        ParallelScanRequest parallelScanRequest = new ParallelScanRequest();
        parallelScanRequest.setTableName(tableName);
        parallelScanRequest.setIndexName(indexName);
        ScanQuery scanQuery = new ScanQuery();
        //该query决定了扫描出的数据范围，可用于构建嵌套的复杂的query。
        Query query = new MatchAllQuery();
        scanQuery.setQuery(query);
        //设置单次请求返回的数据行数。
        scanQuery.setLimit(2000);
        parallelScanRequest.setScanQuery(scanQuery);
        ColumnsToGet columnsToGet = new ColumnsToGet();
        columnsToGet.setColumns(Arrays.asList("col_1", "col_2"));
        parallelScanRequest.setColumnsToGet(columnsToGet);
        parallelScanRequest.setSessionId(sessionId);
        /*
         * 使用builder模式创建并发扫描数据请求，功能与前面一致。
         */
        ParallelScanRequest parallelScanRequestByBuilder = ParallelScanRequest.newBuilder
()
            .tableName(tableName)
            .indexName(indexName)
            .scanQuery(ScanQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(2000)
                .build())
            .addColumnsToGet("col_1", "col_2")
            .sessionId(sessionId)
            .build();
        List<Row> result = new ArrayList<>();
        /*
         * 使用原生的API扫描数据。
         */
        {
            ParallelScanResponse parallelScanResponse = client.parallelScan(parallelScanR
equest);
            //下次请求的ScanQuery的token。
            byte[] nextToken = parallelScanResponse.getNextToken();
            //获取数据。
            List<Row> rows = parallelScanResponse.getRows();
            result.addAll(rows);
            while (nextToken != null) {
                //设置token。
                parallelScanRequest.getScanQuery().setToken(nextToken);
                //继续扫描数据。
                parallelScanResponse = client.parallelScan(parallelScanRequest);
                //获取数据。
                rows = parallelScanResponse.getRows();
                result.addAll(rows);
                nextToken = parallelScanResponse.getNextToken();
```

```
            }
        }
        /*
         *  推荐方式。
         *  使用iterator方式扫描所有匹配数据。使用方式上更简单，速度和前面方法一致。
         */
        {
            RowIterator iterator = client.createParallelScanIterator(parallelScanRequestB
yBuilder);
            while (iterator.hasNext()) {
                Row row = iterator.next();
                result.add(row);
                //获取具体的值。
                String col_1 = row.getLatestColumn("col_1").getValue().asString();
                long col_2 = row.getLatestColumn("col_2").getValue().asLong();
            }
        }
        /*
         * 关于失败重试的问题，如果本函数的外部调用者有重试机制或者不需要考虑失败重试问题，可以忽略
此部分内容。
         * 为了保证可用性，遇到任何异常均推荐进行任务级别的重试，重新开始一个新的ParallelScan任务。
         * 异常分为如下两种：
         * 1、服务端Session异常OTSSessionExpired。
         * 2、调用者客户端网络等异常。
         */
        try {
            //正常处理逻辑。
            {
                RowIterator iterator = client.createParallelScanIterator(parallelScanRequ
estByBuilder);
                while (iterator.hasNext()) {
                    Row row = iterator.next();
                    //处理row，内存足够大时可直接放到list中。
                    result.add(row);
                }
            }
        } catch (Exception ex) {
            //重试。
            {
                result.clear();
                RowIterator iterator = client.createParallelScanIterator(parallelScanRequ
estByBuilder);
                while (iterator.hasNext()) {
                    Row row = iterator.next();
                    //处理row，内存足够大时可直接放到list中。
                    result.add(row);
                }
            }
        }
        return result;
    }
}
```

- 多线程并发扫描数据

```
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.ComputeSplitsRequest;
import com.alicloud.openservices.tablestore.model.ComputeSplitsResponse;
import com.alicloud.openservices.tablestore.model.Row;
import com.alicloud.openservices.tablestore.model.SearchIndexSplitsOptions;
import com.alicloud.openservices.tablestore.model.iterator.RowIterator;
import com.alicloud.openservices.tablestore.model.search.ParallelScanRequest;
import com.alicloud.openservices.tablestore.model.search.ScanQuery;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Semaphore;
import java.util.concurrent.atomic.AtomicLong;
public class Test {
    public static void scanQueryWithMultiThread(final SyncClient client, String tableName
, String indexName) throws InterruptedException {
        // 获取机器的CPU数量。
        final int cpuProcessors = Runtime.getRuntime().availableProcessors();
        // 指定客户端多线程的并发数量。建议和客户端的CPU核数一致，避免客户端压力太大，影响查询性能。
        final Semaphore semaphore = new Semaphore(cpuProcessors);
        // 获取sessionId和本次请求支持的最大并发数。
        ComputeSplitsRequest computeSplitsRequest = new ComputeSplitsRequest();
        computeSplitsRequest.setTableName(tableName);
        computeSplitsRequest.setSplitsOptions(new SearchIndexSplitsOptions(indexName));
        ComputeSplitsResponse computeSplitsResponse = client.computeSplits(computeSplitsR
equest);
        final byte[] sessionId = computeSplitsResponse.getSessionId();
        final int maxParallel = computeSplitsResponse.getSplitsSize();
        // 业务统计行数使用。
        AtomicLong rowCount = new AtomicLong(0);
        /*
         *  为了使用一个函数实现多线程功能，此处构建一个内部类继承Thread来使用多线程。
         *  您也可以构建一个正常的外部类，使代码更有条理。
         */
        final class ThreadForScanQuery extends Thread {
            private final int currentParallelId;
            private ThreadForScanQuery(int currentParallelId) {
                this.currentParallelId = currentParallelId;
                this.setName("ThreadForScanQuery:" + maxParallel + "-" + currentParallelI
d);  // 设置线程名称。
            }
            @Override
            public void run() {
                System.out.println("start thread:" + this.getName());
                try {
                    // 正常处理逻辑。
                    {
                        ParallelScanRequest parallelScanRequest = ParallelScanRequest.new
Builder()
                                .tableName(tableName)
                                .indexName(indexName)
                                .scanQuery(ScanQuery.newBuilder()
                                        .query(QueryBuilders.range("col_long").lessThan(1
0_0000)) // 此处的query决定了获取什么数据。
```

```
                                        .limit(2000)
                                        .currentParallelId(currentParallelId)
                                        .maxParallel(maxParallel)
                                        .build())
                                .addColumnsToGet("col_long", "col_keyword", "col_bool")
// 设置要返回的多元索引中的部分字段，或者使用下行注释的内容获取多元索引中全部数据。
                                //.returnAllColumnsFromIndex(true)
                                .sessionId(sessionId)
                                .build();
                        // 使用Iterator形式获取所有数据。
                        RowIterator ltr = client.createParallelScanIterator(parallelScanR
equest);
                        long count = 0;
                        while (ltr.hasNext()) {
                            Row row = ltr.next();
                            // 增加自定义的处理逻辑，此处代码以统计行数为例介绍。
                            count++;
                        }
                        rowCount.addAndGet(count);
                        System.out.println("thread[" + this.getName() + "] finished. this
thread get rows:" + count);
                    }
                } catch (Exception ex) {
                    // 如果有异常，此处需要考虑重试正常处理逻辑。
                } finally {
                    semaphore.release();
                }
            }
        }
        // 多个线程同时执行，currentParallelId取值范围为[0, maxParallel)。
        List<ThreadForScanQuery> threadList = new ArrayList<ThreadForScanQuery>();
        for (int currentParallelId = 0; currentParallelId < maxParallel; currentParallelI
d++) {
            ThreadForScanQuery thread = new ThreadForScanQuery(currentParallelId);
            threadList.add(thread);
        }
        // 同时启动。
        for (ThreadForScanQuery thread : threadList) {
            // 利用semaphore限制同时启动的线程数量，避免客户端瓶颈。
            semaphore.acquire();
            thread.start();
        }
        // 主线程阻塞等待所有线程完成任务。
        for (ThreadForScanQuery thread : threadList) {
            thread.join();
        }
        System.out.println("all thread finished! total rows:" + rowCount.get());
    }
}
```

7.7. 高级功能

7.7.1. 虚拟列

使用虚拟列功能时，您可以通过修改多元索引Schema或者新建多元索引来实现新字段新数据类型的查询功能，而无需修改表格存储的存储结构及数据。

解决什么问题

虚拟列功能支持用户在创建多元索引的时候将表中一列映射到多元索引中的虚拟列。新的虚拟列类型可以不同于表中的原始列类型，以便支持用户在不修改表结构和数据的情况下新建一列，新的列可以用于查询加速或者采用不同的分词器。

- 一个Text字段支持不同的分词器

  单个字符串列可以映射到多元索引多个Text列，不同Text列采用不同的分词，以便满足不同的业务需求。

- 查询加速

  不对表中数据做清洗和重建，只需要将相应列映射为其他类型，即可在部分场景下提升查询性能。例如数字类型转换为keyword类型可以提高精确查询（TermQuery）的性能，string类型转换为数字类型可以提高范围查询（RangeQuery）的性能。

注意事项

- 虚拟列支持不同类型到字符串类型的相互转换，转换规则请参见下表。

数据表中字段类型	虚拟列字段类型
String	Keyword（含数组）
String	Text（含数组）
String	Long（含数组）
String	Double（含数组）
String	Geo-point（含数组）
Long	Keyword
Long	Text
Double	Keyword
Double	Text

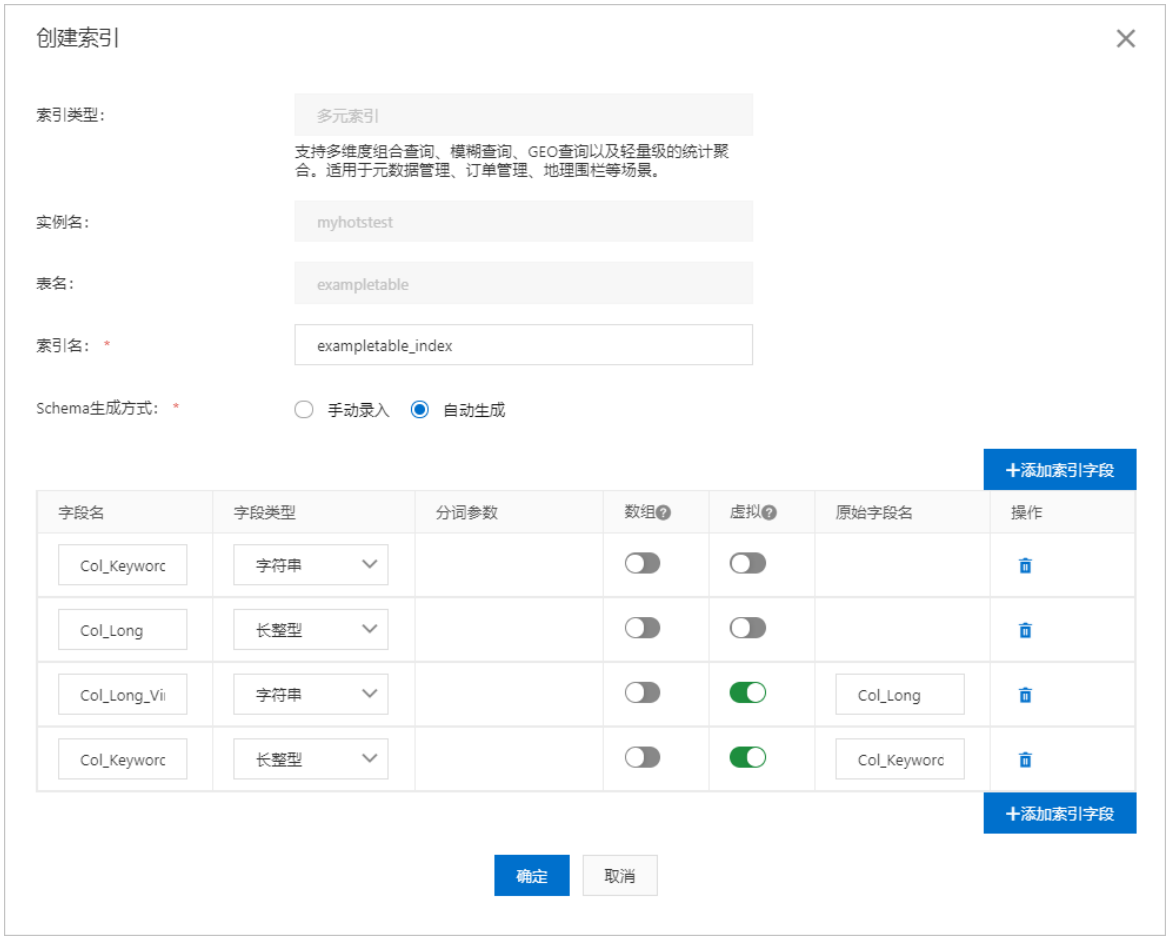
- 虚拟列目前仅支持用在查询语句中，不能用在ColumnsToGet返回列值，如果需要返回列值，可以指定返回该虚拟列的原始列。

通过控制台操作虚拟列

通过控制台在创建多元索引时指定字段为虚拟列后，您可以使用虚拟列查询数据。

1. 登录表格存储控制台。
2. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
3. 在**实例详情**页签的**数据表列表**区域，单击数据表名称或在操作列单击**数据管理**后选择**索引管理**页签。

4. 在**索引管理**页签，单击**创建多元索引**。
5. 在**创建索引**对话框，创建多元索引时指定虚拟列。



i. 系统默认会自动生成索引名，可根据需要输入索引名。

ii. 选择Schema生成方式。

- 当Schema生成方式设置为**手动录入**时，手动输入字段名，选择字段类型以及设置是否开启数组。
- 当Schema生成方式设置为**自动生成**时，系统会自动将数据表的主键列和属性列作为索引字段，可根据需要选择字段类型以及设置是否开启数组。

> **说明** **字段名**和**字段类型**需与数据表匹配。数据表字段类型与多元索引字段类型的对应关系请参见数据类型映射。

iii. 创建虚拟列。

> **注意** 创建虚拟列时，原始字段名必须在数据表中存在，且原始字段的数据类型必须和虚拟列的字段类型相匹配。

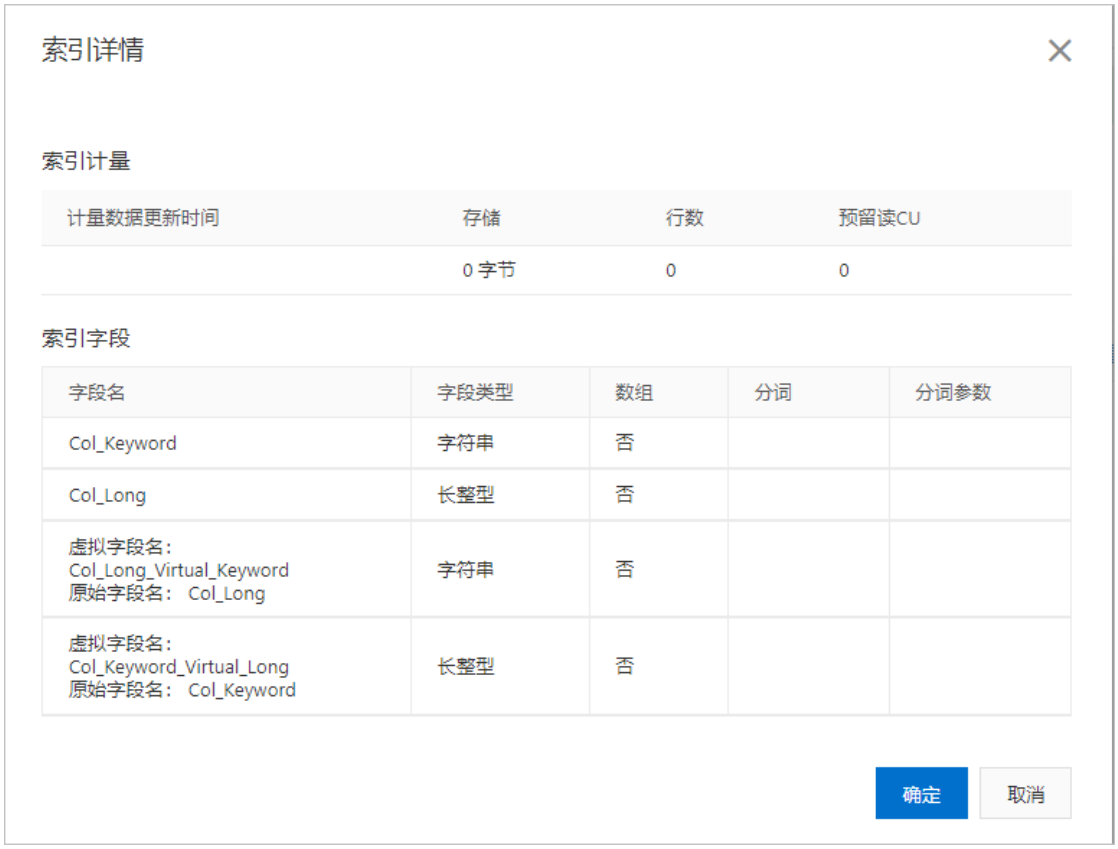
a. 单击**添加索引字段**。

b. 输入**字段名**和**字段类型**。

c. 单击开启虚拟列开关，并输入**原始字段名**。

iv. 单击**确定**。

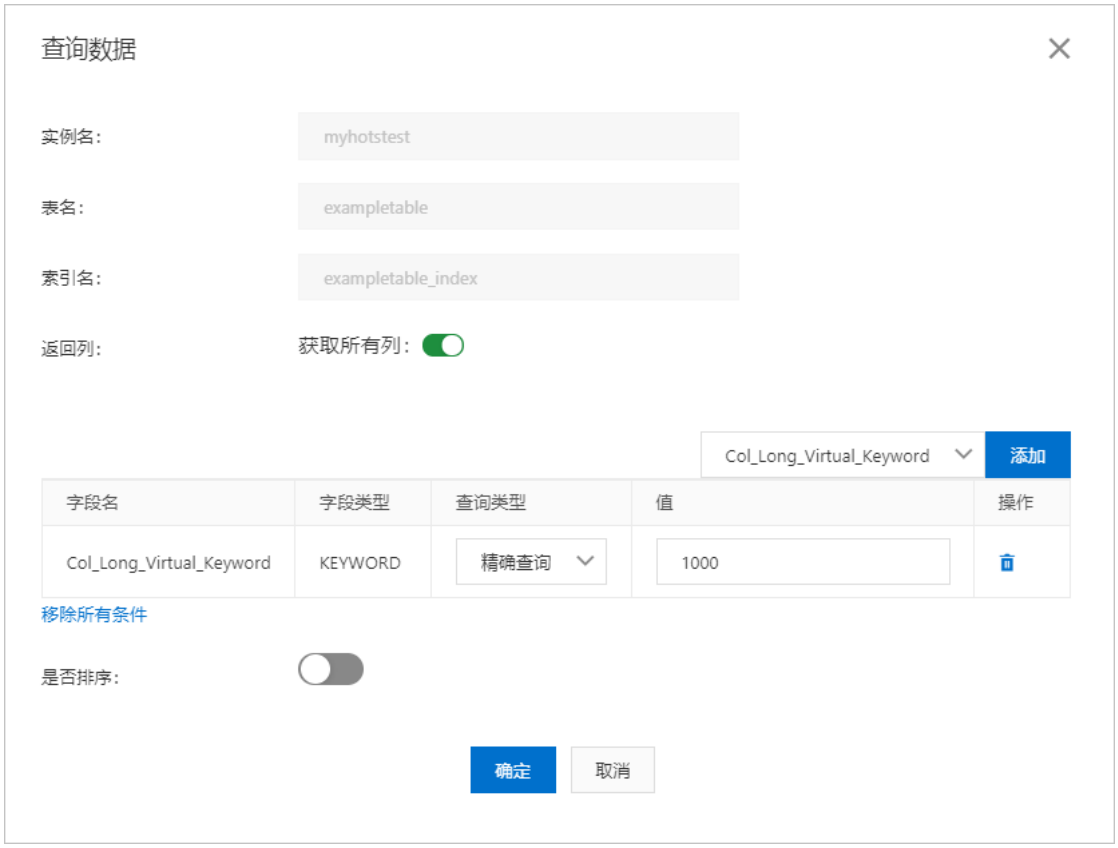
多元索引创建完成后，在索引列表的操作列，单击**索引详情**，可查看索引表的索引计量信息、索引字段等信息。



6. 使用虚拟列查询数据。

i. 在多元索引的操作列单击**搜索**。

ii. 在**查询数据**对话框，查询数据。



a. 系统默认返回所有列，如果需要显示指定属性列，关闭**获取所有列**并输入需要返回的属性列，多个属性列之间用英文逗号（,）隔开。

b. 选择索引字段，单击**添加**，并设置索引字段的查询类型和值。

c. 系统默认关闭排序功能，如需根据索引字段对返回结果进行排序，打开**是否排序**后，根据需要添加索引字段并配置排序方式。

d. 单击**确定**。

符合查询条件的数据会显示在**索引管理**页签中。

通过SDK操作虚拟列

通过SDK在创建多元索引时指定字段为虚拟列后，您可以使用虚拟列查询数据。

1. 创建多元索引时指定虚拟列。
   - 参数

     关于参数的详细说明，请参见创建多元索引。
   - 示例

     创建一个多元索引，多元索引包含Col_Keyword和Col_Long两列，同时创建虚拟列Col_Keyword_Virtual_Long和Col_Long_Virtual_Keyword。Col_Keyword_Virtual_Long映射为数据表中Col_Keyword列，虚拟列Col_Long_Virtual_Keyword映射为数据表中Col_Long列。

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName); //设置数据表名称。
    request.setIndexName(indexName); //设置多元索引名称。
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
        new FieldSchema("Col_Keyword", FieldType.KEYWORD) //设置字段名和类型。
            .setIndex(true) //设置开启索引。
            .setEnableSortAndAgg(true) //设置开启排序和统计功能。
            .setStore(true),
        new FieldSchema("Col_Keyword_Virtual_Long", FieldType.LONG) //设置字段名和类型
。
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true) //设置字段是否为虚拟列。
            .setSourceFieldName("Col_Keyword"), //虚拟列对应的数据表中字段。
        new FieldSchema("Col_Long", FieldType.LONG)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true),
        new FieldSchema("Col_Long_Virtual_Keyword", FieldType.KEYWORD)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true)
            .setSourceFieldName("Col_Long")));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); //调用client创建多元索引。
}
```

2. 使用虚拟列查询数据。

   查询表中Col_Long_Virtual_Keyword列的值能够匹配"1000"的数据，返回匹配到的总行数和一些匹配成功的行。

```
private static void query(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermsQuery termsQuery = new TermsQuery(); //设置查询类型为TermsQuery。
    termsQuery.setFieldName("Col_Long_Virtual_Keyword"); //设置要匹配的字段。
    termsQuery.addTerm(ColumnValue.fromString("1000")); //设置要匹配的值。
    searchQuery.setQuery(termsQuery);
    searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("tableName", "indexName", searchQue
ry);
    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    columnsToGet.setReturnAll(true); //设置返回所有列，不支持返回虚拟列。
    searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount()); //匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

7.7.2. 动态修改schema

通过此功能，您可以动态修改多元索引的schema，例如在多元索引中新增、更新或者删除索引列，修改多元索引的路由键等。

功能概述

表格存储数据表是schema free的，而多元索引是强schema的。创建多元索引时，您需要指定添加到多元索引中的列，这样使用多元索引查询数据时才能查询到这些列。但是出于业务变更、性能优化等目的，修改多元索引schema成为一个非常常见的需求。例如以下场景：

- 新增索引列：随着业务发展，需要查询更多的列，则可以添加更多的索引列。
- 更新索引列：修改Text类型字段的分词器。
- 删除索引列：创建多元索引时，添加了不会作为查询条件的列，需要移除掉。
- 修改路由键：查询时合理指定路由键，减少查询时的读放大。

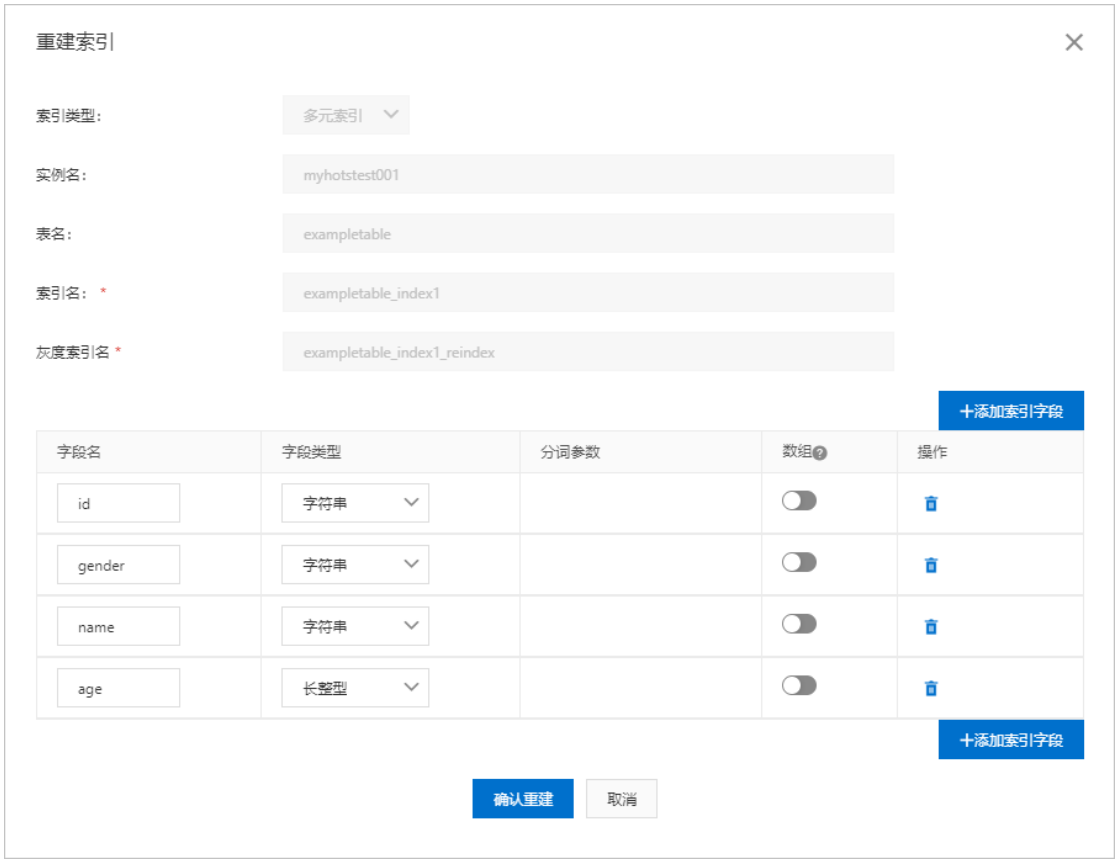
动态修改schema的具体流程如下所示，整个过程对业务层透明，您无需变更业务代码，即可实现轻量级动态修改schema。

1. 在数据表上创建一个灰度索引，并根据需要新增、修改和删除多元索引的schema。
2. 等待数据表的存量和增量数据同步到灰度索引，直到同步进度与多元索引的进度相同。
3. 通过A/B测试逐步引流查询流量到灰度索引，直到100%查询流量均切到灰度索引。
4. 验证无误后，交换源索引和灰度索引的schema。
5. 删除源索引。

操作步骤

1. 进入**索引管理**页签。
   i. 登录表格存储控制台。
   ii. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
   iii. 在**实例详情**页签的**数据表列表**区域，单击数据表名称或在操作列单击**数据管理**后选择**索引管理**页签。
2. 基于源索引创建灰度索引。
   i. 在**索引管理**页签，单击多元索引操作列的**修改schema**。

ii. 在**重建索引**对话框，根据需要添加、修改或者删除索引字段。

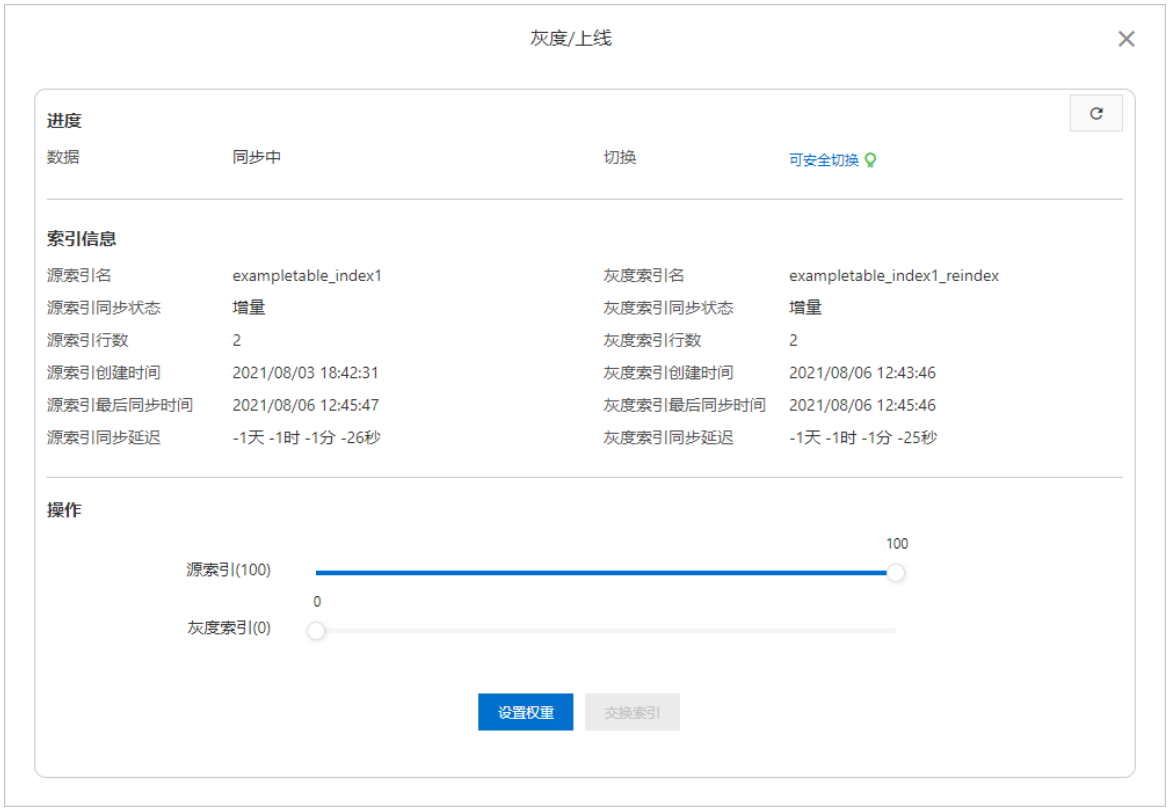


iii. 单击**确认重建**。

iv. 在**schema对比**对话框，查看源索引和灰度索引的schema对比信息，确认无误后，单击**确定**。

3. 查看索引同步信息。

灰度索引会经历“存量同步”和“增量同步”两个阶段。数据同步完成前，系统会提示**切换有风险**，禁止用户切换；当灰度索引同步进度追上源索引的同步进度后，系统会提示**可安全切换**，此时请继续执行后续操作。



i. 单击源索引前的 ＋ 图标或者源索引名称。

系统会显示源索引的灰度索引。

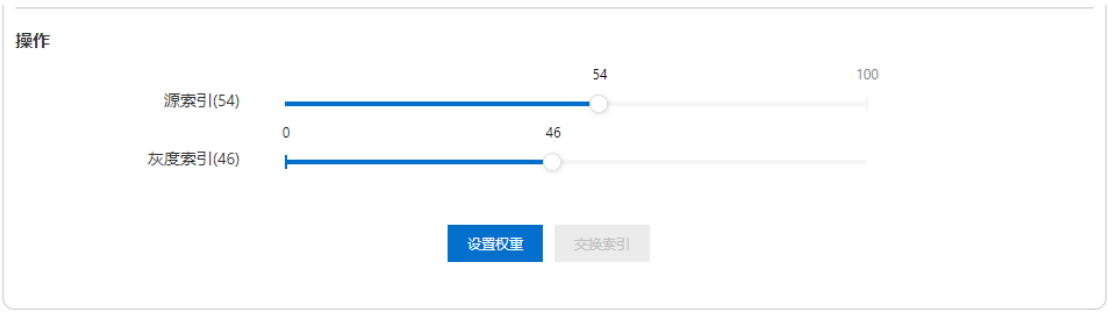
ii. 单击灰度索引操作列的**灰度/上线**。

iii. 在**灰度/上线**对话框，查看索引同步信息。

4. 索引同步完成后，通过设置权重进行A/B测试。

A/B测试功能支持将查询流量按照一定比例分摊到源索引和灰度索引来验证修改schema的效果。仅当查询流量全部都切到灰度索引时，才能继续执行后续步骤。

i. 在**灰度/上线**对话框的**操作**区域，拖动滑块调整源索引和灰度索引的权重后，单击**设置权重**。



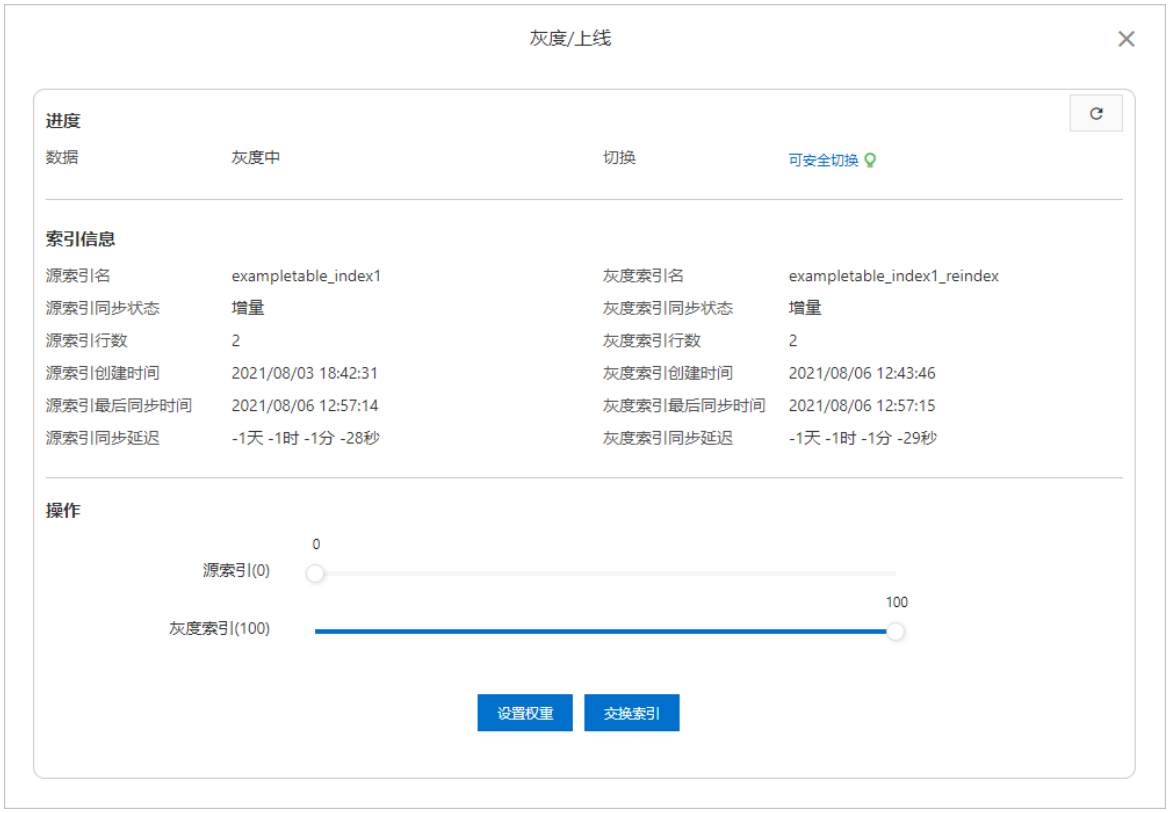
ii. 在**设置权重**对话框，查看权重数据和schema对比信息。

iii. 确认无误后，单击**设置权重**。

iv. 在系统提示框中单击**确定**。

5. 查询流量全部切换到灰度索引后，交换源索引和灰度索引的schema。

交换索引后，源索引名会关联到新schema，而灰度索引名会关联到老schema，并且100%查询流量会查询源索引名关联的新schema。

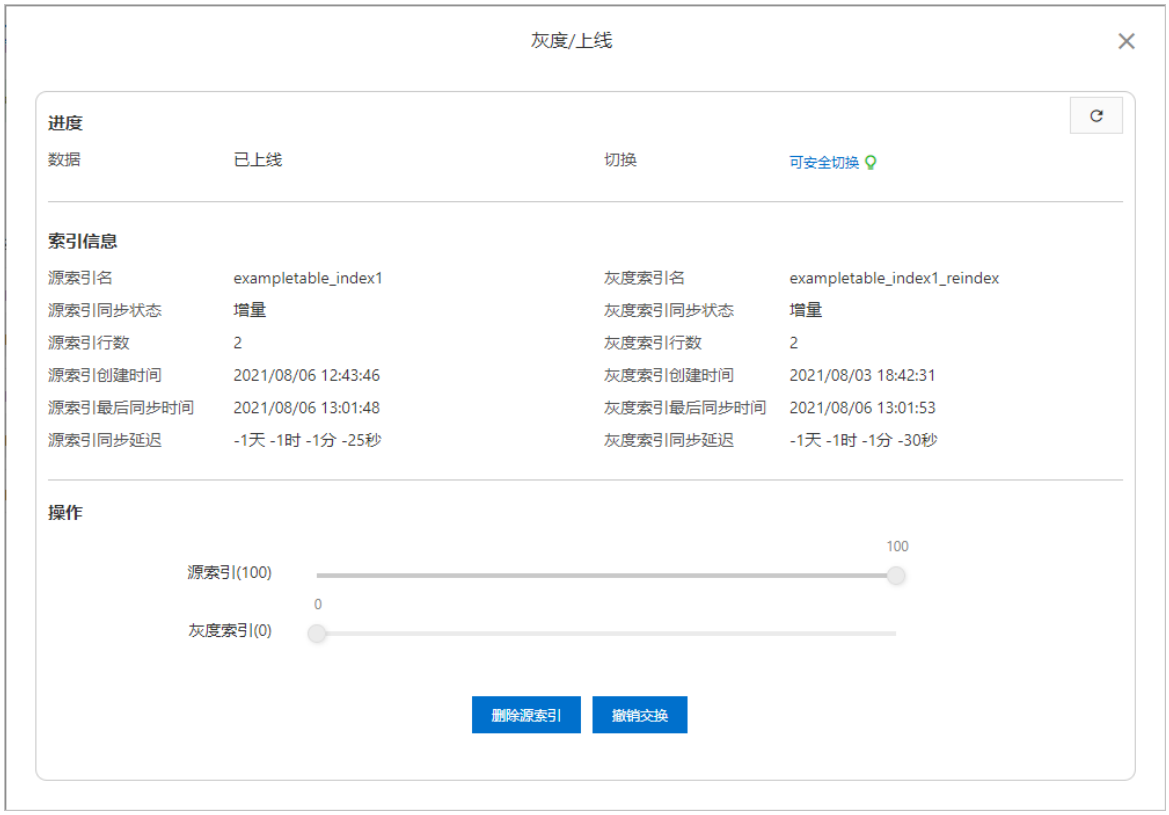


i. 在**灰度/上线**对话框的**操作**区域，单击**交换索引**。

ii. 在**交换索引**对话框，查看源索引和灰度索引的schema对比信息，确认无误后，单击**确认交换**。

6. 交换索引并验证索引无误后，建议静置一段时间（例如一天）再删除源索引。

在**灰度/上线**对话框，单击**删除源索引**，即可删除源索引。



安全性

为了保证操作的安全性，表格存储提供了“回滚机制”和“切换提醒”，最大限度地降低修改索引过程中可能的风险。

- 回滚机制

  动态修改Schema的关键步骤均支持回滚。

  - 创建灰度索引后，如果发现灰度索引的Schema不符合预期，您可以删除后重新创建。
  - 在A/B测试阶段，通过设置查询权重，将查询流量逐步引流到灰度索引。在此过程中，如果发现问题，可以随时重新设置查询权重，将查询流量重新引流到源索引。
  - 交换源索引和灰度索引的Schema后，如果发现问题，您可以随时**撤销交换**，切回源索引的Schema。**交换索引**和**撤销交换**互为逆向操作。

- 切换提醒

  如果在灰度索引的同步进度落后于源索引时将流量切到灰度索引，则可能会出现查询数据回退。此时表格存储会通过源索引和灰度索引的同步状态和最后同步时间，判断“是否可以安全切换”。

  当出现如下情况时，表格存储会判断为**可安全切换**。

  - 当源索引处于全量阶段，灰度索引为全量或者增量阶段，即灰度索引已追上源索引。
  - 源索引和灰度索引均处于增量阶段，且“源索引最后同步时间 - 60 s <= 灰度索引最后同步时间”，即灰度索引落后源索引的时间不超过1分钟。

7.7.3. 模糊查询

对于通配符查询（WildcardQuery）中查询模式为 `*word*` 的场景，您可以使用模糊分词方式（即模糊分词和短语匹配查询组合使用）来实现性能更好的模糊查询。

背景信息

模糊查询是数据库业务中常见的需求，例如查询文件名、手机号码等。在表格存储中要实现模糊查询，通常使用多元索引的通配符查询来实现类似于MySQL中的like功能，但是通配符查询存在查询词长度有限制（最长20个字符）以及性能会随着数据量增长而下降的限制。

为了解决通配符查询存在的问题，多元索引支持使用模糊分词方式来实现性能更好的模糊查询。当使用模糊分词方式时，查询词长度无限制，但是原文内容会限制最大1024字符或者汉字，超过后会截断，只保留前1024个字符或者汉字。

适用场景

请根据查询场景选择合适的方式实现模糊查询。

- 对于通配符查询中查询模式为 `*word*` 的场景，例如通过 `"123"` 匹配手机号码中任意位置包含 `123` 的号码，请使用模糊分词方式来实现模糊查询。

  在此场景中，大部分情况下使用模糊分词方式会比使用通配符查询有10倍以上的性能提升。

  假设数据表中包含file_name列，该列在多元索引中的字段类型为Text且分词类型为模糊分词（Fuzzy_Analyzer）。如果使用多元索引查询需要查询到file_name列值为 `2021 woRK@杭州` 的行，则查询时必须使用短语匹配查询（MatchPhraseQuery）且设置查询词为位置连续的子字符串。

  - 如果查询词为 `2021` 、 `20` 、 `21` 、 `work` 、 `WORK` 、 `@` 、 `杭` 、 `州` 、 `杭州` 、 `@杭州` 中的任意一个，则可以匹配到file_name列值为 `2021 woRK@杭州` 的行。

- 如果查询词为 `21work` 、 `2021杭州` 、 `2120` 、 `#杭州` 中的任意一个，则无法匹配到file_name列值为 `2021 woRK@杭州` 的行。

- 对于其他复杂查询场景，请使用通配符查询方式来实现模糊查询。更多信息，请参见通配符查询。

使用方式

使用模糊分词方式实现模糊查询的具体步骤如下：

1. 创建多元索引时，指定列类型为Text且分词类型为模糊分词（Fuzzy Analyzer），其他参数保持默认配置即可。

   > **说明** 如果已创建多元索引，您可以通过动态修改schema功能为指定列添加虚拟列，同时设置虚拟列为Text类型且分词类型为模糊分词来实现。具体操作，请分别参见动态修改schema和虚拟列。

2. 使用多元索引查询数据时，使用MatchPhraseQuery。更多信息，请参见短语匹配查询。

示例

以下代码通过测试用例的方式展示了使用模糊分词方式实现模糊查询的效果。

```
package com.aliyun.tablestore.search.test;
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.*;
import com.alicloud.openservices.tablestore.model.search.*;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
import com.aliyun.tablestore.search.common.Conf;
import com.aliyun.tablestore.search.common.TableStoreHelper;
import org.junit.Test;
import java.util.Arrays;
import java.util.Collections;
import static org.junit.Assert.assertEquals;
public class Test {
    private static final Conf conf = Conf.newInstance("src/test/resources/conf.json");
    private static final SyncClient ots = new SyncClient(conf.getEndpoint(), conf.getAccess
Id(), conf.getAccessKey(), conf.getInstanceName());
    private static final String tableName = "analysis_test";
    private static final String indexName = "analysis_test_index";
    @Test
    public void testFuzzyMatchPhrase() {
        // 清理表和索引。
        TableStoreHelper.deleteTableAndIndex(ots, tableName);
        // 创建表。
        TableStoreHelper.createTable(ots, tableName);
        // 定义表schema。
        IndexSchema indexSchema = new IndexSchema();
        indexSchema.setFieldSchemas(Collections.singletonList(
                // 注意：当原来查询的name字段为Keyword类型时，如果修改该字段为Text类型并为该字段设
置分词后，查询可能会出现异常。
                // 如果需要同时保留Keyword和Text类型，请参见"虚拟列"功能的示例。假如使用name字段只
需要完成匹配*abc*的查询功能，则只用Text类型的字段即可，无需Keyword类型。
                new FieldSchema("name", FieldType.TEXT).setAnalyzer(FieldSchema.Analyzer.Fu
zzy)
        ));
```

```
        // 创建多元索引。
        TableStoreHelper.createIndex(ots, tableName, indexName, indexSchema);
        // 写入一行数据。
        PrimaryKey primaryKey = PrimaryKeyBuilder.createPrimaryKeyBuilder()
                .addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("1"))
                .addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong(1))
                .addPrimaryKeyColumn("pk3", PrimaryKeyValue.fromBinary(new byte[]{1, 2, 3})
)
                .build();
        RowPutChange rowPutChange = new RowPutChange(tableName, primaryKey);
        // 写入属性列。
        rowPutChange.addColumn("name", ColumnValue.fromString("调音师1024x768P.mp4"));
        PutRowRequest request = new PutRowRequest(rowPutChange);
        ots.putRow(request);
        // 等待多元索引中同步完成一条数据。
        TableStoreHelper.waitDataSync(ots, tableName, indexName, 1);
        // 匹配*abc*的查询功能场景展示。
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "调", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "调音", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "调 音", 0);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "调音师102", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "调音师1024", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "调音师1024x", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "调音师1024x7", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "调音师1024x768P.mp4", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "24x768P.mp4", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "24x76 8P.mp4", 0);
        assertMatchPhraseQuery(ots, tableName, indexName, "name", "24x7 P.mp4", 0);
    }
    @Test
    // 使用虚拟列。
    public void testFuzzyMatchPhraseWithVirtualField() {
        // 清理表和索引。
        TableStoreHelper.deleteTableAndIndex(ots, tableName);
        // 创建数据表。
        TableStoreHelper.createTable(ots, tableName);
        // 定义表schema。
        IndexSchema indexSchema = new IndexSchema();
        indexSchema.setFieldSchemas(Arrays.asList(
                // 原始字段为Keyword类型，方便进行等值查询。
                new FieldSchema("name", FieldType.KEYWORD).setIndex(true).setStore(true),
                // 创建一个虚拟列“name_virtual_text”，同时设置虚拟列为Text类型且分词类型为Fuzzy。
该虚拟列的来源为“name”字段。
                new FieldSchema("name_virtual_text", FieldType.TEXT).setIndex(true).setAnal
yzer(FieldSchema.Analyzer.Fuzzy).setVirtualField(true).setSourceFieldName("name")
        ));
        // 创建多元索引。
        TableStoreHelper.createIndex(ots, tableName, indexName, indexSchema);
        // 写入一行数据。
        PrimaryKey primaryKey = PrimaryKeyBuilder.createPrimaryKeyBuilder()
                .addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("1"))
                .addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong(1))
                .addPrimaryKeyColumn("pk3", PrimaryKeyValue.fromBinary(new byte[]{1, 2, 3})
)
```

```
                .build();
        RowPutChange rowPutChange = new RowPutChange(tableName, primaryKey);
        // 写入属性列。
        rowPutChange.addColumn("name", ColumnValue.fromString("调音师1024x768P.mp4"));
        PutRowRequest request = new PutRowRequest(rowPutChange);
        ots.putRow(request);
        // 等待多元索引中同步完成一条数据。
        TableStoreHelper.waitDataSync(ots, tableName, indexName, 1);
        // 配置*abc*的查询场景展示。
        // 请注意查询字段为虚拟列“name_virtual_text”，而不是“name”。
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "调", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "调音", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "调 音", 0);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "调音师102",
1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "调音师1024",
1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "调音师1024x"
, 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "调音师1024x7
", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "调音师1024x7
68P.mp4", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "24x768P.mp4
", 1);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "24x76 8P.mp
4", 0);
        assertMatchPhraseQuery(ots, tableName, indexName, "name_virtual_text", "24x7 P.mp4"
, 0);
    }
    // 展示MatchPhraseQuery如何实现。
    public static void assertMatchPhraseQuery(SyncClient ots, String tableName, String inde
xName, String fieldName, String searchContent, long exceptCount) {
        SearchRequest searchRequest = new SearchRequest();
        searchRequest.setTableName(tableName);
        searchRequest.setIndexName(indexName);
        SearchQuery searchQuery = new SearchQuery();
        // 使用MatchPhraseQuery查询分词字段。
        searchQuery.setQuery(QueryBuilders.matchPhrase(fieldName, searchContent).build());
        searchQuery.setLimit(0);
        // 为了展示功能需要，此处设置返回匹配总行数。如果不需要关心匹配总行数，请设置为false，来实现更
高性能。
        searchQuery.setGetTotalCount(true);
        searchRequest.setSearchQuery(searchQuery);
        SearchResponse response = ots.search(searchRequest);
        assertEquals(String.format("field:[%s], searchContent:[%s]", fieldName, searchConte
nt), exceptCount, response.getTotalCount());
    }
}
```

7.8. 实践

本文介绍多元索引的功能详解以及解决方案。

概要

Tablestore发布多元索引功能，打造统一的在线数据平台

功能详解

- 翻页功能
- Array和Nested对比
- 路由功能

解决方案

- 气象格点数据
- 用户画像
- 交通数据
- 物联网元数据
- 订单系统

8.二级索引

8.1. 简介

通过本文您可以了解二级索引的区别、基本概念、注意事项、功能等。

背景

通过创建一张或多张索引表，使用索引表的主键列查询，二级索引相当于把数据表的主键查询能力扩展到了不同的列。使用二级索引能加快数据查询的效率。

为了满足用户的强一致性查询等需求，表格存储在支持全局二级索引的同时，推出了本地二级索引。

二级索引区别

二级索引包括全局二级索引和本地二级索引，请根据实际需要选择合适的二级索引。

名称	区别
全局二级索引	• 以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 • 索引表的第一列主键可根据需要选择任意主键列或者预定义列。
本地二级索引	• 以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 • 索引表的第一列主键必须和数据表的第一列主键相同。

基本概念

名词	描述
索引表	对数据表中某些列数据的索引。 索引表只能用于读取数据，不能写入数据。
预定义列	在创建数据表时预先定义一些非主键列以及其类型，作为索引表属性列。 **说明** 表格存储为Schema-free模型，原则上一行数据可以写入任意属性列，无需在SCHEMA中指定属性列。
单列索引	只为某一列建立索引。
组合索引	多个列组合成索引，组合索引中包含组合索引列1、列2。
索引表属性列	被映射到索引表中的预定义列。
索引列补齐	系统自动将未出现在索引列中的数据表主键补齐到索引表主键中。

功能

- 单列索引和组合索引

  支持为数据表中的某一列或者多个列建立索引。

- 索引同步

  全局二级索引和本地二级索引的数据同步方式不同。

  - 使用全局二级索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。
  - 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。

- 覆盖索引（Covered Indexes）

  支持索引表中带有属性列。在创建数据表时预先定义一些列（称为预定义列）后，可以对任意预定义列和数据表主键列进行索引，指定数据表的若干个预定义列作为索引表属性列。索引表中也可以不包含任何属性列。

  当指定数据表的若干个预定义列作为索引表属性列时，读取索引表可以直接得到数据表中对应预定义列的值，无需反查数据表。

- 存量索引

  支持新建的索引表中包含数据表中的存量数据。

- 稀疏索引（Sparse Indexes）

  如果数据表的某个预定义列作为索引表的属性列，当数据表某行中不存在该预定义列时，只要索引列全部存在，仍会为此行建立索引。但是如果部分索引列缺失，则不会为此行建立索引。

  例如数据表有PK0、PK1、PK2三列主键，Defined0、Defined1、Defined2三列预定义列，设置索引表主键为PK0、Defined0、Defined1，索引表属性列为Defined2。

  - 当数据表某行的属性列中，只包含Defined0、Defined1两列，不包含Defined2列时，会为此行建立索引。
  - 当数据表某行的属性列中，只包含Defined0、Defined2两列，不包含Defined1列时，不会为此行建立索引。

使用限制

更多信息，请参见二级索引限制。

注意事项

索引表的索引列和属性列设置注意事项如下：

- 系统会自动对索引表进行索引列补齐。在对索引表进行扫描时，请注意填充对应主键列的范围（一般为负无穷到正无穷）。

  在创建索引表时，只需要指定索引列，其它列会由系统自动添加。例如数据表有PK0和PK1两列主键，Defined0作为预定义列。

  使用全局二级索引时，根据实际需要创建索引。

  - 如果在Defined0上创建索引，则生成的索引表主键会是Defined0、PK0、PK1三列。
  - 如果在Defined0和PK1上创建索引，则生成的索引表主键会是Defined0、PK1、PK0三列。
  - 如果在PK1上创建索引，则生成的索引表主键会是PK1、PK0两列。

  使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。

  - 如果在PK0和Defined0上创建索引，则生成的索引表主键会是PK0、Defined0、PK1三列。
  - 如果在PK0、PK1和Defined0上创建索引，则生成的索引表主键会是PK0、PK1、Defined0三列。
  - 如果在PK0和PK1上创建索引，则生成的索引表主键会是PK0、PK1两列。
- 根据查询模式和成本的考虑选择合适的数据表预定义列作为索引表属性列。

  将数据表的一个预定义列作为索引表的属性列后，查询数据时无需反查数据表即可得到该列的值，但是同时增加了相应的存储成本。反之则需要根据索引表反查数据表。
- 使用全局二级索引时，请选择合适的列作为索引表的第一个主键列。
  - 由于可能导致索引表更新速度变慢，不建议把时间相关列作为索引表主键的第一列。

    建议将时间列进行哈希，然后在哈希后的列上创建索引，如果有需求请使用钉钉联系表格存储技术支持。
  - 由于会导致索引表水平扩展能力受限，影响索引表写入性能，不建议将取值范围非常小，甚至可枚举的列作为索引表主键的第一列，例如性别。

使用索引表时的注意事项如下：

- 在带有索引表的数据表中写入数据时需遵循如下规则，否则在数据表中写入数据会失败。
  - 写入数据时，不能自定义数据的版本号。
  - 批量写入数据时，一个批量写请求中，同一行数据（即主键相同）不能重复存在。

计量计费

更多信息，请参见二级索引计量计费。

8.2. 使用场景

二级索引支持在指定列上建立索引，生成的索引表中的数据按照指定的索引列进行排序，数据表的每一个数据写入都会自动同步到索引表中。您只需向数据表中写入数据，然后根据索引表进行查询，在许多场景下能提高查询的效率。

全局二级索引

在电话话单查询场景下，使用全局二级索引可以满足不同的查询需求。

数据表的信息请参见下表，每次用户通话结束后，都会将此次通话的信息记录到该数据表中。

- CellNumber、StartTime作为数据表的主键，分别代表主叫号码和通话发生时间。
- CalledNumber、Duration和BaseStationNumber三列为数据表的预定义列，分别代表被叫号码、通话时长和基站号码。

CellNumber	StartTime（Unix时间戳）	CalledNumber	Duration	BaseStationNumber
123456	1532574644	654321	60	1
234567	1532574714	765432	10	1
234567	1532574734	123456	20	3
345678	1532574795	123456	5	2

CellNumber	StartTime（Unix时间戳）	CalledNumber	Duration	BaseStationNumber
345678	1532574861	123456	100	2
456789	1532584054	345678	200	3

通过在CalledNumber和BaseStationNumber列上创建索引表，可以满足不同的查询需求。创建索引表的示例代码请参见附录。

假设有如下几种查询需求：

- 查询号码234567的所有主叫话单。

  表格存储的模型是对所有行按照主键进行排序，并且提供顺序扫描（getRange）接口，所以只需要在调用getRange接口时，将CellNumber列（主叫号码）的最大及最小值均设置为234567，StartTime列（通话发生时间）的最小值设置为0，最大值设置为INT_MAX，对数据表进行扫描即可。

```
private static void getRangeFromMainTable(SyncClient client, long cellNumber){
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    //构造主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.fromLo
ng(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLo
ng(0));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //构造主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.fromLong
(cellNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MAX)
;
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    String strNum = String.format("%d", cellNumber);
    System.out.println("号码" + strNum + "的所有主叫话单:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // 如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- 查询号码123456的被叫话单。

表格存储的模型是对所有行按照主键进行排序，由于被叫号码存在于数据表的预定义列中，所以无法进行快速查询。因此可以在被叫号码索引表IndexOnBeCalledNumber上进行查询，由于索引表IndexOnBeCalledNumber是按照被叫号码作为主键，可以直接调用getRange接口扫描索引表得到结果。

索引表IndexOnBeCalledNumber的信息请参见下表。

> **说明** 系统会自动进行索引列补齐。即把数据表的主键添加到索引列后，共同作为索引表的主键，所以索引表中有三列主键。

PK0	PK1	PK2
CalledNumber	CellNumber	StartTime
123456	234567	1532574734
123456	345678	1532574795
123456	345678	1532574861
654321	123456	1532574644
765432	234567	1532574714
345678	456789	1532584054

```
private static void getRangeFromIndexTable(SyncClient client, long cellNumber) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX0_NAME);
    //构造主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.fromLo
ng(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MI
N);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MI
N);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //构造主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.fromLong
(cellNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX)
;
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MAX)
;
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    String strNum = String.format("%d", cellNumber);
    System.out.println("号码" + strNum + "的所有被叫话单:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- 查询基站002从时间1532574740开始的所有话单。

  与上述示例类似，但是查询不仅把BaseStationNumber列作为条件，同时把StartTime列作为查询条件，因此可以在BaseStationNumber和StartTime列上建立组合索引，索引表名称为IndexOnBaseStation1，然后在索引表IndexOnBaseStation1上进行查询。

  索引表IndexOnBaseStation1的信息请参见下表。

PK0	PK1	PK2
BaseStationNumber	StartTime	CellNumber
001	1532574644	123456

PK0	PK1	PK2
001	1532574714	234567
002	1532574795	345678
002	1532574861	345678
003	1532574734	234567
003	1532584054	456789

```
private static void getRangeFromIndexTable(SyncClient client,
                                           long baseStationNumber,
                                           long startTime) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX1_NAME);
    //构造主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLo
ng(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLo
ng(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MI
N);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //构造主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLong
(baseStationNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MAX)
;
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX)
;
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    String strBaseStationNum = String.format("%d", baseStationNumber);
    String strStartTime = String.format("%d", startTime);
    System.out.println("基站" + strBaseStationNum + "从时间" + strStartTime + "开始的所有被
叫话单:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- 查询发生在基站003上时间从1532574861到1532584054的所有通话记录的通话时长。

  在该查询中不仅把BaseStationNumber列和StartTime列作为查询条件，而且只把Duration列作为返回结果。您可以使用上一个查询中的索引表IndexOnBaseStation1，查询索引表成功后反查数据表获取通话时长。

```
private static void getRowFromIndexAndMainTable(SyncClient client,
                                                long baseStationNumber,
                                                long startTime,
                                                long endTime,
                                                String colName) {
```

```
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX1_NAME);
    //构造主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLo
ng(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLo
ng(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MI
N);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //构造主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLong
(baseStationNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong
(endTime));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX)
;
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    String strBaseStationNum = String.format("%d", baseStationNumber);
    String strStartTime = String.format("%d", startTime);
    String strEndTime = String.format("%d", endTime);
    System.out.println("基站" + strBaseStationNum + "从时间" + strStartTime + "到" + strEnd
Time + "的所有话单通话时长:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn mainCalledNumber = curIndexPrimaryKey.getPrimaryKeyColumn(PR
IMARY_KEY_NAME_1);
            PrimaryKeyColumn callStartTime = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMA
RY_KEY_NAME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBuil
der();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, mainCalledNumber.g
etValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, callStartTime.getV
alue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); //构造主表PK。
            //反查数据表。
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, main
TablePK);
            criteria.addColumnsToGet(colName); //读取主表的"通话时长"列。
            //设置读取最新版本。
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(mainTableRow);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
```

```
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

为了提高查询效率，可以在BaseStationNumber列和StartTime列上建立组合索引，并把Duration列作为索引表的属性列，索引表名称为IndexOnBaseStation2，然后在索引表IndexOnBaseStation2上进行查询。

索引表IndexOnBaseStation2的信息请参见下表。

PK0	PK1	PK2	Defined0
BaseStationNumber	StartTime	CellNumber	Duration
001	1532574644	123456	60
001	1532574714	234567	10
002	1532574795	345678	5
002	1532574861	345678	100
003	1532574734	234567	20
003	1532584054	456789	200

```
private static void getRangeFromIndexTable(SyncClient client,
                                           long baseStationNumber,
                                           long startTime,
                                           long endTime,
                                           String colName) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX2_NAME);
    //构造主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLo
ng(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLo
ng(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MI
N);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //构造主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3, PrimaryKeyValue.fromLong
(baseStationNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong
(endTime));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX)
;
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    //设置读取列。
    rangeRowQueryCriteria.addColumnsToGet(colName);
    rangeRowQueryCriteria.setMaxVersions(1);
    String strBaseStationNum = String.format("%d", baseStationNumber);
    String strStartTime = String.format("%d", startTime);
    String strEndTime = String.format("%d", endTime);
    System.out.println("基站" + strBaseStationNum + "从时间" + strStartTime + "到" + strEnd
Time + "的所有话单通话时长:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

由此可见，如果不把Duration列作为索引表的属性列，在每次查询时都需先从索引表中获取数据表的主键，然后对数据表进行随机读。把Duration列作为索引表的属性列后，该列被同时存储在了数据表和索引表中，增加了占用的总存储空间。

- 查询发生在基站003上时间从1532574861到1532584054的所有通话记录的总通话时长、平均通话时长、最大通话时长和最小通话时长。

  相对于上一条查询，此处不要求返回每一条通话记录的时长，只要求返回所有通话时长的统计信息。您可以使用与上一条查询相同的查询方式，然后自行对返回的每条通话时长做计算并得到最终结果。也可以使用Data Lake Analytics，无需客户端计算，直接使用SQL语句返回最终统计结果。

  > **说明** 关于Data Lake Analytics使用的更多信息，请参见OLAP on Tablestore：基于Data Lake Analytics的Serverless SQL大数据分析。其兼容绝大多数MySQL语法，可以进行更复杂、更贴近用户业务逻辑的计算。

本地二级索引

在电话话单查询场景下，使用本地二级索引可以满足不同的查询需求。

数据表的信息请参见下表，每次用户通话结束后，都会将此次通话的信息记录到该数据表中。

- CellNumber、StartTime作为数据表的主键，分别代表主叫号码和通话发生时间。
- CalledNumber、Duration和BaseStationNumber三列为数据表的预定义列，分别代表被叫号码、通话时长和基站号码。

CellNumber	StartTime（Unix时间戳）	CalledNumber	Duration	BaseStationNumber
123456	1532574644	654321	60	1
123456	1532574704	236789	60	1
234567	1532574714	765432	10	1
234567	1532574734	123456	20	3
345678	1532574795	123456	5	2
345678	1532574861	123456	100	2
456789	1532584054	345678	200	3
456789	1532585054	123456	200	3
456789	1532586054	234567	200	3
456789	1532587054	123456	200	3

您可以在被叫号码列上建立本地二级索引LocalIndexOnBeCalledNumber上进行查询，来满足查询指定的主叫号码和指定的被叫号码之间通话记录的需求。

索引表LocalIndexOnBeCalledNumber的信息请参见下表。

> **说明** 系统会自动进行索引列补齐。即把数据表的主键添加到索引列后，共同作为索引表的主键，所以索引表中有三列主键。

PK0	Defined0	PK1	Defined1	Defined2
CellNumber	CalledNumber	StartTime（Unix时间戳）	Duration	BaseStationNumber
123456	236789	1532574704	60	1
123456	654321	1532574644	60	1
234567	123456	1532574734	20	3
234567	765432	1532574714	10	1
345678	123456	1532574795	5	2
345678	123456	1532574861	100	2
456789	123456	1532585054	200	3
456789	123456	1532587054	200	3
456789	234567	1532586054	200	3
456789	345678	1532584054	200	3

如果要查询主叫号码456789到被叫号码123456的所有话单，您只需要在调用getRange接口时，将PK0列（主叫号码）的最大值和最小值均设置为456789，Defined0列（被叫号码）的最大值和最小值均设置为123456，PK1（开始时间）的最小值设置为0， 最大值设置为INT_MAX，对数据表进行扫描即可。示例代码如下：

```
private static void getRangeFromMainTable(SyncClient client, long cellNumber, long calledNu
mber){
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    // 构造起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_0, PrimaryKeyValue.fromLong
(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_0, PrimaryKeyValue.fromLong
(calledNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.fromLong
(0));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    // 构造结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_0, PrimaryKeyValue.fromLong(c
ellNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_0, PrimaryKeyValue.fromLong(c
alledNumber));
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX);
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    String strNum = String.format("%d", cellNumber);
  String strCalledNum = String.format("%d", calledNumber);
    System.out.println("号码" + strNum + "和号码" +strCalledNum+ "的所有话单:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQue
ryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        // 如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextStart
PrimaryKey());
        } else {
            break;
        }
    }
}
```

8.3. 接口

通过本文您可以了解二级索引的接口。

二级索引相关接口请参见下表。

接口	说明

接口	说明
CreateIndex	为已存在数据表创建索引表。 说明 • 目前使用CreateIndex单独创建的索引表中可以选择包含或者不包含在数据表执行CreateIndex前已存在的数据。 • 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。
GetRow	单行读取索引表中数据。
GetRange	范围读索引表中数据。
DropIndex	删除数据表上指定的索引表。 说明 使用DeleteTable接口删除数据表前，必须先删除数据表下的索引，否则数据表将删除失败。

8.4. 使用SDK

8.4.1. 全局二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

使用

您可以使用如下语言的SDK实现全局二级索引功能。

- Java SDK：全局二级索引
- Go SDK：全局二级索引
- Python SDK：全局二级索引
- Node.js SDK：全局二级索引
- .NET SDK：全局二级索引
- PHP SDK：全局二级索引

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建一个索引表。

说明 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
mainTableName	数据表名称。
indexMeta	索引表的结构信息，包括如下内容： ○ indexName：索引表名称。 ○ primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ definedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 ○ indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ■ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ■ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。
includeBaseData	索引表中是否包含数据表中已存在的数据。 当CreateIndexRequest中的最后一个参数includeBaseData设置为true时，表示包含存量数据；设置为false时，表示不包含存量数据。

- 示例

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX2_NAME); //设置索引表名称。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列，设置DEFINED_COL_NAME_1列为索引表的第一列主键。
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2); //为索引表添加主键列，设置PRIMARY_KEY_NAME_2列为索引表的第二列主键。
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); //为索引表添加属性列，设置DEFINED_COL_NAME_2列为索引表的属性列。
    //CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); //添加索引表到数据表，包含存量数据。
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); //添加索引表到数据表，不包含存量数据。
    /**通过将IncludeBaseData参数设置为true，创建索引表后会开启数据表中存量数据的同步，然后可以通过索引表查询全部数据，
        同步时间和数据量的大小有一定的关系。
    */
    //request.setIncludeBaseData(true);
    client.createIndex(request); //创建索引表。
}
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - tableName需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  - 参数

    使用GetRange接口读取索引表中数据时有如下注意事项：

    - tableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

  - 示例

    当需要返回的属性列在索引表中时，可以直接读取索引表获取数据。

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; //设置索引表名称。
    //设置起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); //设置数据表主键最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); //设置数据表主键最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("扫描索引表的结果为:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

当需要返回的属性列不在索引表中时，请自行反查数据表获取数据。

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; //设置索引表名称。
    //设置起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); //设置数据表主键最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
```

```
MIN); //设置数据表主键最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBu
ilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, ke2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); //根据索引表主键构造数据
表主键。
            //反查数据表。
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, ma
inTablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME_3); //设置读取数据表的DEFINED_COL_N
AME_3列。
            //设置读取最新版本。
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
mainTableName	数据表名称。
indexName	索引表名称。

- 示例

```
private static void deleteIndex(SyncClient client) {
    DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME, INDEX_NAME); //设置数据表名称和索引表名称。
    client.deleteIndex(request); //删除索引表。
}
```

8.4.2. 本地二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

使用

您可以使用如下语言的SDK实现本地二级索引功能。

- Java SDK：本地二级索引
- Go SDK：本地二级索引
- Python SDK：本地二级索引
- Node.js SDK：本地二级索引

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建一个索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
mainTableName	数据表名称。

参数	说明
indexMeta	索引表的结构信息，包括如下内容： ○ indexName：索引表名称。 ○ primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ definedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 ○ indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ■ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ■ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。
includeBaseData	索引表中是否包含数据表中已存在的数据。 当CreateIndexRequest中的最后一个参数includeBaseData设置为true时，表示包含存量数据；设置为false时，表示不包含存量数据。

- 示例

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME); //设置索引表名称。
    indexMeta.setIndexType(IT_LOCAL_INDEX);     //设置索引类型为本地二级索引（IT_LOCAL_INDEX）。
    indexMeta.setIndexUpdateMode(IUM_SYNC_INDEX);  //设置索引更新模式为同步更新（IUM_SYNC_INDEX）。当索引类型为本地二级索引时，索引更新模式必须为同步更新。
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1); //为索引表添加主键列。索引表的第一列主键必须与数据表的第一列主键相同。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_2); //为索引表添加主键列，设置DEFINED_COL_NAME_2列为索引表的第二列主键。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列，设置DEFINED_COL_NAME_1列为索引表的第三列主键。
    //CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); //添加索引表到数据表，包含存量数据。
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); //添加索引表到数据表，不包含存量数据。
    /**通过将IncludeBaseData参数设置为true，创建索引表后会开启数据表中存量数据的同步，然后可以通过索引表查询全部数据，
        同步时间和数据量的大小有一定的关系。
    */
    //request.setIncludeBaseData(true);
    client.createIndex(request); // 创建索引表。
}
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - tableName需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  - 参数

    使用GetRange接口读取索引表中数据时有如下注意事项：

    - tableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

  - 示例

    当需要返回的属性列在索引表中时，可以直接读取索引表获取数据。

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; //设置索引表名称。
    //设置起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); //设置数据表主键最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("扫描索引表的结果为:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

当需要返回的属性列不在索引表中时，请自行反查数据表获取数据。

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; //设置索引表名称。
    //设置起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
```

```
MIN); //设置数据表主键最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBu
ilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, pk2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); //根据索引表主键构造数据
表主键。
            //反查数据表。
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, ma
inTablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME3); // 读取主表的DEFINED_COL_NAME3
列
            //设置读取最新版本。
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
mainTableName	数据表名称。
indexName	索引表名称。

- 示例

```
private static void deleteIndex(SyncClient client) {
    DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME, INDEX_NAME); //设置数据表名称和索引表名称。
    client.deleteIndex(request); //删除索引表。
}
```

8.5. 附录

全局二级索引附录。

创建主表及索引表

```
private static final String TABLE_NAME = "CallRecordTable";
    private static final String INDEX0_NAME = "IndexOnBeCalledNumber";
    private static final String INDEX1_NAME = "IndexOnBaseStation1";
    private static final String INDEX2_NAME = "IndexOnBaseStation2";
    private static final String PRIMARY_KEY_NAME_1 = "CellNumber";
    private static final String PRIMARY_KEY_NAME_2 = "StartTime";
    private static final String DEFINED_COL_NAME_1 = "CalledNumber";
    private static final String DEFINED_COL_NAME_2 = "Duration";
    private static final String DEFINED_COL_NAME_3 = "BaseStationNumber";
    private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyTy
pe.INTEGER));
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyTy
pe.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColum
nType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColum
nType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_3, DefinedColum
nType.INTEGER));
        int timeToLive = -1; // 数据的过期时间，单位秒，-1代表永不过期．带索引表的主表数据过期时间
必须为-1。
        int maxVersions = 1; // 保存的最大版本数，带索引表的主表最大版本数必须为1。
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
        IndexMeta indexMeta0 = new IndexMeta(INDEX0_NAME);
        indexMeta0.addPrimaryKeyColumn(DEFINED_COL_NAME_1);
        indexMetas.add(indexMeta0);
        IndexMeta indexMeta1 = new IndexMeta(INDEX1_NAME);
        indexMeta1.addPrimaryKeyColumn(DEFINED_COL_NAME_3);
        indexMeta1.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
        indexMetas.add(indexMeta1);
        IndexMeta indexMeta2 = new IndexMeta(INDEX2_NAME);
        indexMeta2.addPrimaryKeyColumn(DEFINED_COL_NAME_3);
        indexMeta2.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
        indexMeta2.addDefinedColumn(DEFINED_COL_NAME_2);
        indexMetas.add(indexMeta2);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexM
etas);
        client.createTable(request);
    }
```

9.SQL查询

9.1. SQL概述

通过SQL查询功能，您可以对表格存储中数据进行复杂的查询和高效的分析，为多数据引擎提供统一的访问接口。

背景信息

表格存储（Tablestore）是阿里云自研的多模型结构化数据存储，提供海量结构化数据存储以及快速的查询和分析服务。表格存储的分布式存储和强大的索引引擎能够支持PB级存储、千万TPS以及毫秒级延迟的服务能力。使用表格存储您可以方便的存储和查询海量数据。

为了满足用户业务平滑迁移到表格存储以及使用SQL方式访问表格存储的需求，表格存储在传统的NoSQL结构化存储之上，提供云原生的SQL引擎能力，SQL查询兼容MySQL的查询语法，同时提供基础的SQL DDL建表能力。对于已有数据表，通过执行CREATE TABLE一键自动建立映射关系后，您可以使用SQL方式访问表中数据。

SQL查询功能支持通过多元索引来快速查询满足查询条件的数据。使用SQL查询时，系统会根据SQL语句自动选择合适的方式来加速SQL访问。

SQL查询功能适用于在海量数据中进行在线数据访问类型的场景，SQL访问的延时从毫秒、秒到分钟级别不等，包括基于数据表主键的Point Query （GetRow操作）、多元索引的精确查询（TermQuery ）等以及通过多元索引的统计聚合能力进行查询，例如数据表中满足某个条件的个数、某列求和等。

基本概念

SQL的使用会涉及到很多传统数据库中的概念，此处介绍下相关概念以及与现有表格存储概念的映射关系。

名词	描述
数据库	按照数据结构来组织、存储和管理数据的仓库。一个数据库中可以包含一个或者多个表，映射为表格存储的实例。
表	由行和列组成，映射为表格存储的表。
索引	映射为表格存储的二级索引或者多元索引。

功能特性

- SQL功能
  - 支持单次请求单条SQL。
  - 支持基础的DDL语句，主要包括创建表的映射关系、创建多元索引的映射关系、更新映射表属性列、删除映射关系和查询表的描述信息。
  - 支持DQL操作，主要包括查询数据。
  - 支持基础的Database Administration功能，主要包括列出表名称列表和查询索引描述信息。
- 字符集和排序规则
  - 字符集：UTF-8
  - 排序规则：二进制排序规则
- 操作符

支持算术运算符、关系运算符、逻辑运算符等SQL操作符。更多信息，请参见SQL操作符。

- 存量表绑定

  通过CREATE TABLE语句，您可以为存量表创建映射关系。创建映射关系时，请确保主键和存量表的主键一致，属性列和存量表的属性列类型以及预定义列类型一致。关于数据类型映射关系的更多信息，请参见数据类型映射。

注意事项

在使用SQL查询时，不支持事务功能。

使用限制

更多信息，请参见SQL使用限制。

SQL中大小写

表格存储的SQL引擎遵循通用的SQL规范，对列名大小写不敏感，例如操作`SELECT Aa FROM exampletable;`语句与`SELECT aa FROM exampletable;`语句是等价的。

由于表格存储中原始表的列名大小写敏感，当使用SQL时，原始表的列名会统一转换为小写字母进行匹配，即如果要操作表格存储表中的Aa列，在SQL中使用AA、aa、aA、Aa均可，因此表格存储原始表的列名不能同时为AA、aa、aA和Aa。

保留字与关键字

表格存储将SQL语句中的关键字作为保留字。如果在命名表或者列时需要使用关键字，则请添加``符号对关键字进行转义。关键字不区分大小写。

关于保留字与关键字的更多信息，请参见保留字与关键字。

SQL与多元索引功能对比

多元索引功能		SQL函数/语句
精确查询		等于（=）
范围查询		大于（>）、小于（<）、大于等于（>=）、小于等于（<=）、BETWEEN ... AND ...
多条件组合查询	MustQueries	AND
	MustNotQueries	!=
	ShouldQueries	OR
排序和翻页	FieldSort	ORDER BY
	SetLimit	LIMIT
	最小值	MIN()
	最大值	MAX()
	和	SUM()

多元索引功能		SQL函数/语句
统计聚合	平均值	AVG()
	统计行数	COUNT()
	统计去重行数	COUNT(DISTINCT)
	获取统计聚合分组中的行	ANY_VALUE()
	字段值分组	GROUP BY

计费

更多信息，请参见SQL查询计量计费。

9.2. SQL支持功能说明

本文介绍了表格存储SQL引擎对SQL语句的支持情况。

> **注意** SQL查询功能从2022年05月26日正式开始收费。如果使用过程中遇到问题，请提交工单或者加入钉钉群23307953（表格存储技术交流群-2）联系我们。

SQL语句	描述	支持情况
CREATE TABLE	当数据表存在时，创建表的映射关系。	支持
	当多元索引存在时，创建多元索引的映射关系。	支持
ALTER TABLE	添加或者删除属性列。	支持
SHOW TABLES	列出当前数据库中的表名称列表。	支持
SELECT	查询数据。	支持
DROP MAPPING TABLE	删除表映射关系。	支持
DROP TABLE	删除数据表。	即将支持
CREATE INDEX	创建索引。	暂不支持
SHOW INDEX	查询索引描述信息。	支持
INSERT	写入数据。	暂不支持
SELECT JOIN	跨表JOIN数据查询。	暂不支持

9.3. 数据类型映射

本文介绍了SQL中字段数据类型、数据表中字段数据类型和多元索引中字段数据类型的映射关系。使用过程时请确保SQL中和数据表中字段数据类型相匹配。

> **说明**
> - SQL中主键列的数据类型使用VARBINARY和VARCHAR时，建议最大长度设置为1024，即使用VARBINARY(1024)和VARCHAR(1024)。
> - SQL中的BIGINT和数据表中的Integer均为64位整型。
> - SQL中只有BIGINT、VARBINARY和VARCHAR数据类型支持作为主键列的数据类型。

SQL中字段数据类型	数据表中字段数据类型	多元索引中的字段数据类型
BIGINT	Integer	Integer
• VARBINARY（主键） • MEDIUMBLOB（属性列）	Binary	Binary
• VARCHAR （主键） • MEDIUMTEXT（属性列）	String	KeyWord
		Text
DOUBLE	Double	Double
BOOL	Boolean	Boolean

9.4. 使用控制台

表格存储支持使用SQL查询功能快速查询数据。使用控制台创建映射关系后，您可以执行SELECT语句快速查询所需数据。

前提条件

- 如果要使用RAM用户进行操作，请确保已创建RAM用户，并为RAM用户授予所有SQL操作权限，即在自定义权限策略中配置 `"Action": "ots:SQL*"` 。具体操作，请参见配置RAM用户权限。
- 已创建数据表。

注意事项

目前支持使用SQL查询功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

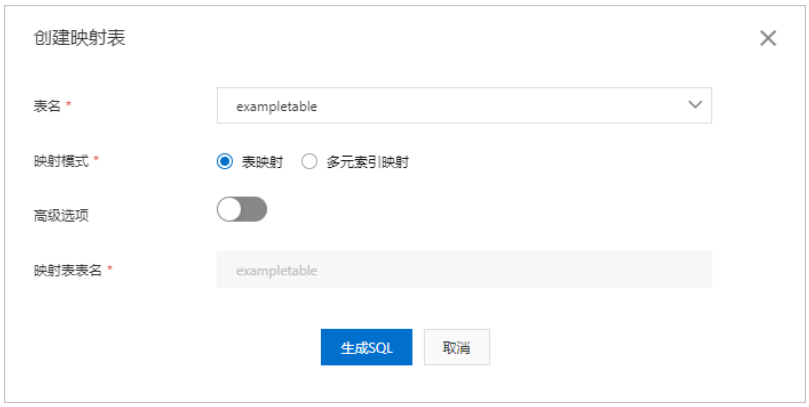
创建映射关系

1. 登录表格存储控制台。
2. 在页面上方，选择地域，例如华东1（杭州）、华南1（深圳）等。
3. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
4. 在**SQL查询**页签，创建映射关系。

> **说明** 您也可以直接手动编写创建映射关系的SQL语句。更多信息，请参见创建表的映射关系和创建多元索引的映射关系。

i. 单击 ＋ 图标。

> **说明** 当不存在映射表时，单击**SQL查询**页签，系统会自动弹出**创建映射表**对话框。



ii. 在**创建映射表**对话框，根据下表说明配置参数。

参数	描述
表名	数据表名称。
映射模式	创建映射关系的模式。取值范围如下： ■ *表映射*（默认）：为已存在的数据表创建映射关系。 ■ *多元索引映射*：为已存在的多元索引创建映射关系。
高级选项	用于配置映射表的一致性模式和是否使用不准确聚合。打开**高级选项**开关，即可进行配置。只有当映射模式选择为**表映射**时才能配置。
一致性模式	执行引擎支持的一致性模式。取值范围如下： ■ *最终一致*（默认）：执行的查询结果满足最终一致。此时新数据写入后会在几秒后影响到查询结果。 ■ *强一致性*：执行的查询结果满足强一致性。此时新数据写入后立刻影响到查询结果。 只有打开了高级选项开关后才能配置。
不准确聚合	是否允许通过牺牲聚合操作的精准度提升查询性能。取值范围如下： ■ *是*（默认）：允许通过牺牲聚合操作的精度提升查询性能。 ■ *否*：不允许通过牺牲聚合操作的精度提升查询性能 只有开启了高级选项开关后才能配置。
多元索引表	映射表绑定的多元索引名称。只有当映射模式选择为**多元索引映射**时才能配置。
映射表表名	映射表名称。 ■ 当映射模式选择为**表映射**时，映射表表名与数据表名称相同，不能更改。 ■ 当映射模式选择为**多元索引映射**时，需要填写映射表名称。

iii. 单击**生成SQL**。

系统会自动生成创建映射表的SQL语句。SQL示例如下：

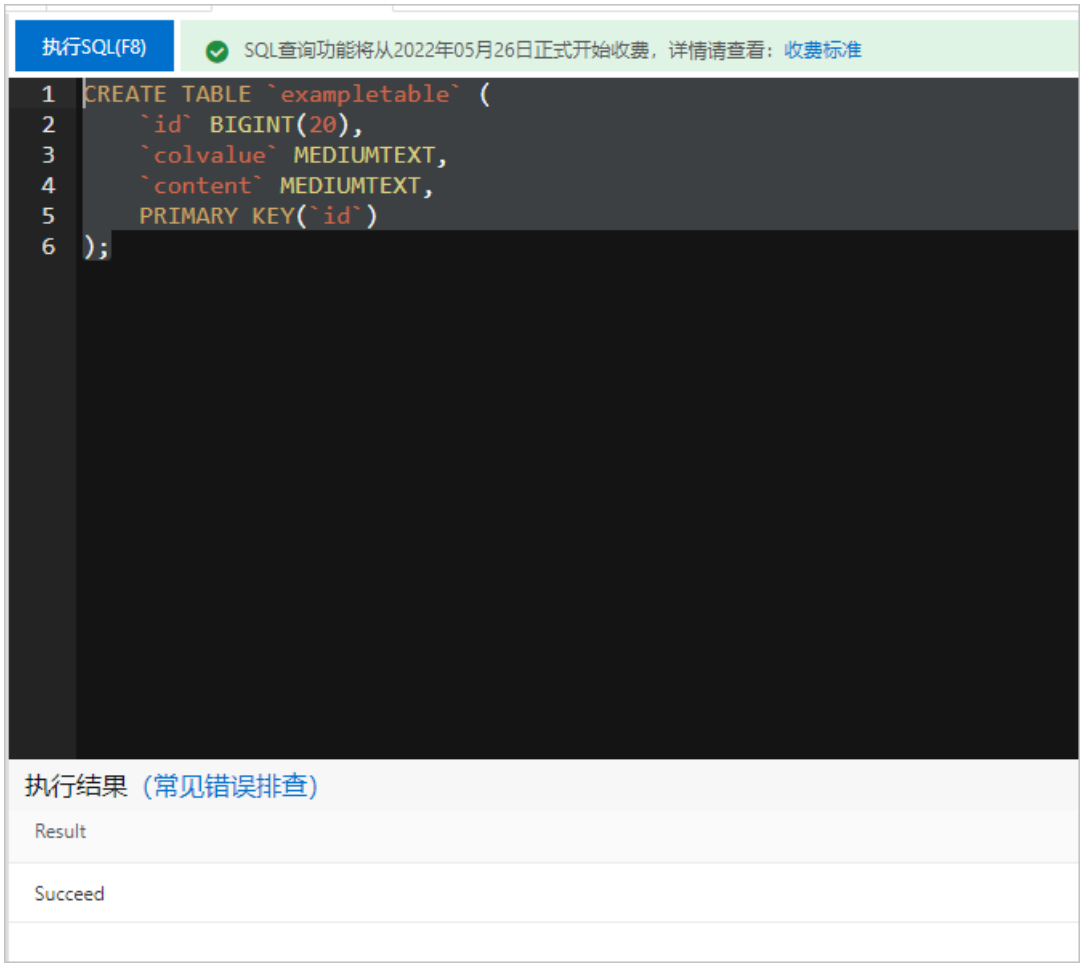
```
CREATE TABLE `exampletable` (
    `id` BIGINT(20),
    `colvalue` MEDIUMTEXT,
    `content` MEDIUMTEXT,
    PRIMARY KEY(`id`)
);
```

注意 请确保映射关系中字段数据类型和数据表中字段数据类型相匹配。关于数据类型映射的更多信息，请参见数据类型映射。

iv. 根据实际需要修改映射表的Schema后，按住鼠标左键拖动选中一条SQL语句并单击**执行SQL(F8)**。

执行成功后，在**执行结果**区域会显示执行结果。

> 注意
>
> - 创建映射表时设置的Schema中需要包括后续查询数据所需的列。
> - 执行SQL语句时，请选中一条所需SQL语句，否则系统默认执行第一条SQL语句。
> - 执行SQL语句时，一次只能选中一条SQL语句，否则系统会报错。



查询数据

创建映射表后，在**SQL查询**页签，执行SELECT语句查询所需数据。更多信息，请参见查询数据。

9.5. 使用SDK

使用sqlQuery接口，您可以通过SQL访问表格存储。

注意事项

目前支持使用SQL查询功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

前提条件

- 如果要使用RAM用户进行操作，请确保已创建RAM用户，并为RAM用户授予所有SQL操作权限，即在自定义权限策略中配置 `"Action": "ots:SQL*"` 。具体操作，请参见配置RAM用户权限。
- 已获取AccessKey（包括AccessKey ID和AccessKey Secret）。具体操作，请参见获取AccessKey。
- 已创建数据表。
- 已初始化Client。具体操作，请参见初始化。

使用

您可以使用如下语言的SDK实现SQL查询功能。

- Java SDK
- Go SDK

参数

参数	说明
query	SQL语句，请根据所需功能进行设置。

示例

通过create table语句为已存在的表创建映射关系后，您可以通过select语句查询表中数据。

1. 通过create table语句为已存在的表创建映射关系。

   使用 `create table test_table (pk varchar(1024), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primary key(pk))` 语句创建test_table表的映射关系。

   ```
   private static void createTable(SyncClient client) {
       // 创建SQL请求。
       SQLQueryRequest request = new SQLQueryRequest("create table test_table (pk varchar(1024), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primary key(pk))");
       // 获取SQL的响应结果。
       SQLQueryResponse response = client.sqlQuery(request);
   }
   ```

2. 通过select语句查询表中数据。

   使用 `select pk, long_value, double_value, string_value, bool_value from test_table limit 20` 语句查询test_table表中数据且最多返回20行数据。系统会返回查询语句的请求类型、返回值Schema、返回结果等信息。

```
private static void queryData(SyncClient client) {
    // 创建SQL请求。
    SQLQueryRequest request = new SQLQueryRequest("select pk, long_value, double_value,
string_value, bool_value from test_table limit 20");
    // 获取SQL的响应结果。
    SQLQueryResponse response = client.sqlQuery(request);
    // 获取SQL的请求类型。
    System.out.println("response type: " + response.getSQLStatementType());
    // 获取SQL返回值的Schema。
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table meta: " + tableMeta.getSchema());
    // 获取SQL的返回结果。
    SQLResultSet resultSet = response.getSQLResultSet();
    System.out.println("response resultset:");
    while (resultSet.hasNext()) {
        SQLRow row = resultSet.next();
        System.out.println(row.getString(0) + ", " + row.getString("pk") + ", " +
                           row.getLong(1) + ", " + row.getLong("long_value") + ", " +
                           row.getDouble(2) + ", " + row.getDouble("double_value") + ",
" +
                           row.getString(3) + ", " + row.getString("string_value") + ",
" +
                           row.getBoolean(4) + ", " + row.getBoolean("bool_value"));
    }
}
```

返回结果示例如下：

```
response type: SQL_SELECT
response table meta: [pk:STRING, long_value:INTEGER, double_value:DOUBLE, string_value:
STRING, bool_value:BOOLEAN]
response resultset:
binary_null, binary_null, 1, 1, 1.0, 1.0, a, a, false, false
bool_null, bool_null, 1, 1, 1.0, 1.0, a, a, null, null
double_null, double_null, 1, 1, null, null, a, a, true, true
long_null, long_null, null, null, 1.0, 1.0, a, a, true, true
string_null, string_null, 1, 1, 1.0, 1.0, null, null, false, false
```

9.6. 使用JDBC

9.6.1. JDBC连接表格存储

本文介绍如何使用JDBC访问表格存储。

注意事项

目前支持使用SQL查询功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

前提条件

- 如果要使用RAM用户进行操作，请确保已创建RAM用户，并为RAM用户授予所有SQL操作权限，即在自定

义权限策略中配置 `"Action": "ots:SQL*"` 。具体操作，请参见配置RAM用户权限。

- 已获取AccessKey（包括AccessKey ID和AccessKey Secret）。具体操作，请参见获取AccessKey。
- 已创建数据表并为数据表创建映射关系。具体操作，请分别参见创建数据表和创建表的映射关系。

步骤一：安装JDBC驱动

您可以通过以下两种方式安装JDBC驱动。

- 下载表格存储JDBC驱动并导入到项目中。具体下载路径请参见表格存储JDBC驱动。
- 在Maven项目中加入依赖项

  在Maven工程中使用表格存储JDBC驱动，只需在pom.xml中加入相应依赖即可。以5.13.5版本为例，在<dependencies>内加入如下内容：

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>tablestore-jdbc</artifactId>
  <version>5.13.5</version>
</dependency>
```

步骤二：使用JDBC直连

1. 使用 `Class.forName()` 加载表格存储JDBC驱动。

   表格存储JDBC驱动名称为 `com.alicloud.openservices.tablestore.jdbc.OTSDriver` 。

```
Class.forName("com.alicloud.openservices.tablestore.jdbc.OTSDriver");
```

2. 使用JDBC连接表格存储实例。

```
String url = "jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance";
String user = "************************";
String password = "********************************";
Connection conn = DriverManager.getConnection(url, user, password);
```

   参数说明请参见下表。

参数	示例	说明

参数	示例	说明
url	jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance	表格存储JDBC的URL。格式为 `jdbc:ots:schema://[accessKeyId:accessKeySecret@]endpoint/instanceName[?param1=value1&...&paramN=valueN]` 。主要字段说明如下： ○ schema（必选）：表格存储JDBC驱动使用的协议，一般设置为https。 ○ accessKeyId:accessKeySecret（可选）：阿里云账号或者RAM用户的AccessKey ID和AccessKey Secret。 ○ endpoint（必选）：实例的服务地址。更多信息，请参见服务地址。 ○ instanceName（必选）：实例名称。 其他常用配置项的说明，请参见配置项。
user	**********	阿里云账号或者RAM用户的AccessKey ID。
password	**********	阿里云账号或者RAM用户的AccessKey Secret。

您可以通过URL或者Properties方式传递AccessKey和配置项，此处以通过公网访问华东1（杭州）地域下myinstance实例为例介绍。

- 通过URL传递AccessKey和配置项

```
DriverManager.getConnection("jdbc:ots:https://**********:**********@myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance?enableRequestCompression=true");
```

- 通过Properties传递AccessKey和配置项

```
Properties info = new Properties();
info.setProperty("user", "**********");
info.setProperty("password", "**********");
info.setProperty("enableRequestCompression", "true");
DriverManager.getConnection("jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance", info);
```

3. 执行SQL语句。

   您可以使用createStatement或者prepareStatement方法创建SQL语句。

   > **说明** 当前支持的SQL语句请参见SQL支持功能说明。

   - 使用createStatement创建SQL语句

```
// 设置SQL语句，此处以查询test_table表中id列和name列的数据为例介绍，请根据实际需要设置。
String sql = "SELECT id,name FROM test_table";
Statement stmt = conn.createStatement();
ResultSet resultSet = stmt.executeQuery(sql);
while (resultSet.next()) {
    String id = resultSet.getString("id");
    String name = resultSet.getString("name");
    System.out.println(id);
    System.out.println(name);
}
resultSet.close();
stmt.close();
```

- 使用prepareStatement创建SQL语句

```
// 设置SQL语句，此处以查询test_table表中pk为指定值的数据为例介绍，请根据实际需要设置。
String sql = "SELECT * FROM test_table WHERE pk = ?";
PreparedStatement stmt = connection.prepareStatement(sql);
stmt.setLong(1, 1);
ResultSet resultSet = stmt.executeQuery();
ResultSetMetaData metaData = resultSet.getMetaData();
while (resultSet.next()) {
    int columnCount = metaData.getColumnCount();
    for (int i=0; i< columnCount;i++) {
        String columnName = metaData.getColumnName(i+1);
        String columnValue = resultSet.getString(columnName);
        System.out.println(columnName);
        System.out.println(columnValue);
    }
}
resultSet.close();
stmt.close();
```

完整示例

查询华东1（杭州）地域下myinstance实例中test_table的所有数据。

```
public class Demo {
    public static void main(String[] args) throws SQLException, ClassNotFoundException {
        Class.forName("com.alicloud.openservices.tablestore.jdbc.OTSDriver");
        String url = "jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance";
        String user = "************************";
        String password = "*******************************";
        Connection conn = DriverManager.getConnection(url, user, password);
        String sql = "SELECT * FROM test_table";
        Statement stmt = conn.createStatement();
        ResultSet resultSet = stmt.executeQuery(sql);
        ResultSetMetaData metaData = resultSet.getMetaData();
        while (resultSet.next()) {
            int columnCount = metaData.getColumnCount();
            for (int i=0; i< columnCount;i++) {
                String columnName = metaData.getColumnName(i+1);
                String columnValue = resultSet.getString(columnName);
                System.out.println(columnName);
                System.out.println(columnValue);
            }
        }
        resultSet.close();
        stmt.close();
        conn.close();     // 请务必关闭连接，否则程序无法退出。
    }
}
```

配置项

表格存储JDBC驱动基于表格存储的Java SDK实现，通过JDBC您可以修改Java SDK的配置项。常用配置项的详细说明请参见下表。

配置项	示例值	说明
enableRequestCompression	false	是否压缩请求数据。取值范围如下： • true：压缩请求数据。 • false（默认）：不压缩请求数据。
enableResponseCompression	false	是否压缩响应数据。取值范围如下： • true：压缩响应数据。 • false（默认）：不压缩响应数据。
ioThreadCount	2	HttpAsyncClient的IOReactor的线程数，默认与CPU核数相同。
maxConnections	300	允许打开的最大HTTP连接数。
socketTimeoutInMillisecond	30000	Socket层传输数据的超时时间。单位为毫秒。0表示无限等待。
connectionTimeoutInMillisecond	30000	建立连接的超时时间。单位为毫秒。0表示无限等待。

配置项	示例值	说明
retryThreadCount	1	用于执行错误重试的线程池中线程个数。
syncClientWaitFutureTimeoutInMillis	-1	异步等待的超时时间。单位为毫秒。
connectionRequestTimeoutInMillisecond	60000	发送请求的超时时间。单位为毫秒。

数据类型转换

表格存储支持Integer（整型）、Double（浮点数）、String（字符串）、Binary（二进制）和Boolean（布尔值）五种数据类型。通过Java SDK使用JDBC直连表格存储时，JDBC驱动能够对Java类型和表格存储数据类型进行自动转换。

- Java类型转换为表格存储数据类型

  当使用PreparedStatement方法为SQL语句中的参数赋值时，Java中Byte、Short、Int、Long、BigDecimal、Float、Double、String、CharacterStream、Bytes、Boolean类型均能传递给表格存储SQL引擎。

```
PreparedStatement stmt = connection.prepareStatement("SELECT * FROM t WHERE pk = ?");
stmt.setLong(1, 1);                                  // 支持的类型转换。
stmt.setURL(1, new URL("https://aliyun.com/"));      // 不支持的类型转换，系统会抛出异常。
```

- 表格存储数据类型转换为Java类型

  当使用ResultSet方法获取SQL返回结果时，表格存储数据类型自动转换为Java数据类型的注意事项请参见下表。

表格存储数据类型	转换原则说明
Integer Double	○ 转换为整型时，如果值超出类型的数值范围，则系统会抛出异常。 ○ 转换为浮点数时，转换后的值会丢失精度。 ○ 转换为字符串或者二进制时，等效于toString()。 ○ 转换为布尔值时，如果值为非0值，则转换后的值为真。
String Binary	○ 转换为整型或者浮点数时，如果解析失败，则系统会抛出异常。 ○ 转换为布尔值时，如果字符串为true，则转换后的值为真。
Boolean	○ 转换为整型或者浮点数时，如果值为真，则转换后的值为1；如果值为假，则转换后的值为0。 ○ 转换为字符串或者二进制时，等效于toString()。

```
Statement stmt = conn.createStatement();
ResultSet resultSet = stmt.executeQuery("SELECT count(*) FROM t");
while (resultSet.next()) {
    resultSet.getLong(1);                // 支持的类型转换。
    resultSet.getCharacterStream(1);     // 不支持的类型转换，系统会抛出异常。
}
```

表格存储数据类型和Java类型转换的支持情况请参见下表。

> **说明** “✓”表示正常转换，“~”表示可能抛出异常，“×”表示无法转换。

类型转换	Integer	Double	String	Binary	Boolean
Byte	~	~	~	~	✓
Short	~	~	~	~	✓
Int	~	~	~	~	✓
Long	✓	~	~	~	✓
BigDecimal	✓	✓	~	~	✓
Float	✓	✓	~	~	✓
Double	✓	✓	~	~	✓
String	✓	✓	✓	✓	✓
CharacterStream	×	×	✓	✓	×
Bytes	✓	✓	✓	✓	✓
Boolean	✓	✓	✓	✓	✓

9.6.2. 通过Hibernate使用

本文介绍了如何通过Hibernate使用表格存储的JDBC驱动来快速访问表格存储。

背景信息

Hibernate是面向Java环境的对象/关系映射（ORM）解决方案，主要负责从Java类到数据库表的映射以及从Java数据类型到SQL数据类型的映射，同时还支持数据查询。使用Hibernate能有效减少在SQL和JDBC中手动处理数据所花费的时间。更多信息，请参见Hibernate官网文档。

注意事项

目前支持使用SQL查询功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

前提条件

- 如果要使用RAM用户进行操作，请确保已创建RAM用户，并为RAM用户授予所有SQL操作权限，即在自定义权限策略中配置 `"Action": "ots:SQL*"` 。具体操作，请参见配置RAM用户权限。
- 已获取AccessKey（包括AccessKey ID和AccessKey Secret）。具体操作，请参见获取AccessKey。
- 已创建数据表并为数据表创建映射关系。具体操作，请分别参见创建数据表和创建表的映射关系。

步骤一：安装JDBC驱动

您可以通过以下两种方式安装JDBC驱动。

- 下载表格存储JDBC驱动并导入到项目中。具体下载路径请参见表格存储JDBC驱动。
- 在Maven项目中加入依赖项

  在Maven工程中使用表格存储JDBC驱动，只需在pom.xml中加入相应依赖即可。以5.13.5版本为例，在<dependencies>内加入如下内容：

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>tablestore-jdbc</artifactId>
  <version>5.13.5</version>
</dependency>
```

步骤二：安装Hibernate

您可以通过以下两种方式安装Hibernate。

- 下载Hibernate安装包（即hibernate-core-x.x.x.jar）并导入到项目中。具体下载路径请参见Hibernate安装包。

  hibernate-core-x.x.x.jar中的 `x.x.x` 表示Hibernate的版本号，请根据实际下载所需版本的安装包。

- 在Maven项目中加入依赖项

  在Maven工程中使用Hibernate，只需在pom.xml中加入相应依赖即可。以3.6.3.Final版本为例，在<dependencies>内加入如下内容：

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>3.6.3.Final</version>
</dependency>
```

步骤三：映射SQL字段

完成数据库字段对应Java Bean创建后，通过映射配置文件将Java Bean的成员变量和数据表字段一一映射。

1. 创建数据表字段对应的Java Bean。

```
package hibernate;
public class Trip {
    private long tripId;
    private long duration;
    private String startDate;
    private String endDate;
    private long startStationNumber;
    private long endStationNumber;
    private String startStation;
    private String endStation;
    private String bikeNumber;
    private String memberType;
    // ...
}
```

2. 创建映射配置文件，并在映射配置文件中将Java Bean的成员变量和数据表字段相对应。此处以在hibernate目录下创建Trip.hbm.xml为例介绍。

   由于目前表格存储SQL暂时不支持数据插入和更新，因此insert属性和update属性均需要设置为false。关于SQL类型支持情况，请参见数据类型映射。关于SQL功能支持情况，请参见SQL支持功能说明。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <!--此处的类名必须与实际类名称一致。-->
    <class name="hibernate.Trip" table="trips">
        <!-- id元素中配置的字段为数据表中的主键列。-->
        <id name="tripId" column="trip_id" type="long"/>
        <!-- property元素中配置的字段均为数据表中的属性列，属性列必须禁止插入和更新，即insert属性和update属性均需要设置为false。-->
        <property name="duration" column="duration" type="long" insert="false" update="false"/>
        <property name="startDate" column="start_date" type="string" insert="false" update="false"/>
        <property name="endDate" column="end_date" type="string" insert="false" update="false"/>
        <property name="startStationNumber" column="start_station_number" type="long" insert="false" update="false"/>
        <property name="endStationNumber" column="end_station_number" type="long" insert="false" update="false"/>
        <property name="startStation" column="start_station" type="string" insert="false" update="false"/>
        <property name="endStation" column="end_station" type="string" insert="false" update="false"/>
        <property name="bikeNumber" column="bike_number" type="string" insert="false" update="false"/>
        <property name="memberType" column="member_type" type="string" insert="false" update="false"/>
    </class>
</hibernate-mapping>
```

步骤四：构建SessionFactory

完成Hibernate配置文件设置后，通过加载Hibernate配置文件来创建SessionFactory。

1. 创建Hibernate配置文件hibernate.cfg.xml后添加如下内容。并根据实际修改对应配置项。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.alicloud.openservices.tablestore.jdbc.OTSDriver</property>
        <property name="hibernate.connection.url">jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance</property>
        <property name="hibernate.connection.username">************************</property>
        <property name="hibernate.connection.password">********************************</property>
        <property name="hibernate.connection.autocommit">true</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- 设置为映射配置文件的路径。-->
        <mapping resource="hibernate/Trip.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

配置项说明请参见下表。

配置项	类型	是否必选	示例	描述
hibernate.connection.driver_class	class	是	com.alicloud.openservices.tablestore.jdbc.OTSDriver	表格存储JDBC驱动的类名，设置为com.alicloud.openservices.tablestore.jdbc.OTSDriver。
hibernate.connection.url	string	是	jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance	实例访问地址。格式为 `jdbc:ots:endpoint/instanceName` ，其中endpoint为实例的服务地址。更多信息，请参见服务地址。instanceName为实例名称，请根据实际修改。 填写时必须加上前缀 `jdbc:ots:` 。
hibernate.connection.username	string	是	**********	阿里云账号或者RAM用户的AccessKey ID。
hibernate.connection.password	string	是	**********	阿里云账号或者RAM用户的AccessKey Secret。

配置项	类型	是否必选	示例	描述
hibernate.connection.autocommit	boolean	是	true	是否自动提交。由于表格存储目前暂不支持事务，因此该配置项必须设置为true。
hibernate.dialect	string	是	org.hibernate.dialect.MySQLDialect	表格存储SQL继承了MySQL语法，设置为org.hibernate.dialect.MySQLDialect。

2. 通过加载Hibernate配置文件来构建SessionFactory。

```
SessionFactory factory = new Configuration().
  configure("hibernate/hibernate.cfg.xml").
  buildSessionFactory();
```

步骤五：创建Session查询数据

```
Session session = factory.openSession();
Trip trip = (Trip) session.get(Trip.class, 99L);
System.out.println("trip id: " + trip.getTripId());
System.out.println("start date: " + trip.getStartDate());
System.out.println("end date: " + trip.getEndDate());
System.out.println("duration: " + trip.getDuration());
session.close();
factory.close();
```

完整示例

查询表中主键列值为99的行数据，并获取指定列的值。

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import hibernate.Trip;
public class HibernateDemo {
    public static void main(String[] args) {
        SessionFactory factory = new Configuration().
                configure("hibernate/hibernate.cfg.xml"). // 设置Hibernate配置文件的完整路径。
                buildSessionFactory();
        Session session = factory.openSession();
        // 设置主键值为99。如果行数据不存在，则返回null。
        Trip trip = (Trip) session.get(Trip.class, 99L);
        // 打印需要获取的列值。
        System.out.println("trip id: " + trip.getTripId());
        System.out.println("start date: " + trip.getStartDate());
        System.out.println("end date: " + trip.getEndDate());
        System.out.println("duration: " + trip.getDuration());
        session.close();
        factory.close();
    }
}
```

常见问题处理

- 问题现象：当通过Hibernate使用JDBC驱动查询数据时出现如下错误，我该如何解决？

```
Exception in thread "main" org.hibernate.HibernateException: Unable to instantiate defaul
t tuplizer [org.hibernate.tuple.entity.PojoEntityTuplizer]
  at org.hibernate.tuple.entity.EntityTuplizerFactory.constructTuplizer(EntityTuplizerFac
tory.java:108)
  at org.hibernate.tuple.entity.EntityTuplizerFactory.constructDefaultTuplizer(EntityTupl
izerFactory.java:133)
  at org.hibernate.tuple.entity.EntityEntityModeToTuplizerMapping.<init>(EntityEntityMode
ToTuplizerMapping.java:80)
  at org.hibernate.tuple.entity.EntityMetamodel.<init>(EntityMetamodel.java:322)
  at org.hibernate.persister.entity.AbstractEntityPersister.<init>(AbstractEntityPersiste
r.java:485)
  at org.hibernate.persister.entity.SingleTableEntityPersister.<init>(SingleTableEntityPe
rsister.java:133)
  at org.hibernate.persister.PersisterFactory.createClassPersister(PersisterFactory.java:
84)
  at org.hibernate.impl.SessionFactoryImpl.<init>(SessionFactoryImpl.java:286)
  .....
```

可能原因：缺少javassist-x.x.x.jar包。

解决方案：安装javassist-x.x.x.jar包，您可以通过以下两种方式安装。

- 下载javassist安装包（即javassist-x.x.x.jar）并导入到项目中。具体下载路径请参见javassist安装包。

  javassist-x.x.x.jar中的 `x.x.x` 表示javassist的版本号，请根据实际下载所需版本的安装包。

- 在Maven项目中加入依赖项

  在Maven工程中的pom.xml中加入相应依赖即可。此处以3.15.0-GA版本为例，在<dependencies>内加入如下内容：

  ```
  <!-- https://mvnrepository.com/artifact/org.javassist/javassist -->
  <dependency>
      <groupId>org.javassist</groupId>
      <artifactId>javassist</artifactId>
      <version>3.15.0-GA</version>
  </dependency>
  ```

- 问题现象：当通过Hibernate使用JDBC驱动查询数据时出现 `Message: Unknown column '{columnName}' in 'field list'` 错误，我该如何解决？

  可能原因：在SQL映射表中不存在指定列。

  解决方案：确保SQL映射表中存在指定列。您可以通过以下两种方式实现。

  - 在预定义列中添加指定列，该列会自动同步到SQL映射表中。
  - 通过CREATE TABLE语句创建映射关系时指定该列。具体操作，请参见创建表的映射关系。

9.6.3. 通过MyBatis使用

本文介绍了如何通过MyBatis使用表格存储的JDBC驱动来快速访问表格存储。

背景信息

MyBatis是一个Java数据持久层框架，支持自定义SQL、存储过程以及高级映射。使用MyBatis能免除JDBC代码以及设置参数和获取结果集的工作。更多信息，请参见MyBatis官网文档。

注意事项

目前支持使用SQL查询功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

前提条件

- 如果要使用RAM用户进行操作，请确保已创建RAM用户，并为RAM用户授予所有SQL操作权限，即在自定义权限策略中配置 `"Action": "ots:SQL*"` 。具体操作，请参见配置RAM用户权限。
- 已获取AccessKey（包括AccessKey ID和AccessKey Secret）。具体操作，请参见获取AccessKey。
- 已创建数据表并为数据表创建映射关系。具体操作，请分别参见创建数据表和创建表的映射关系。

步骤一：安装JDBC驱动

您可以通过以下两种方式安装JDBC驱动。

- 下载表格存储JDBC驱动并导入到项目中。具体下载路径请参见表格存储JDBC驱动。
- 在Maven项目中加入依赖项

  在Maven工程中使用表格存储JDBC驱动，只需在pom.xml中加入相应依赖即可。以5.13.5版本为例，在<dependencies>内加入如下内容：

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>tablestore-jdbc</artifactId>
  <version>5.13.5</version>
</dependency>
```

步骤二：安装MyBatis

您可以通过以下两种方式安装MyBatis。

- 下载MyBatis安装包（即mybatis-x.x.x.jar）并导入到项目中。具体下载路径请参见MyBatis安装包。

  mybatis-x.x.x.jar中的 `x.x.x` 表示MyBatis的版本号，请根据实际下载所需版本的安装包。

- 在Maven项目中加入依赖项

  在Maven工程中使用MyBatis，只需在pom.xml中加入相应依赖即可。以3.6.3.Final版本为例，在<dependencies>内加入如下内容：

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.9</version>
</dependency>
```

步骤三：映射SQL字段

1. 创建数据表字段对应的Java Bean。在本示例中trip_id为数据表中的唯一主键。

   > 注意 填写时请确保Java Bean的成员变量名称和数据表的字段名相同。

```
package mybatis;
public class Trip {
    private long trip_id;
    private long duration;
    private String start_date;
    private String end_date;
    private long start_station_number;
    private long end_station_number;
    private String start_station;
    private String end_station;
    private String bike_number;
    private String member_type;
    // ...
}
```

2. 创建映射配置文件，并在映射配置文件中定义查询条件。此处以在mybatis目录下创建TripMapper.xml为例介绍。

   关于SQL功能支持情况，请参见SQL支持功能说明。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mybatis.TripMapper">
    <select id="selectTrip" resultType="mybatis.Trip">
        select * from trips where trip_id = #{id}
    </select>
</mapper>
```

步骤四：构建SqlSessionFactory

SqlSessionFactory用于创建MyBatis会话，通过MyBatis会话，可以实现客户端与表格存储之间的连接。

1. 创建MyBatis配置文件mybatis-config.xml后添加如下内容。并根据实际修改对应配置项。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <!-- 设置数据源类型。由于表格存储JDBC驱动需要主动关闭后才能让进程退出，请根据实际使用选择合适的数据源类型。-->
            <!-- 如果程序常驻执行，则您可以使用POOLED维护一个连接池；如果希望程序完成数据查询后退出，则只能使用UNPOOLED。-->
            <dataSource type="UNPOOLED">
                <property name="driver" value="com.alicloud.openservices.tablestore.jdbc.OTSDriver"/>
                <property name="url" value="jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance"/>
                <property name="username" value="**********************"/>
                <property name="password" value="******************************"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <!-- 设置为映射配置文件的路径。-->
        <mapper resource="mybatis/TripMapper.xml"/>
    </mappers>
</configuration>
```

配置项说明请参见下表。

配置项	类型	是否必选	示例	描述
driver	class	是	com.alicloud.openservices.tablestore.jdbc.OTSDriver	表格存储JDBC驱动的类名，设置为com.alicloud.openservices.tablestore.jdbc.OTSDriver。

配置项	类型	是否必选	示例	描述
url	string	是	jdbc:ots:https://myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance	实例访问地址。格式为 `jdbc:ots:endpoint/instanceName`，其中endpoint为实例的服务地址。更多信息，请参见服务地址。instanceName为实例名称，请根据实际修改。 填写时必须加上前缀 `jdbc:ots:`。
username	string	是	**********	阿里云账号或者RAM用户的AccessKey ID。
password	string	是	**********	阿里云账号或者RAM用户的AccessKey Secret。

2. 通过加载MyBatis配置文件来构建SqlSessionFactory。

```
String resource = "mybatis/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
```

步骤五：创建SqlSession查询数据

```
SqlSession session = sqlSessionFactory.openSession(true);
Trip trip = (Trip) session.selectOne("mybatis.TripMapper.selectTrip", 99L);
System.out.println("trip id: " + trip.getTrip_id());
System.out.println("start date: " + trip.getStart_date());
System.out.println("end date: " + trip.getEnd_date());
System.out.println("duration: " + trip.getDuration());
session.close();
```

完整示例

查询表中主键列值为99的行数据，并获取指定列的值。

```
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import mybatis.Trip;
import java.io.IOException;
import java.io.InputStream;
public class MyBatisDemo {
    public static void main(String[] args) throws IOException {
        // 设置MyBatis配置文件的
        String resource = "mybatis/mybatis-config.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStr
eam);
        // 由于表格存储目前暂不支持事务，因此是否自动提交配置必须设置为true。
        SqlSession session = sqlSessionFactory.openSession(true);
        // 填写要执行的SELECT语句对应的标识并设置主键值为99。
        // SELECT语句对应的标识格式为"映射配置文件路径.SELECT语句id"，示例中mybatis.TripMapper.se
lectTrip表示执行mybatis节点下TripMapper.xml文件内id为selectTrip的SELECT语句。
        Trip trip = (Trip) session.selectOne("mybatis.TripMapper.selectTrip", 99L);
        // 打印需要获取的列值。
        System.out.println("trip id: " + trip.getTrip_id());
        System.out.println("start date: " + trip.getStart_date());
        System.out.println("end date: " + trip.getEnd_date());
        System.out.println("duration: " + trip.getDuration());
        session.close();
    }
}
```

9.7. 使用Go语言驱动

本文介绍如何使用Go语言驱动连接表格存储。

注意事项

目前支持使用SQL查询功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、德国（法兰克福）和新加坡。

前提条件

- 如果要使用RAM用户进行操作，请确保已创建RAM用户，并为RAM用户授予所有SQL操作权限，即在自定义权限策略中配置 `"Action": "ots:SQL*"` 。具体操作，请参见配置RAM用户权限。
- 已获取AccessKey（包括AccessKey ID和AccessKey Secret）。具体操作，请参见获取AccessKey。
- 已创建数据表并为数据表创建映射关系。具体操作，请分别参见创建数据表和创建表的映射关系。

步骤一：安装Go语言驱动

通过以下命令安装Go语言驱动。

```
go get github.com/aliyun/aliyun-tablestore-go-sql-driver
```

步骤二：使用Go语言驱动直连

表格存储的Go语言驱动是 `database/sql/driver` 接口的实现，导入包后即可使用 `database/sql` 访问表格存储。

- 参数说明

  使用Go语言驱动访问表格存储时，需要设置Go语言驱动名称和表格存储DSN（数据源名称）。具体参数说明请参见下表。

参数	示例	说明
driverName	ots	表格存储的Go语言驱动名称。固定取值为ots。
dataSourceName	https://**********:**********@myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance	表格存储数据源名称。格式为 `schema://accessKeyId:accessKeySecret@endpoint/instanceName[?param1=value1&...&paramN=valueN]` 。主要字段说明如下： ○ schema（必选）：表格存储驱动使用的协议，一般设置为https。 ○ accessKeyId:accessKeySecret（必选）：阿里云账号或者RAM用户的AccessKey ID和AccessKey Secret。 ○ endpoint（必选）：实例的服务地址。更多信息，请参见服务地址。 ○ instanceName（必选）：实例名称。 其他常用配置项的说明，请参见配置项。

- 示例

```
import (
    "database/sql"
    _ "github.com/aliyun/aliyun-tablestore-go-sql-driver"
)
// 设置Go语言驱动名称和表格存储数据源名称。
db, err := sql.Open("ots", "https://access_key_id:access_key_secret@endpoint/instance_name")
if err != nil {
    panic(err)    // 处理错误。
}
```

步骤三：查询数据

表格存储的Go语言驱动支持直接使用Query方法执行查询语句，也支持使用Prepare创建一个Stmt进行查询。

> **注意** 获取查询结果时，接收字段数据的变量类型必须和表格存储中字段数据类型相匹配。关于数据类型映射的更多信息，请参见数据类型映射。

- 使用Query方法查询

```
// 设置SQL语句，此处以查询test_table表中pk1列、col1列和col2列的数据为例介绍，请根据实际需要设置。
rows, err := db.Query("SELECT pk1, col1, col2 FROM test_table WHERE pk1 = ?", 3)
if err != nil {
    panic(err)    // 处理错误。
}
for rows.Next() {
    var pk1 int64
    var col1 float64
    var col2 string
    err := rows.Scan(&pk1, &col1, &col2)
    if err != nil {
        panic(err)    // 处理错误。
    }
}
```

- 使用Prepare创建一个Stmt查询

```
// 设置SQL语句，此处以查询test_table表中pk1列、col1列和col2列的数据为例介绍，请根据实际需要设置。
stmt, err := db.Prepare("SELECT pk1, col1, col2 FROM test_table WHERE pk1 = ?")
if err != nil {
    panic(err)    // 处理错误。
}
rows, err := stmt.Query(3)
if err != nil {
    panic(err)    // 处理错误。
}
for rows.Next() {
    var pk1 int64
    var col1 float64
    var col2 string
    err := rows.Scan(&pk1, &col1, &col2)
    if err != nil {
        panic(err)    // 处理错误。
    }
}
```

完整示例

查询华东1（杭州）地域下myinstance实例中test_table的所有数据。

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/aliyun/aliyun-tablestore-go-sql"
)
func main() {
    db, err := sql.Open("ots", "https://**********************:*************************
******@myinstance.cn-hangzhou.ots.aliyuncs.com/myinstance")
    if err != nil {
        panic(err)
    }
    rows, err := db.Query("SELECT * FROM test_table")
    if err != nil {
        panic(err)
    }
    for rows.Next() {
        // 获取所有列。
        columns, err := rows.Columns()
        if err != nil {
            panic(err)
        }
        // 创建存放数据的数组和指针。
        values := make([]interface{}, len(columns))
        pointers := make([]interface{}, len(columns))
        for i := range values {
            pointers[i] = &values[i]
        }
        // 扫描数据行。
        err = rows.Scan(pointers...)
        if err != nil {
            panic(err)
        }
        fmt.Println(values...)
    }
    rows.Close()
    db.Close()
}
```

配置项

通过表格存储Go语言驱动支持修改Go SDK的配置项。常用配置项的详细说明请参见下表。

配置项	示例值	说明
retryTimes	10	重试次数。默认值为10。
connectionTimeout	15	建立连接的超时时间。默认值为15。单位为秒。0表示无限等待。
requestTimeout	30	发送请求的超时时间。默认值为30。单位为秒。
maxRetryTime	5	最大触发重试时间。默认值为5。单位为秒。

配置项	示例值	说明
maxIdleConnections	2000	最大空闲连接数。默认值为2000。

数据类型映射

表格存储和Go语言中字段数据类型的对应关系请参见下表。如果字段数据类型不匹配，则会产生错误。

表格存储中字段数据类型	Go语言中字段数据类型
Integer	int64
Binary	[]byte
String	string
Double	float64
Boolean	bool

9.8. DDL操作

9.8.1. 创建表的映射关系

执行CREATE TABLE语句为已存在的表或者多元索引创建映射关系。本文介绍如何为已存在的表创建映射关系。

> **说明** 关于创建多元索引的映射关系的具体操作，请参见创建多元索引的映射关系。

语法

```
CREATE TABLE [IF NOT EXISTS] table_name(column_name data_type [NOT NULL | NULL],...
| PRIMARY KEY(key_part[,key_part])
)
ENGINE='tablestore',
ENGINE_ATTRIBUTE='{"consistency": consistency [,"allow_inaccurate_aggregation": allow_inacc
urate_aggregation]}';
```

当表只有一个主键时，您还可以使用如下语法为已存在的表创建映射关系。

```
CREATE TABLE [IF NOT EXISTS] table_name(
column_name data_type PRIMARY KEY,column_name data_type [NOT NULL | NULL],...
)
ENGINE='tablestore',
ENGINE_ATTRIBUTE='{"consistency": consistency [,"allow_inaccurate_aggregation": allow_inacc
urate_aggregation]}';
```

参数

参数	是否必选	说明
IF NOT EXISTS	否	如果指定IF NOT EXISTS，无论表是否存在，均会返回成功；如果未指定IF NOT EXISTS，则只有当表不存在时，才会返回成功。
table_name	是	表名，用于唯一标识一张表。 SQL中的表名必须和原始表名相同。
column_name	是	列名。 SQL中的列名必须和原始表中的列名等效，例如原始表中列名为Aa，在SQL中列名必须使用Aa、AA、aA或者aa中的一个。
data_type	是	列的数据类型，包含BIGINT、DOUBLE、BOOL等多种数据类型。 SQL中列的数据类型必须和原始表中列的数据类型相匹配。关于数据类型映射的更多信息，请参见数据类型映射。
NOT NULL | NULL	否	列值是否允许为NULL。取值范围如下： • NOT NULL：禁止该列的值为NULL，主键列默认不允许为NULL。 • NULL：允许该列的值为NULL，属性列默认允许为NULL。 当某属性列值不能为NULL时，请指定该列为NOT NULL。
key_part	是	主键列名称。支持设置多个主键列，多个主键列之间用英文逗号（,）分隔。 主键列名称必须包含在列名中。
ENGINE	否	使用映射表查询数据时的执行引擎。取值范围如下： • tablestore（默认）：SQL引擎将自动选择合适的索引执行查询。 • searchindex：SQL引擎将通过指定的多元索引执行查询。设置为此项时，您必须在ENGINE_ATTRIBUTE中配置index_name和table_name。

参数	是否必选	说明
ENGINE_ATTRIBUTE	否	执行引擎的属性，以JSON格式表示。包括如下选项： • index_name：多元索引映射表绑定的多元索引名称。只有创建多元索引的映射关系时才需要配置。 • table_name：多元索引映射表绑定的多元索引所属的数据表名称。只有创建多元索引的映射表时才需要配置。 • consistency：执行引擎支持的一致性模式。 创建表的映射关系时，取值范围如下： ◦ eventual（默认）：执行的查询结果满足最终一致。此时新数据写入后会在几秒后影响到查询结果。 ◦ strong：执行的查询结果满足强一致性。此时新数据写入后立刻影响到查询结果。 创建多元索引的映射关系时，consistency取值固定为eventual。 • allow_inaccurate_aggregation：是否允许通过牺牲聚合操作的精准度提升查询性能。类型为Boolean。 创建表的映射关系时，默认值为true，表示允许通过牺牲聚合操作的精度提升查询性能。您可以根据需要配置此项为false。 创建多元索引的映射关系时，allow_inaccurate_aggregation的取值固定为true。

示例

- 创建名称为exampletable1的表，表包含主键列id（BIGINT类型）以及属性列colvalue（BIGINT类型）和content（MEDIUMTEXT）。

```
CREATE TABLE exampletable1 (id BIGINT PRIMARY KEY, colvalue BIGINT, content MEDIUMTEXT);
```

- 创建名称为exampletable2的表，表包含主键列id（BIGINT类型）、colvalue（VARCHAR类型）以及属性列content（MEDIUMTEXT类型），并且所有对该表的查询必须满足强一致性。

```
CREATE TABLE exampletable2 (id BIGINT, colvalue VARCHAR(1024), content MEDIUMTEXT, PRIMAR
Y KEY(colvalue, id)) ENGINE_ATTRIBUTE='{"consistency": "strong"}';
```

9.8.2. 创建多元索引的映射关系

执行CREATE TABLE语句为已存在的表或者多元索引创建映射关系。本文介绍如何为已存在的多元索引创建映射关系。

> **说明** 关于创建表的映射关系的具体操作，请参见创建表的映射关系。

背景信息

由于执行CREATE TABLE创建表的映射关系时，数据表可能会包含多种不同的索引类型，当用户使用映射表查询数据时，SQL引擎会自动选择合适的数据表主键查询、二级索引或者多元索引执行查询。为了让用户更清晰地选择指定多元索引执行查询，SQL引擎支持通过CREATE TABLE语句创建指定多元索引的映射关系。

语法

```
CREATE TABLE [IF NOT EXISTS] user_defined_name(column_name data_type L,column_name data_typ
e])
ENGINE='searchindex',
ENGINE_ATTRIBUTE='{"index_name": index_name, "table_name": table_name}';
```

参数

参数	是否必选	说明
IF NOT EXISTS	否	如果指定IF NOT EXISTS，无论表是否存在，均会返回成功；如果未指定IF NOT EXISTS，则只有当表不存在时，才会返回成功。
user_defined_name	是	多元索引映射表名，用于唯一标识一张SQL绑定表。 后续执行SQL操作时要使用该表名。
column_name	是	列名。 SQL中的列名必须和原始表中的列名等效，例如原始表中列名为Aa，在SQL中列名必须使用Aa、AA、aA或者aa中的一个。
data_type	是	列的数据类型，包含BIGINT、DOUBLE、BOOL等多种数据类型。 SQL中列的数据类型必须和原始表中列的数据类型相匹配。关于数据类型映射的更多信息，请参见数据类型映射。
ENGINE	是	使用映射表查询数据时的执行引擎。取值范围如下： • tablestore（默认）：SQL引擎将自动选择合适的索引执行查询。 • searchindex：SQL引擎将通过指定的多元索引执行查询。设置为此项时，您必须在ENGINE_ATTRIBUTE中配置index_name和table_name。

参数	是否必选	说明
ENGINE_ATTRIBUTE	是	执行引擎的属性，以JSON格式表示。包括如下选项： • index_name：多元索引映射表绑定的多元索引名称。只有创建多元索引的映射关系时才需要配置。 • table_name：多元索引映射表绑定的多元索引所属的数据表名称。只有创建多元索引的映射表时才需要配置。 • consistency：执行引擎支持的一致性模式。 创建表的映射关系时，取值范围如下： ○ eventual（默认）：执行的查询结果满足最终一致。此时新数据写入后会在几秒后影响到查询结果。 ○ strong：执行的查询结果满足强一致性。此时新数据写入后立刻影响到查询结果。 创建多元索引的映射关系时，consistency取值固定为eventual。 • allow_inaccurate_aggregation：是否允许通过牺牲聚合操作的精准度提升查询性能。类型为Boolean。 创建表的映射关系时，默认值为true，表示允许通过牺牲聚合操作的精度提升查询性能。您可以根据需要配置此项为false。 创建多元索引的映射关系时，allow_inaccurate_aggregation的取值固定为true。

示例

为exampletable1数据表的多元索引exampletable1_index创建多元索引映射表search_exampletable1，多元索引映射表包含id（BIGINT类型）、colvalue（MEDIUMTEXT类型）和content（MEDIUMTEXT类型）。

```
CREATE TABLE search_exampletable1(id BIGINT, colvalue MEDIUMTEXT, content MEDIUMTEXT) ENGIN
E='searchindex' ENGINE_ATTRIBUTE='{"index_name": "exampletable1_index", "table_name": "exam
pletable1"}';
```

创建多元索引映射表search_exampletable1后，您可以执行以下操作：

- 查询search_exampletable1的描述信息。

  ```
  SHOW INDEX IN search_exampletable1;
  ```

  关于查询索引描述信息的更多信息，请参见查询索引描述信息。

- 查询search_exampletable1中content列值匹配"tablestore cool"字符串至少1个分词的行数据，最多返回10行，并且返回id和content列。

```
SELECT id,content FROM search_exampletable1 WHERE TEXT_MATCH(content, "tablestore cool")
LIMIT 10;
```

关于查询匹配字符串的数据的更多信息，请参见查询数据和全文检索。

9.8.3. 更新映射表属性列

执行ALTER TABLE语句为已存在的映射表添加或删除属性列。

> **说明** 关于创建映射表的具体操作，请参见创建表的映射关系。

注意事项

- 执行ALTER TABLE语句仅支持更新映射表的Schema，不会更新表格存储的数据存储Schema。
- 不支持添加或者删除映射表主键列的操作。
- 执行ALTER TABLE语句后，SQL引擎会异步进行刷新，最多30秒完成刷新。在此期间执行返回所有列等操作时可能不会返回新添加的列。
- 只有通过CREATE TABLE语句创建的映射表才支持使用ALTER TABLE语句更新属性列。通过DESCRIBE操作自动绑定的映射表不支持使用ALTER TABLE语句更新属性列。
- ALTER TABLE语句每次仅支持添加或者删除一列。如果需要进行多列操作，请执行多次ALTER TABLE语句。

语法

```
ALTER TABLE table_name option column_name [data_type];
```

参数

参数	是否必选	说明
table_name	是	映射表名，用于唯一标识一张映射表。
option	是	可执行的更新操作，取值范围如下： • ADD COLUMN：在映射表中添加一个属性列。 • DROP COLUMN：在映射表中删除一个属性列。
column_name	是	列名。新添加的列名不能与映射表中已有列名相同。 SQL中的列名必须和原始表中的列名等效，例如原始表中列名为Aa，在SQL中列名必须使用Aa、AA、aA或者aa中的一个。
data_type	否	列的数据类型，包含BIGINT、DOUBLE、BOOL等多种数据类型。只有当option为**ADD COLUMN**时才需要配置此参数。 SQL中列的数据类型必须和原始表中列的数据类型相匹配。关于数据类型映射的更多信息，请参见数据类型映射。

示例

- 为exampletable映射表添加colvalue（BIGINT类型）和content（MEDIUMTEXT）属性列，请依次执行以下两条SQL语句。

```
ALTER TABLE exampletable ADD COLUMN colvalue BIGINT;
```

```
ALTER TABLE exampletable ADD COLUMN content MEDIUMTEXT;
```

- 删除exampletable映射表中的colvalue（BIGINT类型）属性列。

```
ALTER TABLE exampletable DROP COLUMN colvalue;
```

9.8.4. 删除映射关系

当表的属性列发生变化时，您可以执行DROP MAPPING TABLE语句删除表的映射关系后重新创建。单次请求支持删除多个表的映射关系。

> **说明** 执行DROP MAPPING TABLE语句不会删除实际的表。

语法

```
DROP MAPPING TABLE [IF EXISTS] table_name,...;
```

参数

参数	是否必选	说明
table_name	是	表名，单次请求可配置多个表名，多个表名之间用英文逗号（,）分隔。
IF EXISTS	否	如果指定IF EXISTS，无论映射关系是否存在，均会返回成功；如果未指定IF EXISTS，则只有当映射关系存在时，才会返回成功。

示例

删除exampletable表的映射关系。

```
DROP MAPPING TABLE IF EXISTS exampletable;
```

9.8.5. 查询表的描述信息

执行DESCRIBE语句查询表的描述信息，例如字段名称、字段类型等。

语法

```
DESCRIBE table_name;
```

参数

参数	是否必选	说明
table_name	是	表名

示例

查询exampletable表的描述信息。

```
DESCRIBE exampletable;
```

返回结果如下图所示。

执行结果

Extra	Field	Key	Null	Type
	id	PRI	NO	bigint(20)
	value		YES	bigint(20)
	content		YES	mediumtext

9.9. DQL操作

9.9.1. 查询数据

执行SELECT语句查询表中数据。

注意事项

SELECT语句中子句的执行优先级为WHERE子句 > GROUP BY分组查询 > HAVING子句 > ORDER BY排序 > LIMIT和OFFSET。

语法

```
SELECT
    [ALL | DISTINCT | DISTINCTROW]
    select_expr [, select_expr] ...
    [FROM table_references]
    [WHERE where_condition]
    [GROUP BY groupby_condition]
    [HAVING having_condition]
    [ORDER BY order_condition]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

参数

参数	是否必选	说明

参数	是否必选	说明
ALL | DISTINCT | DISTINCTROW	否	是否去掉重复的字段，取值范围如下： • ALL（默认）：返回字段中所有重复的值。 • DISTINCT：去掉重复的字段，返回去重字段后的值。 • DISTINCTROW：去掉重复的行，返回去重行后的值。
select_expr	是	列名或者列表达式，格式为 `column_name[, column_name][, column_exp],...` 。更多信息，请参见列表达式（select_expr）。
table_references	是	目标表信息，可以是表名或者SELECT语句，格式为 `table_name | select_statement` 。更多信息，请参见目标表信息（table_references）。
where_condition	否	WHERE子句，可配合不同条件实现相应功能。 • 配合关系运算符查询符合指定条件的数据，格式为 `column_name operator value [AND | OR] [column_name operator value]` 。更多信息，请参见WHERE子句（where_condition）。 • 配合匹配查询或者短语匹配查询条件实现全文检索。更多信息，请参见全文检索。
groupby_condition	否	GROUP BY分组查询，可配合聚合函数使用，格式为 `column_name` 。更多信息，请参见GROUP BY分组查询（groupby_condition）。
having_condition	否	HAVING子句，可配合聚合函数使用，格式为 `aggregate_function(column_name) operator value` 。更多信息，请参见HAVING子句（having_condition）。
order_condition	否	ORDER BY排序，格式为 `column_name [ASC | DESC][,column_name [ASC | DESC],...]` 。更多信息，请参见ORDER BY排序（order_condition）。
row_count	否	本次查询需要返回的最大行数。
offset	否	本次查询的数据偏移量，默认偏移量为0。

列表达式（select_expr）

通过列表达式指定需要查询的列。使用规则如下：

- 使用星号（*）查询所有列，支持配合WHERE子句指定查询条件。

```
SELECT * FROM orders;
```

使用WHERE子句作为查询条件的示例如下：

```
SELECT * FROM orders WHERE orderprice >= 100;
```

- 使用列名指定查询的列。

```
SELECT username FROM orders;
```

目标表信息（table_references）

通过目标表信息指定需要查询的表。

```
SELECT orderprice FROM orders;
```

WHERE子句（where_condition）

通过WHERE子句查询满足指定条件的数据。使用规则如下：

- 配合算术运算符、关系运算符等构造的简单表达式使用。

```
SELECT * FROM orders WHERE username = 'lily';
SELECT * FROM orders WHERE orderprice >= 100;
```

- 配合逻辑运算符构造的组合表达式使用。

```
SELECT * FROM orders WHERE username = 'lily' AND orderprice >= 100;
```

关于操作符的更多信息，请参见SQL操作符。

GROUP BY分组查询（groupby_condition）

通过GROUP BY子句对SELECT语句的结果集按照指定条件进行分组。使用规则如下：

- 按照字段分组

```
SELECT username FROM orders GROUP BY username;
```

- 在分组的列上支持使用聚合函数

```
SELECT username,COUNT(*) FROM orders GROUP BY username;
```

- SELECT的所有列中没有使用聚合函数的列，必须出现在GROUP BY中。

```
SELECT username,orderprice FROM orders GROUP BY username,orderprice;
```

关于聚合函数的更多信息，请参见聚合函数。

HAVING子句（having_condition）

通过HAVING子句对WHERE子句和GROUP BY分组查询后的分组结果集进行过滤，查询满足条件的分组结果。

通常HAVING子句与聚合函数配合使用，实现过滤。

```
SELECT username,SUM(orderprice) FROM orders GROUP BY username HAVING SUM(orderprice) < 500;
```

ORDER BY排序（order_condition）

通过ORDER BY子句按照指定字段和排序方式对查询结果集进行排序。使用规则如下：

- 支持使用ASC或者DESC关键字设置排序方式。 默认按照升序（ASC）排列。

```
SELECT * FROM orders ORDER BY orderprice DESC LIMIT 10;
```

- 支持设置多个字段进行排序。

```
SELECT * FROM orders ORDER BY username ASC,orderprice DESC LIMIT 10;
```

- 通常与LIMIT配合使用限定返回的行数。

```
SELECT * FROM orders ORDER BY orderprice LIMIT 10;
```

9.9.2. 聚合函数

通过本文您可以了解使用SQL查询时支持的聚合函数。

函数	说明
COUNT()	返回匹配指定条件的行数。
COUNT(DISTINCT)	返回指定列不同值的行数。
SUM()	返回数值列的总和。
AVG()	返回数值列的平均值。
MAX()	返回一列中的最大值。
MIN()	返回一列中的最小值。

9.9.3. 全文检索

使用匹配查询（TEXT_MATCH）或者短语匹配查询（TEXT_MATCH_PHRASE）条件作为SELECT语句中的WHERE子句，您可以查询表中匹配指定字符串的数据，实现全文检索功能。

前提条件

已创建多元索引并为要匹配的列设置了分词。具体操作，请参见创建多元索引。

> **说明** 关于分词的更多信息，请参见分词。

匹配查询

采用近似匹配的方式查询表中的数据。对Text类型的列值和查询关键词会先按照设置好的分词器做切分，然后按照切分好后的词去查询。对于进行模糊分词的列，建议使用TEXT_MATCH_PHRASE实现高性能的模糊查询。

- SQL表达式

```
TEXT_MATCH(fieldName, text, [options])
```

- 参数说明

参数	类型	是否必选	示例值	说明
fieldName	string	是	col1	要匹配的列。匹配查询可应用于Text类型。

参数	类型	是否必选	示例值	说明
text	string	是	"tablestore is cool"	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如当要匹配的列为Text类型时，分词类型为单字分词，则查询词为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
options	string	否	"or", "2"	匹配选项。包括如下选项： ○ operator：关系逻辑符，可选值包括OR和AND，默认值为OR。 ○ minimum_should_match：最小匹配个数，默认值为1。 如果operator为OR，只有当某一行数据的fieldName列的值中至少包括最小匹配个数的词时，才表示行数据满足查询条件。 如果operator为AND，则只有分词后的所有词都在列值中时，才表示行数据满足查询条件。

- 返回值

  返回值为Boolean类型，表示该行是否满足查询条件。取值为true时，表示满足查询条件；取值为false时，表示不满足查询条件。

- 示例

  查询exampletable表中col1列值匹配"tablestore is cool"字符串至少2个分词的数据。

```
SELECT * FROM exampletable WHERE TEXT_MATCH(col1, "tablestore is cool", "or", "2")
```

  查询exampletable表中col1列值匹配"tablestore is cool"字符串所有分词的数据。

```
SELECT * FROM exampletable WHERE TEXT_MATCH(col1, "tablestore is cool", "and")
```

短语匹配查询

类似于TEXT_MATCH，但是分词后多个词的位置关系会被考虑，只有分词后的多个词在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。

- SQL表达式

```
TEXT_MATCH_PHRASE(fieldName, text)
```

- 参数说明

参数	类型	是否必选	示例值	说明
fieldName	string	是	col1	要匹配的列。匹配查询可应用于Text类型。
text	string	是	"tablestore is cool"	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如查询的值为“this is”，可以匹配到“...，this is tablestore”、“this is a table”，但是无法匹配到“this table is ...”以及“is this a table”。

- 返回值

  返回值为Boolean类型，表示该行是否满足查询条件。取值为true时，表示满足查询条件；取值为false时，表示不满足查询条件。

- 示例

  查询exampletable表中col1列值匹配"tablestore is cool"字符串的数据。

```
SELECT * FROM exampletable WHERE TEXT_MATCH_PHRASE(col1, "tablestore is cool")
```

9.10. Database Administration操作

9.10.1. 查询索引描述信息

执行SHOW INDEX语句查询表的索引描述信息。

语法

```
SHOW INDEX {FROM | IN} table_name;
```

参数

参数	是否必选	说明
table_name	是	表名

示例

查询exampletable表的索引描述信息。

```
SHOW INDEX IN exampletable;
```

9.10.2. 列出表名称列表

执行SHOW TABLES语句列出当前数据库中的表名称列表。

语法

```
SHOW TABLES;
```

示例

列出当前数据库中的表名称列表。

```
SHOW TABLES;
```

返回结果如下：

```
Tables_in_tpch
exampletable1
exampletable2
```

9.11. 查询优化

9.11.1. 索引选择策略

表格存储作为海量结构化大数据存储，支持不同的索引结构，便于不同场景的查询分析加速使用。使用SQL查询功能时，您可以通过显式访问二级索引表进行索引查询。对于多元索引，表格存储提供了自动多元索引选择策略和显式访问多元索引两种方式。

> **说明** 关于二级索引和多元索引的更多信息，请分别参见二级索引简介和多元索引简介。

使用二级索引表

> **注意** 当前表格存储不支持自动选择二级索引进行数据查询。

表格存储只支持显式访问二级索引表。具体操作如下：

1. 通过CREATE TABLE语句创建二级索引表的映射关系。具体操作，请参见创建表的映射关系。
2. 执行SELECT语句查询数据。具体操作，请参见查询数据。

使用多元索引

当使用SQL进行非主键列查询、多列自由组合查询等复杂查询需求时，推荐您为数据表创建多元索引。创建多元索引后，要通过SQL使用多元索引查询数据时，您可以通过如下任意一种方式进行操作。

- 显式访问多元索引

当要通过指定的多元索引查询数据时，您可以使用显式访问多元索引的方式。具体操作如下：

i. 通过CREATE TABLE语句创建多元索引的映射关系。具体操作，请参见创建多元索引的映射关系。

ii. 执行SELECT语句查询数据。具体操作，请参见查询数据。

- 自动选择多元索引

> **注意** 如果创建表的映射关系时设置了执行的查询结果要满足强一致性或者不允许通过牺牲聚合操作的精度提升查询性能，则表格存储不会自动选择多元索引进行数据查询。

当未显式指定要访问的多元索引时，如果WHERE子句中所有的过滤列以及SELECT语句中的返回列均在一个多元索引中，表格存储会自动选择该多元索引进行数据查询。例如 `Select A,B,C from exampletable where A=XXX and D = YY;` 语句，如果A、B、C、D列均在exampletable表的同一个多元索引中，则表格存储会自动选择多元索引进行数据查询。

此外，当Groupby、聚合函数等组合使用时，如果符合多元索引Search接口的统计聚合能力，则表格存储也会进行识别并下推算子，关于下推算子的更多信息，请参见计算下推。

多元索引中Search查询与SQL表达式的映射

SQL表达式	示例	多元素引中Search查询
without predicate	不涉及	全匹配查询（MatchAllQuery）
=	• a = 1 • b = "hello world"	精确查询（TermQuery）
>	a > 1	范围查询（RangeQuery）
>=	a >= 2	
<	a < 5	
<=	a <= 10	
is null	a is null	列存在性查询（ExistsQuery）
is not null	a is not null	
and	a = 1 and b = "hello world"	多条件组合查询（BoolQuery）
or	a > 1 or b = 2	
not	not a = 1	
!=	a !=1	
like	a like "%s%"	通配符查询（WildcardQuery）
in	a in (1,2,3)	多词精确查询（TermsQuery）
text_match	text_match(a, "tablestore cool")	匹配查询（MatchQuery）

SQL表达式	示例	多元素引中Search查询
text_match_phrase	text_match_phrase(a, "tablestore cool")	短语匹配查询（MatchPhraseQuery）

9.11.2. 计算下推

多元索引提供了条件过滤、聚合、排序等功能，在创建多元索引后，使用SQL查询时，系统能够充分利用多元索引的计算能力，将部分SQL计算任务下推到多元索引执行，避免全表扫描，提高计算效率。

前提条件

已创建多元索引。具体操作，请参见创建多元索引。

使用场景

如果多元索引包含SQL中涉及的数据列，则SQL引擎会通过多元索引读取数据并下推多元索引支持的算子。例如有一张表exampletable有a, b, c, d四列，多元索引中包含了b, c, d三列并均建立了索引，只有当SQL语句中只涉及b, c, d中的数据列时，才会多元索引读取数据。

```
SELECT a, b, c, d FROM exampletable; /* 多元索引不包含a,b,c,d，扫描全表读取数据，不支持算子下推 */
SELECT b, c, d FROM exampletable;    /* 多元索引包含b,c,d，通过多元索引读取数据，支持算子下推 */
```

支持下推的算子

算子类型	下推算子	下推限制
逻辑运算符	AND、OR	不支持NOT算子下推。

关系运算符	=、!=、<、<=、>、>=、BETWEEN ... AND ...	只有数据列和常数的比较才支持算子下推，不支持数据列和数据列比较的算子下推。
		`SELECT * FROM exampletable WHERE a > 1; /* 数据列和常数比较，支持算子下推 */`
		`SELECT * FROM exampletable WHERE a > b; /* 数据列和数据列比较，不支持算子下推 */`

算子类型	下推算子	下推限制

聚合函数	• 基础聚合：MIN、MAX、COUNT、AVG、SUM、ANY_VALUE
	• 去重聚合：COUNT(DISTINCT col_name)
	• 分组函数：GROUP BY col_name
		聚合函数可以对全部数据或者GROUP BY分组中的数据进行聚合，只有聚合函数支持下推并且函数参数为数据列时才支持算子下推。
		`SELECT COUNT(*) FROM exampletable;`
		`/* 特殊用法COUNT(*)，支持算子下推 */`
		`SELECT SUM(a) FROM exampletable;`
		`/* 参数为数据列，支持算子下推 */`
		`SELECT a, b FROM exampletable GROUP BY a, b; /* 按照数据列分组，支持算子下推 */`
		`SELECT a FROM exampletable GROUP BY a+1;`
		`/* 按照表达式分组，不支持算子下推 */`
		`SELECT SUM(a+b) FROM exampletable;`
		`/* 参数为表达式，不支持算子下推 */`

LIMIT	• LIMIT row_count
	• ORDER BY col_name LIMIT row_count
		ORDER BY的参数为数据列时才支持算子下推。
		`SELECT * FROM exampletable ORDER BY a LIMIT 1; /* 按照数据列排序，支持算子下推 */`
		`SELECT * FROM exampletable ORDER BY a, b LIMIT 1; /* 按照数据列排序，支持算子下推 */`
		`SELECT * FROM exampletable ORDER BY a+1 LIMIT 1; /* 按照表达式排序，不支持算子下推 */`

9.12. 附录

9.12.1. SQL操作符

通过本文您可以了解表格存储SQL中支持使用的操作符，SQL操作符包括算术运算符、关系运算符、逻辑运算符和位运算符。

算术运算符

算术运算符可用于SELECT或者WHERE子句中进行数值计算。

运算符	名称	说明
A+B	加法	返回A+B的结果。
A-B	减法	返回A-B的结果。
A*B	乘法	返回A*B的结果。
A/B或者A DIV B	除法	返回A÷B的结果。
A%B或者A MOD B	取余	返回A÷B后取余数的结果。

关系运算符

关系运算符用于判断表中符合指定条件的行数据。

- 如果比较结果为真（TRUE），则返回1。
- 如果比较结果为假（FALSE），则返回0。

关系运算符可应用于WHERE子句中作为限定条件。返回1表示满足条件，返回0表示不满足条件。

运算符	名称	说明
A:=B	赋值	将B的值赋给A。
A=B	等于	当A等于B时返回1，否则返回0。
A!=B或者A<>B	不等于	当A不等于B时返回1，否则返回0。
A>B	大于	当A大于B时返回1，否则返回0。
A<B	小于	当A小于B时返回1，否则返回0。
A>=B	大于等于	当A大于等于B时返回1，否则返回0。
A<=B	小于等于	当A小于等于B时返回1，否则返回0。
BETWEEN A AND B	值在区间内	当值大于等于A且小于等于B时返回1，否则返回0。
Not BETWEEN A AND B	值不在区间内	当值大于B或者小于A时返回1，否则返回0。
A LIKE B	模式匹配	字符串匹配操作。A为字符串，B为匹配模式，当A和B匹配时返回1，否返回0。 下划线（_）表示匹配一个字符，百分号（%）表示匹配任意个字符。
A NOT LIKE B	不符合模式匹配	字符串不匹配操作。A为字符串，B为匹配模式，当A和B不匹配时返回1，否则返回0。 下划线（_）表示匹配一个字符，百分号（%）表示匹配任意个字符。

逻辑运算符

逻辑运算符用来判断表达式的真假。

- 如果表达式为真（TRUE），则返回1。
- 如果表达式为假（FALSE），则返回0。

逻辑运算符可应用于WHERE子句中组成复杂的限定条件。返回1表示满足条件，返回0表示不满足条件。

运算符	名称	说明
A AND B或者A&&B	逻辑与	当A和B均为TRUE时返回1，否则返回0。
A OR B	逻辑或	当A和B中至少有一个为TRUE时返回1，否则返回0。

运算符	名称	说明
A XOR B	逻辑异或	当A和B不同时返回1，否则返回0。
NOT A 或 ！A	逻辑非	当A为FALSE时返回1，否则返回0。

位运算符

位运算符用于对二进制数进行计算。位运算将操作数转为二进制数后再进行位运算，最后再将计算结果从二进制数转为十进制数。

运算符	名称	说明
A&B	按位与	返回A和B按位与运算的结果。
A|B	按位或	返回A和B按位或运算的结果。
A^B	按位异或	返回A和B按位异或运算的结果。
~A	按位非	返回A按位取反的结果。

9.12.2. 保留字与关键字

通过本文您可以了解表格存储SQL中的所有保留字与关键字。

字母序	关键字和保留字
A	ACCESSIBLE ACCOUNT ACTION ADD AFTER AGAINST AGGREGATE ALGORITHM ALL ALTER ALWAYS ANALYSE ANALYZE ANDANY AS ASC ASCII ASENSITIVE AT AUTOEXTEND_SIZE AUTO_INCREMENT AVG AVG_ROW_LENGTH
B	BACKUP BEFORE BEGIN BETWEEN BIGINT BINARY BINLOG BIT BLOB BLOCK BOOL BOOLEAN BOTH BTREE BY BYTE
C	CACHE CALL CASCADE CASCADED CASE CATALOG_NAME CHAIN CHANGE CHANGED CHANNEL CHAR CHARACTER CHARSET CHECK CHECKSUM CIPHER CLASS_ORIGIN CLIENT CLOSE COALESCE CODE COLLATE COLLATION COLUMN COLUMNS COLUMN_FORMAT COLUMN_NAME COMMENT COMMIT COMMITTED COMPACT COMPLETION COMPRESSED COMPRESSION CONCURRENT CONDITION CONNECTION CONSISTENT CONSTRAINT CONSTRAINT_CATALOG CONSTRAINT_NAME CONSTRAINT_SCHEMA CONTAINS CONTEXT CONTINUE CONVERT CPU CREATE CROSS CUBE CURRENT CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP CURRENT_USER CURSOR CURSOR_NAME
D	DATA DATABASE DATABASES DATAFILE DATE DATETIME DAY DAY_HOUR DAY_MICROSECOND DAY_MINUTE DAY_SECOND DEALLOCATE DEC DECIMAL DECLARE DEFAULT DEFAULT_AUTH DEFINER DELAYED DELAY_KEY_WRITE DELETE DESC DESCRIBE DES_KEY_FILE DETERMINISTIC DIAGNOSTICS DIRECTORY DISABLE DISCARD DISK DISTINCT DISTINCTROW DIV DO DOUBLE DROP DUAL DUMPFILE DUPLICATE DYNAMIC
E	EACH ELSE ELSEIF ENABLE ENCLOSED ENCRYPTION END ENDS ENGINE ENGINES ENUM ERROR ERRORS ESCAPE ESCAPED EVENT EVENTS EVERY EXCHANGE EXECUTE EXISTS EXIT EXPANSION EXPIRE EXPLAIN EXPORT EXTENDED EXTENT_SIZE

字母序	关键字和保留字
F	FALSE FAST FAULTS FETCH FIELDS FILE FILE_BLOCK_SIZE FILTER FIRST FIXED FLOAT FLOAT4 FLOAT8 FLUSH FOLLOWS FOR FORCE FOREIGN FORMAT FOUND FROM FULL FULLTEXT FUNCTION
G	GENERAL GENERATED GEOMETRY GEOMETRYCOLLECTION GET GET_FORMAT GLOBAL GRANT GRANTS GROUP GROUP_REPLICATION
H	HANDLER HASH HAVING HELP HIGH_PRIORITY HOST HOSTS HOUR HOUR_MICROSECOND HOUR_MINUTE HOUR_SECOND
I	IDENTIFIED IF IGNORE IGNORE_SERVER_IDS IMPORT IN INDEX INDEXES INFILE INITIAL_SIZE INNER INOUT INSENSITIVE INSERT INSERT_METHOD INSTALL INSTANCE INT INT1 INT2 INT3 INT4 INT8 INTEGER INTERVAL INTO INVOKER IO IO_AFTER_GTIDS IO_BEFORE_GTIDS IO_THREAD IPC IS ISOLATION ISSUER ITERATE
J	JOIN JSON
K	KEY KEYS KEY_BLOCK_SIZE KILL
L	LANGUAGE LAST LEADING LEAVE LEAVES LEFT LESS LEVEL LIKE LIMIT LINEAR LINES LINESTRING LIST LOAD LOCAL LOCALTIME LOCALTIMESTAMP LOCK LOCKS LOGFILE LOGS LONG LONGBLOB LONGTEXT LOOP LOW_PRIORITY
M	MASTER MASTER_AUTO_POSITION MASTER_BIND MASTER_CONNECT_RETRY MASTER_DELAY MASTER_HEARTBEAT_PERIOD MASTER_HOST MASTER_LOG_FILE MASTER_LOG_POS MASTER_PASSWORD MASTER_PORT MASTER_RETRY_COUNT MASTER_SERVER_ID MASTER_SSL MASTER_SSL_CA MASTER_SSL_CAPATH MASTER_SSL_CERT MASTER_SSL_CIPHER MASTER_SSL_CRL MASTER_SSL_CRLPATH MASTER_SSL_KEY MASTER_SSL_VERIFY_SERVER_CERT MASTER_TLS_VERSION MASTER_USER MATCH MAXVALUE MAX_CONNECTIONS_PER_HOUR MAX_QUERIES_PER_HOUR MAX_ROWS MAX_SIZE MAX_UPDATES_PER_HOURMAX_USER_CONNECTIONS MEDIUM MEDIUMBLOB MEDIUMINT MEDIUMTEXT MEMORY MERGE MESSAGE_TEXT MICROSECOND MIDDLEINT MIGRATE MINUTE MINUTE_MICROSECOND MINUTE_SECOND MIN_ROWS MOD MODE MODIFIES MODIFY MONTH MULTILINESTRING MULTIPOINT MULTIPOLYGON MUTEX MYSQL_ERRNO
N	NAME NAMES NATIONAL NATURAL NCHAR NDB NDBCLUSTER NEVER NEW NEXT NO NODEGROUP NONBLOCKING NONE NOT NO_WAIT NO_WRITE_TO_BINLOG NULL NUMBER NUMERIC NVARCHAR
O	OFFSET OLD_PASSWORD ON ONE ONLY OPEN OPTIMIZE OPTIMIZER_COSTS OPTION OPTIONALLY OPTIONS OR ORDER OUT OUTER OUTFILE OWNER
P	PACK_KEYS PAGE PARSER PARSE_GCOL_EXPR PARTIAL PARTITION PARTITIONING PARTITIONS PASSWORD PHASE PLUGIN PLUGINS PLUGIN_DIR POINT POLYGON PORT PRECEDES PRECISION PREPARE PRESERVE PREV PRIMARY PRIVILEGES PROCEDURE PROCESSLIST PROFILE PROFILES PROXY PURGE
Q	QUARTER QUERY QUICK

字母序	关键字和保留字
R	RANGE READ READS READ_ONLY READ_WRITE REAL REBUILD RECOVER REDOFILE REDO_BUFFER_SIZE REDUNDANT REFERENCES REGEXP RELAY RELAYLOG RELAY_LOG_FILE RELAY_LOG_POS RELAY_THREAD RELEASE RELOAD REMOVE RENAME REORGANIZE REPAIR REPEAT REPEATABLE REPLACE REPLICATE_DO_DB REPLICATE_DO_TABLE REPLICATE_IGNORE_DB REPLICATE_IGNORE_TABLE REPLICATE_REWRITE_DB REPLICATE_WILD_DO_TABLE REPLICATE_WILD_IGNORE_TABLE REPLICATION REQUIRE RESET RESIGNAL RESTORE RESTRICT RESUME RETURN RETURNED_SQLSTATE RETURNS REVERSE REVOKE RIGHT RLIKE ROLLBACK ROLLUP ROTATE ROUTINE ROW ROWS ROW_COUNT ROW_FORMAT RTREE
S	SAVEPOINT SCHEDULE SCHEMA SCHEMAS SCHEMA_NAME SECOND SECOND_MICROSECOND SECURITY SELECT SENSITIVE SEPARATOR SERIAL SERIALIZABLE SERVER SESSION SET SHARE SHOW SHUTDOWN SIGNAL SIGNED SIMPLE SLAVE SLOW SMALLINT SNAPSHOT SOCKET SOME SONAME SOUNDSSOURCE SPATIAL SPECIFIC SQL SQLEXCEPTION SQLSTATE SQLWARNING SQL_AFTER_GTIDS SQL_AFTER_MTS_GAPS SQL_BEFORE_GTIDS SQL_BIG_RESULT SQL_BUFFER_RESULT SQL_CACHE SQL_CALC_FOUND_ROWS SQL_NO_CACHE SQL_SMALL_RESULT SQL_THREAD SQL_TSI_DAY SQL_TSI_HOUR SQL_TSI_MINUTE SQL_TSI_MONTH SQL_TSI_QUARTER SQL_TSI_SECOND SQL_TSI_WEEK SQL_TSI_YEAR SSL STACKED START STARTING STARTS STATS_AUTO_RECALC STATS_PERSISTENT STATS_SAMPLE_PAGES STATUS STOP STORAGE STORED STRAIGHT_JOIN STRING SUBCLASS_ORIGIN SUBJECT SUBPARTITION SUBPARTITIONS SUPER SUSPEND SWAPS SWITCHES
T	TABLE TABLES TABLESPACE TABLE_CHECKSUM TABLE_NAME TEMPORARY TEMPTABLE TERMINATED TEXT THAN THEN TIME TIMESTAMP TIMESTAMPADD TIMESTAMPDIFF TINYBLOB TINYINT TINYTEXT TO TRAILING TRANSACTION TRIGGER TRIGGERS TRUE TRUNCATE TYPE TYPES
U	UNCOMMITTED UNDEFINED UNDO UNDOFILE UNDO_BUFFER_SIZE UNICODE UNINSTALL UNION UNIQUE UNKNOWN UNLOCK UNSIGNED UNTIL UPDATE UPGRADE USAGE USE USER USER_RESOURCES USE_FRM USING UTC_DATE UTC_TIME UTC_TIMESTAMP
V	VALIDATION VALUE VALUES VARBINARY VARCHAR VARCHARACTER VARIABLES VARYING VIEW VIRTUAL
W	WAIT WARNINGS WEEK WEIGHT_STRING WHEN WHERE WHILE WITH WITHOUT WORK WRAPPER WRITE
X	X509 XA XID XML XOR
Y	YEAR YEAR_MONTH
Z	ZEROFILL

10.通道服务

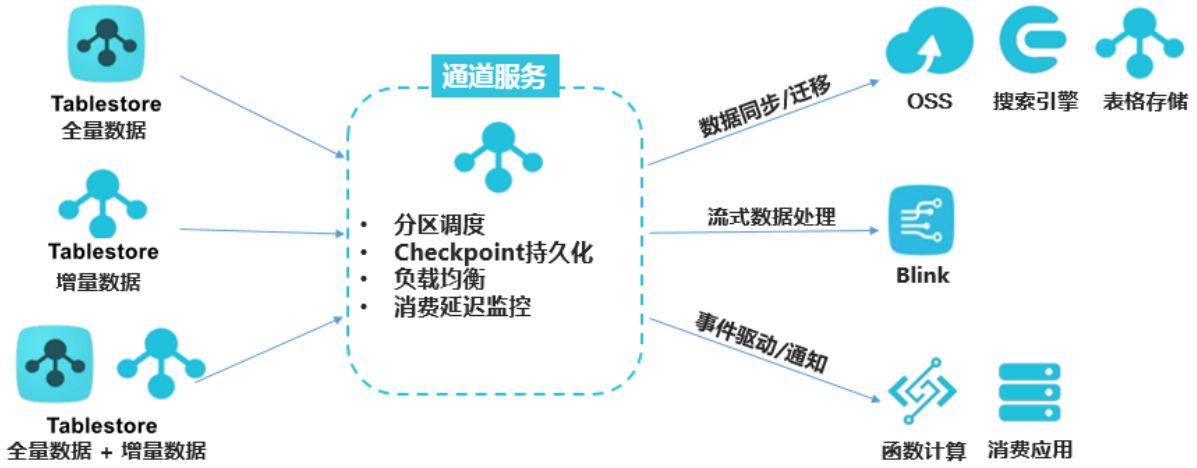
10.1. 概述

通道服务（Tunnel Service）是基于表格存储数据接口上的全增量一体化服务。通道服务提供了增量、全量、增量加全量三种类型的分布式数据实时消费通道。通过为数据表建立数据通道，您可以简单地实现对表中历史存量和新增数据的消费处理。

表格存储适合元数据管理、时序数据监控、消息系统等服务应用，这些应用通常利用增量数据流或者先全量后增量的数据流来触发一些附加的操作逻辑，这些附加操作包括。

- 数据同步：将数据同步到缓存、搜索引擎或者数据仓库中。
- 事件驱动：触发函数计算、通知消费端消费或者调用一些API。
- 流式数据处理：对接流式或者流批一体计算引擎。
- 数据搬迁：数据备份到OSS、迁移到容量型的表格存储实例等。

您可以利用通道服务针对这些模式轻松构建高效、弹性的解决方案，如下图。



10.2. 功能

通道服务提供了全增量一体的数据通道、增量数据变化保序、消费延迟监控和数据消费能力水平扩展功能。

> **说明** 通道服务支持表写入在10万每秒的量级下，从数据被更新到获取数据更新记录，延迟在毫秒级，并能按照更新顺序返回。

全增量一体的数据通道

通道服务不仅提供增量数据消费能力，还提供了可并行的全量数据消费以及全量加增量数据消费功能。

增量数据变化保序

通道服务为数据划分一到多个可并行消费的逻辑分区，每个逻辑分区的增量数据按写入时间顺序保序，不同逻辑分区的数据可以并行消费。

消费延迟监控

通道服务通过DescribeTunnel API提供了客户端消费数据延迟（即当前消费到的数据的时间点）信息，并在控制台提供了通道数据消费监控。

数据消费能力水平扩展

通道服务提供了逻辑分区的自动负载均衡功能，负载均衡增加了消费端数量，从而提高水平扩展数据消费速度。

10.3. 数据消费框架原理

Tunnel Client为通道服务的自动化数据消费框架。使用通道服务前，需要了解Tunnel Client的自动化数据处理流程、自动化的负载均衡和良好的水平扩展性以及自动化的资源清理和容错处理。

背景信息

Tunnel Client可以解决全量和增量数据处理时的常见问题，例如负载均衡、故障恢复、Checkpoint、分区信息同步确保分区信息消费顺序等。使用Tunnel Client后，您只需要关心每条记录的处理逻辑。

Tunnel Client的代码详情请参见Github。

自动化数据处理流程

Tunnel Client通过每一轮的定时心跳探测（Heartbeat）进行活跃Channel的探测，Channel和ChannelConnect状态的更新，和数据处理任务的初始化、运行和结束等。

1. Tunnel Client资源的初始化
   i. 将Tunnel Client状态由Ready置为Started。
   ii. 根据TunnelWorkerConfig中的HeartbeatTimeout和ClientTag（客户端标识）等配置进行ConnectTunnel操作，并和Tunnel服务端进行联通，以获取当前Tunnel Client对应的ClientId。
   iii. 初始化ChannelDialer，用于新建ChannelConnect。

   每一个ChannelConnect都会和一个Channel一一对应，ChannelConnect上会记录有数据消费的位点。
   iv. 根据用户传入的处理数据的Callback和TunnelWorkerConfig中CheckpointInterval（向服务端记数据位点的间隔）包装出一个带自动记Checkpoint功能的数据处理器。
   v. 初始化TunnelStateMachine，用于进行Channel状态机的自动化处理。
2. 固定间隔进行Heartbeat

   心跳的间隔由TunnelWorkerConfig中的heartbeatIntervalInSec参数决定。
   i. 进行Heartbeat请求，从Tunnel服务端获取最新可用的Channel列表，Channel中会包含有ChannelId、Channel的版本和Channel的状态信息。

ii. 将服务端获取到的Channel列表和本地内存中的Channel列表进行Merge，然后进行ChannelConnect的新建和update，规则如下：

- Merge：基于本轮从服务端获取的最新Channel列表，对于相同ChannelId，认定版本号更大的为最新状态，直接进行覆盖，对于未出现的Channel，则直接插入。
- 新建ChannelConnect：如果此Channel未新建其对应的ChannelConnect，则会新建一个WAIT状态的ChannelConnect，如果对应的Channel状态为OPEN状态，则同时会启动该ChannelConnect上处理数据的循环流水线任务（ReadRecords&&ProcessRecords），处理详情请参见代码中的ProcessDataPipeline类。
- Update已有ChannelConnect：Merge完成后，如果Channel对应的ChannelConnect存在，则根据相同ChannelId的Channel状态来更新ChannelConnect的状态，例如Channel为CLOSE状态也需要将ChannelConnect的状态置为Closed，进而终止处理任务的流水线任务，详情请参见代码中的ChannelConnect.notifyStatus方法。

3. Channel状态自动机说明

在心跳模式下，Tunnel服务端会根据保持心跳的Tunnel Client数量，调度可以消费的分区到不同Tunnel Client上，以达到负载均衡的目的。

Tunnel服务端通过Channel状态机来驱动每个Channel的消费以及进行负载均衡，如下图所示。

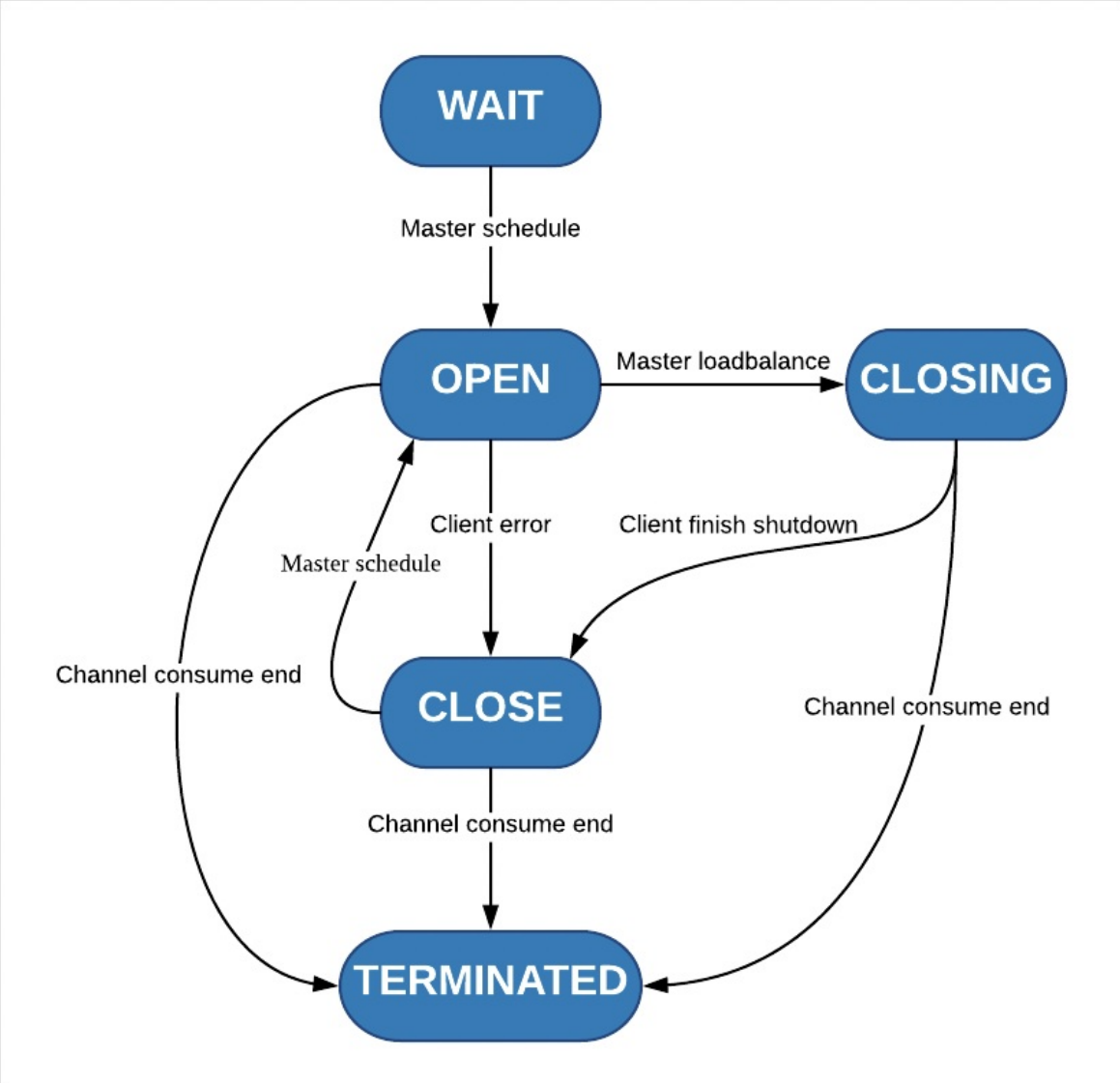
Tunnel服务端和Tunnel Client通过一个心跳和Channel版本号更新机制进行状态变换通信。

i. 每个Channel最初均处于WAIT状态。

ii. 增量类型Channel需要等待父分区上Channel消费完毕转为TERMINATED后才可以转为可消费状态OPEN。

iii. OPEN状态的分区会调度到各个Tunnel Client上。

iv. 在需要负载均衡时，Tunnel服务端和Tunnel Client有一个Channel状态OPEN->CLOSING->CLOSED的调度协议，Tunnel Client在消费完一个全量Channel split或者发生了分裂的增量Channel后，会将Channel汇报为TERMINATED。



自动化的负载均衡和良好的水平扩展性

- 运行多个Tunnel Client对同一个Tunnel（TunnelId相同）进行消费时，在Tunnel Client执行Heartbeat时，Tunnel服务端会自动对Channel资源进行重分配，让活跃的Channel尽可能的均摊到每一个Tunnel Client上，达到对资源进行负载均衡的目的。
- 在水平扩展性方面，可以通过增加Tunnel Client的数量来完成，Tunnel Client可以在同一个机器或者不同机器上。

自动化的资源清理和容错处理

- 资源清理：当Tunnel Client没有被正常shutdown时（例如异常退出或者手动结束），会自动进行资源回收，包括释放线程池、自动调用在Channel上注册的shutdown方法、关闭Tunnel连接等。
- 容错处理：当Tunnel Client出现Heartbeat超时等非参数类错误时，表格存储会自动Renew Connect，以保证数据消费可以稳定的进行持续同步。

10.4. 快速入门

在表格存储管理控制台快速体验通道服务功能。

创建数据通道

1. 登录表格存储控制台。
2. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
3. 在**实例详情**页签的**数据表列表**区域，单击数据表名称后选择**实时消费通道**页签或单击 ⋮ 后选择**实时消费通道**。
4. 在**实时消费通道**页签，单击**创建通道**。
5. 在**创建通道**对话框，输入**通道名**，并选择**通道类型**。

   通道服务提供了**增量**、**全量**、**全量加增量**三种类型的分布式数据实时消费通道。本文以**增量**类型为例介绍。

   创建成功后，在操作列单击**展示通道分区列表**，可以查看通道中的数据内容、消费延迟监控、通道分区下的消费数据行数统计。

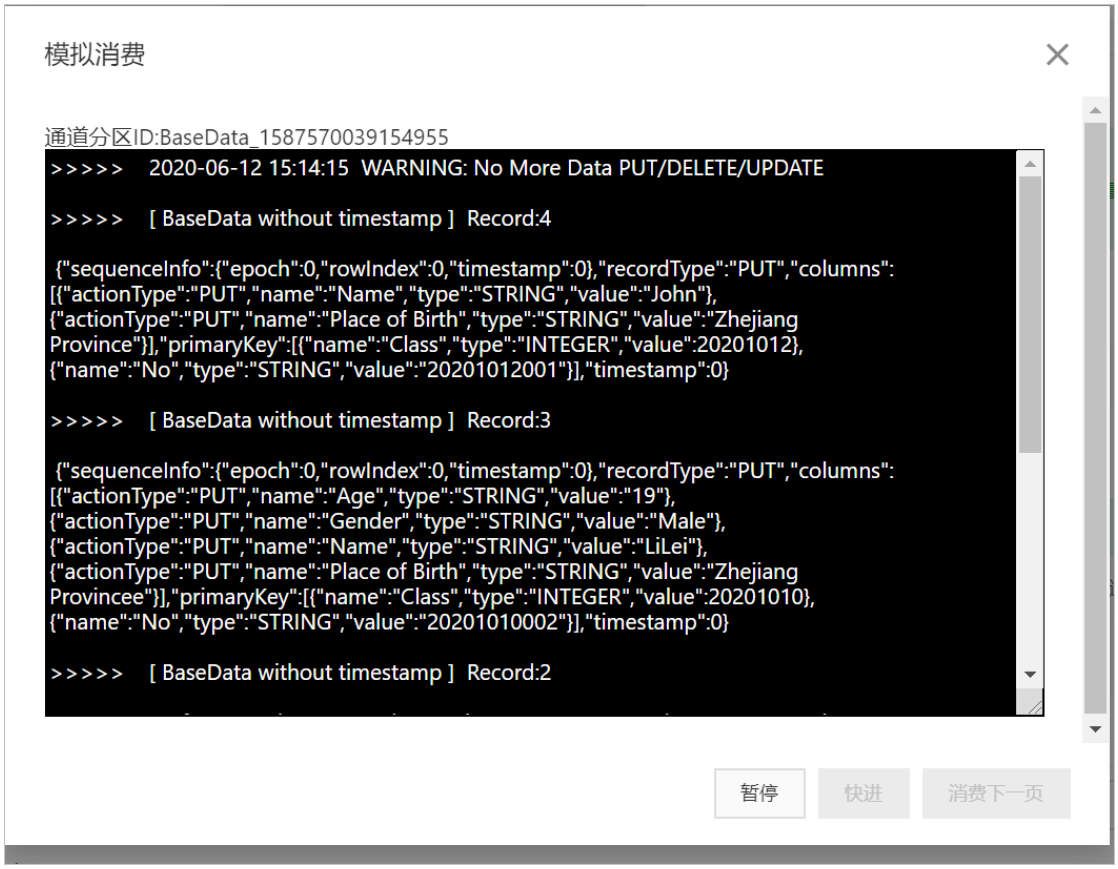
预览通道中的数据格式

创建通道后，通过模拟数据消费可以预览通道中的数据格式。

1. 写入或删除数据，详情请参见控制台读写数据。
2. 预览通道中的数据格式。
   i. 在**概览页**页面，单击实例名称或在操作列单击**实例管理**。
   ii. 在**实例详情**页签的**数据表列表**区域，单击数据表名称后选择**实时消费通道**页签或单击 ⋮ 后选择**实时消费通道**。
   iii. 在**实时消费通道**页签，单击通道操作列的**展示通道分区列表**。
   iv. 在通道分区的右侧单击**模拟消费**。

v. 在**模拟消费**对话框，单击**开始消费**。

消费的数据信息显示在对话框中，如下图所示。



开启通道的数据消费

1. 在通道列表中复制通道ID。
2. 使用任一语言的通道SDK，开启通道的数据消费。

```
//用户自定义数据消费Callback，即实现IChannelProcessor接口（process和shutdown）。
private static class SimpleProcessor implements IChannelProcessor {
    @Override
    public void process(ProcessRecordsInput input) {
        System.out.println("Default record processor, would print records count");
        System.out.println(
            String.format("Process %d records, NextToken: %s", input.getRecords().size(
), input.getNextToken()));
        try {
            //模拟消费处理。
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void shutdown() {
        System.out.println("Mock shutdown");
    }
}
//强烈建议共用一个TunnelWorkerConfig，TunnelWorkerConfig中包括更多的高级参数。
TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
//配置TunnelWorker，并启动自动化的数据处理任务。
TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
try {
    worker.connectAndWorking();
} catch (Exception e) {
    e.printStackTrace();
    worker.shutdown();
    tunnelClient.shutdown();
}
```

查看数据消费日志

数据消费后，可以查看增量数据消费日志，例如消费统计、增量通道分区最新同步时间等。在控制台或者使用describeTunnel接口也可以查看消费延迟、通道分区下的消费数据行数更新。

10.5. 使用SDK

使用SDK快速体验通道服务功能。使用前，您需要了解使用通道服务的注意事项、接口等信息。

注意事项

- TunnelWorkerConfig中默认会启动读数据和处理数据的线程池。如果使用的是单台机器，当需要启动多个TunnelWorker时，建议共用一个TunnelWorkerConfig。
- 在创建全量加增量类型的Tunnel时，由于Tunnel的增量日志最多会保留7天（具体的值和数据表的Stream的日志过期时间一致），全量数据如果在7天内没有消费完成，则此Tunnel进入增量阶段会出现OTSTunnelExpired错误，导致增量数据无法消费。如果您预估全量数据无法在7天内消费完成，请及时联系表格存储技术支持或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。
- TunnelWorker的初始化需要预热时间，该值受TunnelWorkerConfig中的heartbeatIntervalInSec参数影响，可以通过TunnelWorkerConfig中的setHeartbeatIntervalInSec方法配置，默认为30s，最小值为5s。
- 当Tunnel从全量切换至增量阶段时，全量的Channel会结束，增量的Channel会启动，此阶段会有初始化

时间，该值也受TunnelWorkerConfig中的heartbeatIntervalInSec参数影响。

- 当客户端（TunnelWorker）没有被正常shutdown时（例如异常退出或者手动结束），TunnelWorker会自动进行资源的回收，包括释放线程池，自动调用用户在Channel上注册的shutdown方法，关闭Tunnel连接等。

接口

接口	说明
CreateTunnel	创建一个通道。
ListTunnel	列举某个数据表内通道的具体信息。
DescribeTunnel	描述某个通道里的具体Channel信息。
DeleteTunnel	删除一个通道。

使用

您可以使用如下语言的SDK实现通道服务。

- Java SDK
- Go SDK

体验通道服务

使用Java SDK最小化的体验通道服务。

1. 初始化Tunnel Client。

```
//endPoint为表格存储实例的endPoint，例如https://instance.cn-hangzhou.ots.aliyuncs.com。
//accessKeyId和accessKeySecret分别为访问表格存储服务的AccessKey的Id和Secret。
//instanceName为实例名称。
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";
TunnelClient tunnelClient = new TunnelClient(endPoint, accessKeyId, accessKeySecret, in
stanceName);
```

2. 创建新通道。

   请提前创建一张测试表或者使用已有的一张数据表。如果需要新建测试表，可以使用SyncClient中的createTable方法或者使用官网控制台等方式创建。

```
//支持创建TunnelType.BaseData（全量）、TunnelType.Stream（增量）、TunnelType.BaseAndStream（全量加增量）三种类型的Tunnel。
//如下示例为创建全量加增量类型的Tunnel，如果需创建其它类型的Tunnel，则将CreateTunnelRequest中的TunnelType设置为相应的类型。
final String tableName = "testTable";
final String tunnelName = "testTunnel";
CreateTunnelRequest request = new CreateTunnelRequest(tableName, tunnelName, TunnelType.BaseAndStream);
CreateTunnelResponse resp = tunnelClient.createTunnel(request);
//tunnelId用于后续TunnelWorker的初始化，该值也可以通过ListTunnel或者DescribeTunnel获取。
String tunnelId = resp.getTunnelId();
System.out.println("Create Tunnel, Id: " + tunnelId);
```

3. 用户自定义数据消费Callback，开始自动化的数据消费。

```
//用户自定义数据消费Callback，即实现IChannelProcessor接口（process和shutdown）。
private static class SimpleProcessor implements IChannelProcessor {
    @Override
    public void process(ProcessRecordsInput input) {
        //ProcessRecordsInput中包含有拉取到的数据。
        System.out.println("Default record processor, would print records count");
        System.out.println(
            //NextToken用于Tunnel Client的翻页。
            String.format("Process %d records, NextToken: %s", input.getRecords().size(), input.getNextToken()));
        try {
            //模拟消费处理。
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void shutdown() {
        System.out.println("Mock shutdown");
    }
}
//TunnelWorkerConfig默认会启动读数据和处理数据的线程池。
//如果使用的是单台机器，当需要启动多个TunnelWorker时，建议共用一个TunnelWorkerConfig。TunnelWorkerConfig中包括更多的高级参数。
TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
//配置TunnelWorker，并启动自动化的数据处理任务。
TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
try {
    worker.connectAndWorking();
} catch (Exception e) {
    e.printStackTrace();
    worker.shutdown();
    tunnelClient.shutdown();
}
```

配置TunnelWorkerConfig

TunnelWorkerConfig提供Tunnel Client的自定义配置，可根据实际需要配置参数，参数说明请参见下表。

配置	参数	说明
Heartbeat的间隔和超时时间	heartbeatTimeoutInSec	Heartbeat的超时间隔。 默认值为300s。 当Heartbeat发生超时，Tunnel服务端会认为当前TunnelClient不可用（失活），客户端需要重新进行ConnectTunnel。
	heartbeatIntervalInSec	进行Heartbeat的间隔。 Heartbeat用于活跃Channel的探测、Channel状态的更新、（自动化）数据拉取任务的初始化等。 默认值为30s，最小支持配置到5s，单位为s。
记录消费位点的时间间隔	checkpointIntervalInMillis	用户消费完数据后，向Tunnel服务端进行记录消费位点操作（checkpoint）的时间间隔。 默认值为5000ms，单位为ms。 **说明** • 因为读取任务所在机器不同，进程可能会遇到各种类型的错误。例如因为环境因素重启，需要定期对处理完的数据做记录（checkpoint）。当任务重启后，会接着上次的checkpoint继续往后做。在极端情况下，通道服务不保证传给您的记录只有一次，只保证数据至少传一次，且记录的顺序不变。如果出现局部数据重复发送的情况，需要您注意业务的处理逻辑。 • 如果希望减少在出错情况下数据的重复处理，可以增加做checkpoint的频率。但是过于频繁的checkpoint会降低系统的吞吐量，请根据自身业务特点决定checkpoint的操作频率。
客户端的自定义标识	clientTag	客户端的自定义标识，可以生成Tunnel Client ID，用于区分TunnelWorker。
数据处理的自定义Callback	channelProcessor	用户注册的处理数据的Callback，包括process和shutdown方法。
	readRecordsExecutor	用于数据读取的线程池资源。无特殊需求，建议使用默认的配置。

配置	参数	说明
数据读取和数据处理的线程池资源配置	processRecordsExecutor	用于处理数据的线程池资源。无特殊需求，建议使用默认的配置。 **说明** • 自定义上述线程池时，线程池中的线程数要和Tunnel中的Channel数尽可能一致，此时可以保障每个Channel都能很快的分配到计算资源（CPU）。 • 在默认线程池配置中，为了保证吞吐量，表格存储进行了如下操作： ◦ 默认预先分配32个核心线程，以保障数据较小时（Channel数较少时）的实时吞吐量。 ◦ 工作队列的大小适当调小，当在用户数据量比较大（Channel数较多）时，可以更快触发线程池新建线程的策略，及时弹起更多的计算资源。 ◦ 设置了默认的线程保活时间（默认为60s），当数据量降下后，可以及时回收线程资源。
内存控制	maxChannelParallel	读取和处理数据的最大Channel并行度，可用于内存控制。 默认值为-1，表示不限制最大并行度。 **说明** 仅Java SDK 5.10.0及以上版本支持此功能。
最大退避时间配置	maxRetryIntervalInMillis	Tunnel的最大退避时间基准值，最大退避时间在此基准值附近随机变化，具体范围为0.75*maxRetryIntervalInMillis~1.25*maxRetryIntervalInMillis。 默认值为2000ms，最小值为200ms。 **说明** • 仅Java SDK 5.4.0及以上版本支持此功能。 • Tunnel对于数据量较小的情况（单次拉取小于900 KB或500条）会进行一定时间的指数退避，直至达到最大退避时间。

附录：完整代码

```
import com.alicloud.openservices.tablestore.TunnelClient;
```

```
import com.alicloud.openservices.tablestore.model.tunnel.CreateTunnelRequest;
import com.alicloud.openservices.tablestore.model.tunnel.CreateTunnelResponse;
import com.alicloud.openservices.tablestore.model.tunnel.TunnelType;
import com.alicloud.openservices.tablestore.tunnel.worker.IChannelProcessor;
import com.alicloud.openservices.tablestore.tunnel.worker.ProcessRecordsInput;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorker;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorkerConfig;
public class TunnelQuickStart {
    private static class SimpleProcessor implements IChannelProcessor {
        @Override
        public void process(ProcessRecordsInput input) {
            System.out.println("Default record processor, would print records count");
            System.out.println(
                //NextToken用于Tunnel Client的翻页。
                String.format("Process %d records, NextToken: %s", input.getRecords().size(
), input.getNextToken()));
            try {
                //模拟消费处理。
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void shutdown() {
            System.out.println("Mock shutdown");
        }
    }
    public static void main(String[] args) throws Exception {
        //1.初始化Tunnel Client。
        final String endPoint = "";
        final String accessKeyId = "";
        final String accessKeySecret = "";
        final String instanceName = "";
        TunnelClient tunnelClient = new TunnelClient(endPoint, accessKeyId, accessKeySecret
, instanceName);
        //2.创建新通道（此步骤需要提前创建一张测试表，可以使用SyncClient的createTable或者使用官网控制台等方式创建）。
        final String tableName = "testTable";
        final String tunnelName = "testTunnel";
        CreateTunnelRequest request = new CreateTunnelRequest(tableName, tunnelName, Tunnel
Type.BaseAndStream);
        CreateTunnelResponse resp = tunnelClient.createTunnel(request);
        //tunnelId用于后续TunnelWorker的初始化，该值也可以通过ListTunnel或者DescribeTunnel获取。
        String tunnelId = resp.getTunnelId();
        System.out.println("Create Tunnel, Id: " + tunnelId);
        //3.用户自定义数据消费Callback，开始自动化的数据消费。
        //TunnelWorkerConfig中有更多的高级参数。
        TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
        TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
        try {
            worker.connectAndWorking();
        } catch (Exception e) {
            e.printStackTrace();
            worker.shutdown();
```

```
            tunnelClient.shutdown();
        }
    }
}
```

10.6. 错误处理

了解通道服务的错误格式定义和错误码信息。

通道服务在收到异常请求后，会返回protobuf格式错误以及HTTP状态码。

格式定义

protobuf错误格式如下：

```
message Error {
    required string code = 1;
    optional string message = 2;
    optional string tunnel_id = 3;
}
```

错误码

使用SDK时，只需要关心处理逻辑为“返回错误”的错误码，其余错误码会被SDK自动处理或重试，建议直接按相应处理逻辑处理错误码。

HTTP状态码	错误码	描述	处理逻辑
400	OTSParameterInvalid	API请求参数错误或数据表不存在。	返回错误。
400	OTSTunnelExpired	增量Tunnel或全量加增量Tunnel日志过期。	返回错误。
403	OTSPermissionDenied	无指定资源的访问权限。	返回错误。
409	OTSTunnelExist	待创建的Tunnel已经在服务端存在。	返回错误。
400	OTSSequenceNumberNotMatch	checkpoint序列号不匹配，通常是序列号滞后或者Channel消费有竞争。	通过checkpoint API重新获取checkpoint和序列号。
410	OTSResourceGone	Tunnel Client心跳超时。	使用TunnelID重新连接Tunnel Service。
503	OTSTunnelServerUnavailable	Tunnel服务内部错误。	退避重试。

10.7. 增量同步性能白皮书

通过本文您可以了解Tunnel增量性能测试的测试环境、测试工具、测试方案、测试指标、测试结果概述以及测试细则等。

测试环境

- 表格存储实例
  - 实例类型：高性能实例
  - 实例地域：华东1（杭州）
  - 实例地址：私网地址，避免网络的不确定性因素对测试造成的干扰
- 测试机器配置
  - 类型：阿里云ECS
  - 区域：华东1（杭州）
  - 型号：共享通用型(mn4) ecs.mn4.4xlarge
  - 配置：
    - CPU：16核
    - 内存：64 GB
    - 网卡：Red Hat、Inc Virtio network device
    - 操作系统：CentOS 7u2

测试工具

- 压力器

  压力器使用表格存储内部使用的压力测试工具进行数据的批量并发写入，底层基于表格存储Java SDK的BatchWrite操作完成。

- 预分区工具

  使用表格存储内部使用的压力测试工具，配置好表名和分区数等信息，进行表格的自动创建和预分区。

- 速率统计器

  增量的实时消费速率统计基于Tunnel Java SDK完成，在传入的Callback中加入下图中类似的速率统计逻辑，进行速率和消费总行数的实时统计。

  示例

```
private static final Gson GSON = new Gson();
    private static final int CAL_INTERVAL_MILLIS = 5000;
    static class PerfProcessor implements IChannelProcessor {
        private static final AtomicLong counter = new AtomicLong(0);
        private static final AtomicLong latestTs = new AtomicLong(0);
        private static final AtomicLong allCount = new AtomicLong(0);
        @Override
        public void process(ProcessRecordsInput input) {
            counter.addAndGet(input.getRecords().size());
            allCount.addAndGet(input.getRecords().size());
            if (System.currentTimeMillis() - latestTs.get() > CAL_INTERVAL_MILLIS) {
                synchronized (PerfProcessor.class) {
                    if (System.currentTimeMillis() - latestTs.get() > CAL_INTERVAL_MILLIS
) {
                        long seconds = TimeUnit.MILLISECONDS.toSeconds(System.currentTime
Millis() - latestTs.get());
                        PerfElement element = new PerfElement(System.currentTimeMillis(),
counter.get() / seconds, allCount.get());
                        System.out.println(GSON.toJson(element));
                        counter.set(0);
                        latestTs.set(System.currentTimeMillis());
                    }
                }
            }
        }
        @Override
        public void shutdown() {
            System.out.println("Mock shutdown");
        }
    }
```

测试方案

使用Tunnel进行数据同步时，在单Channel间是串行同步（串行是为了保障用户数据的有序性），不同Channel间是相互并行的。在增量场景下，Channel数和表的分区数是相等的。由于Tunnel的整体性能和表的分区数有很大的关联性，所以在本次的性能测试中，将主要考虑不同分区数（Channel数）对于Tunnel增量的同步速率的影响。

> **说明**
>
> - 分区数会随着数据规模自动增长，如您需要预先创建分区，请联系表格存储技术支持。
> - 多台机器消费是自动进行的，每台机器用相同tunnelid启动Tunnel客户端即可，详情参见数据消费框架原理。

- 测试场景

  我们将主要测试以下场景。

  - 单机同步单分区
  - 单机同步4个分区
  - 单机同步8个分区
  - 单机同步32分区

- 单机同步64分区
- 2台机器同步64分区
- 2台机器同步128分区

> **说明** 上述测试场景不是产品能力的极限测试，对表格存储服务端的整体压力较小。

- 测试步骤
  i. 创建数据表并进行预分区（不同分区数的测试都会有单独的一张表）。
  ii. 创建增量通道。
  iii. 使用压力器进行增量数据的写入。
  iv. 使用速率统计器进行QPS的实时统计，同时观察程序占用的系统资源（CPU、内存等）。
  v. 通过监控获得增量数据同步消耗的总网络带宽。
- 测试数据说明

  如下图所示，样例数据由4个主键和1~2个属性列组成，单行的大小在220字节左右。第一主键（分区键）会使用4-Byte-Hash的方式产生，这样可以确保压测数据比较均匀的写入到每个分区上。

详细数据	uid(主键)	name(主键)	class(主键)	time(主键)	col0	col1
详细数据	0000000ecef08efc2fb4...	VuaBXOcb	Bjvta	1548944042906	1548944042906	
详细数据	0000001474c46fc8c539...	uLhDrmQL	icgKK	1548941864548	1548941864548	1548941864548
详细数据	00000044aee39b20e965...	wqvueEsFZ	IgFPVmWbN	1548941564684		1548941564684
详细数据	000000e06aa574ae83c2...	HnEJqUPZy	FdJQKw	1548940905974	1548940905974	1548940905974
详细数据	0000012aebd909a8a4ab...	LoQTHvq	RPPxs	1548940722370	1548940722370	
详细数据	000001a7d9db90f0b101...	XrKTbJMEJ	UwSaTVWmK	1548943865626	1548943865626	
详细数据	0000026302f7effcb293...	opVMvvXq	vnLJFDdQ	1548940665078	1548940665078	

测试指标

本次测试主要包含以下几项指标。

- QPS（row）：每秒同步的数据行数。
- Avg Latency（ms/1000行）：同步1000行数据所需的时间，单位为毫秒。
- CPU（核）：数据同步消耗的单核CPU总数。
- Mem（GB）：数据同步消耗的物理总内存。
- 带宽（MBps）：数据同步消耗的总网络带宽。

> **说明** 本次性能测试不是产品能力的极限，是从实际使用角度出发进行的性能测试。

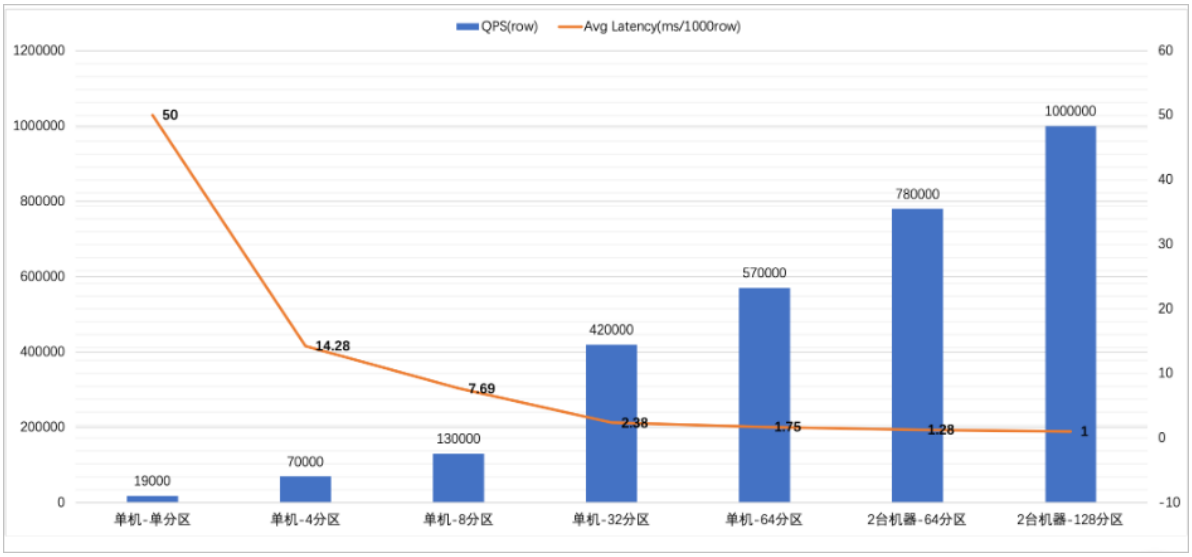
测试结果

该部分主要概述各个场景下的指标测试结果，测试的细节可以参见测试细则部分。

- QPS和延迟

  下图展示的是各个场景下每秒同步的数据行数和同步1000行数据所需的时间。从图中我们看出QPS的增长和分区数的增加呈线性关系。

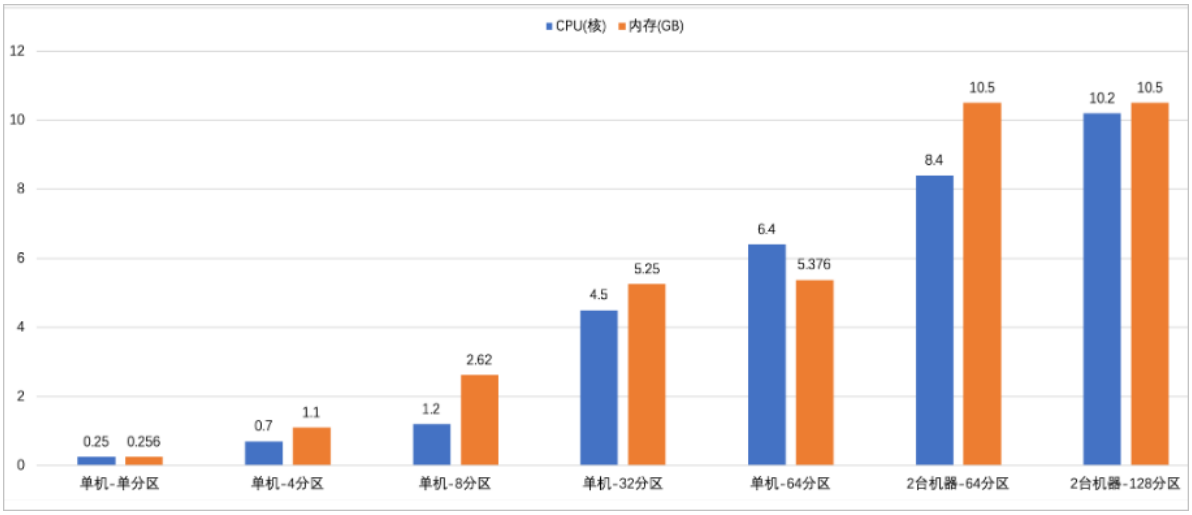
在本次测试中，单机同步64分区场景下，将千兆的网卡成功打爆（参见测试细则部分），导致只有57万的QPS。两台机器对64分区进行同步后，平均QPS成功达到了78万行左右，约等于单机-32分区场景下（42万）的两倍速率。而在最后的两台机器-128分区场景下，Tunnel增量同步的QPS也成功达到了100万行。



- 系统资源消耗

  下图展示的是各个场景下CPU和内存的消耗情况，CPU基本上和分区数呈线性关系。

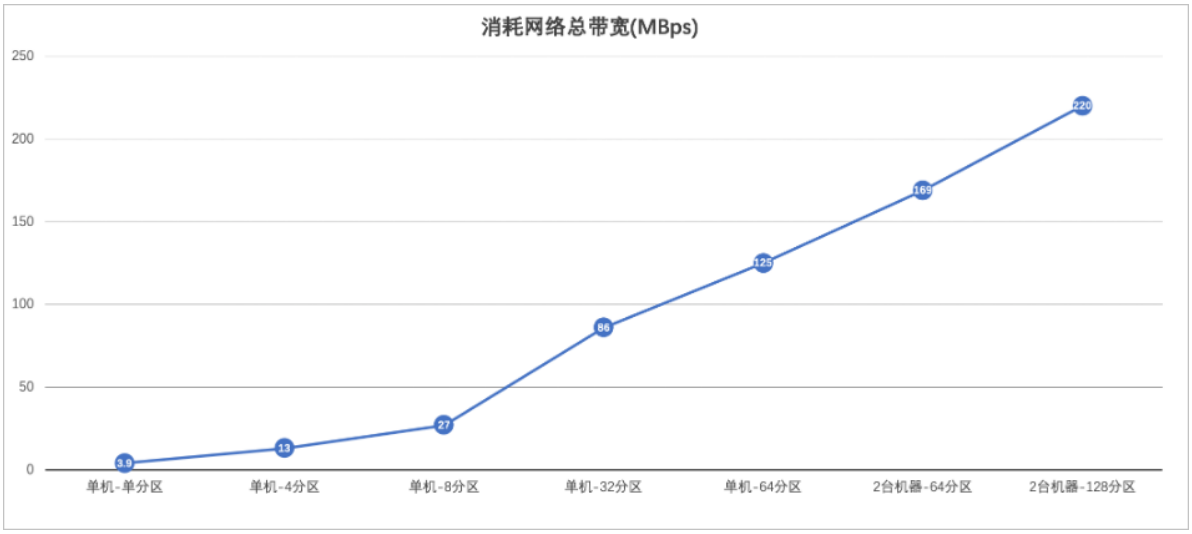
  在单机-单分区场景下，消耗的CPU为0.25个单核CPU。2台机器-128个分区的场景下，当同步QPS达到100万行时，消耗的CPU也仅为10.2个单核CPU。从内存消耗方面看。在分区数较少时，CPU和分区数呈线性关系，而在分区数增加较多（32个和64个）时，单机内存消耗基本维持在5.3 GB左右。



- 网络总带宽消耗

  下图展示的是增量同步消耗的总带宽，从图中我们可以看出带宽和Channel数的线性关系（略单机-16分区场景）。

  在单机-64分区场景下，我们可以看到带宽总消耗为125 MBps，已经成功把千兆网卡打爆，而在换成2台机器-64分区进行数据消费后，我们发现64分区真正的吞吐量为169 MBps，和单机-32分区的86 MBps的两倍近乎相等。而在两台机器-128分区的100万QPS场景中，总吞吐量也达到了220 MBps。



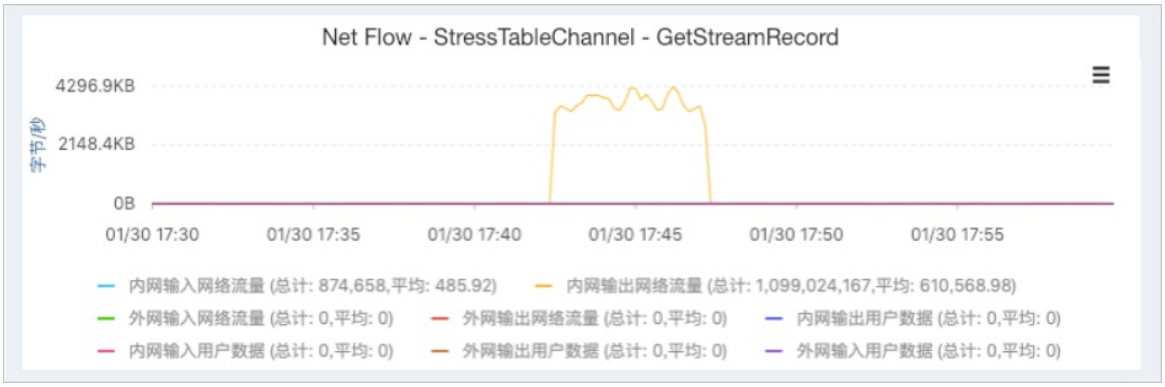
测试细则

- 单机单Channel：1.9万QPS
  - 测试时间：2019/1/30 17:40
  - QPS：稳定速率19000行/秒左右；峰值速率：21800行
  - Latency：50ms/1000行左右


```
{"timestamp":1548841516239,"speed":19000,"totalCount":3094000}
{"timestamp":1548841521290,"speed":19200,"totalCount":3190000}
{"timestamp":1548841526318,"speed":20400,"totalCount":3292000}
{"timestamp":1548841531357,"speed":19600,"totalCount":3390000}
{"timestamp":1548841536396,"speed":19400,"totalCount":3487000}
{"timestamp":1548841541418,"speed":17800,"totalCount":3576000}
{"timestamp":1548841546472,"speed":17600,"totalCount":3664000}
{"timestamp":1548841551532,"speed":17200,"totalCount":3750000}
{"timestamp":1548841556572,"speed":17400,"totalCount":3837000}
{"timestamp":1548841561631,"speed":17400,"totalCount":3924000}
{"timestamp":1548841566664,"speed":20000,"totalCount":4024000}
{"timestamp":1548841571693,"speed":21600,"totalCount":4132000}
{"timestamp":1548841576721,"speed":21200,"totalCount":4238000}
{"timestamp":1548841581765,"speed":21800,"totalCount":4347000}
{"timestamp":1548841586787,"speed":21400,"totalCount":4454000}
{"timestamp":1548841591798,"speed":17800,"totalCount":4543000}
{"timestamp":1548841596812,"speed":17800,"totalCount":4632000}
{"timestamp":1548841601825,"speed":17800,"totalCount":4721000}
{"timestamp":1548841606861,"speed":16200,"totalCount":4802000}
{"timestamp":1548841611884,"speed":17400,"totalCount":4889000}
{"timestamp":1548841616912,"speed":17200,"totalCount":4975000}
{"timestamp":1548841621966,"speed":18000,"totalCount":5065000}
{"timestamp":1548841626988,"speed":17600,"totalCount":5153000}
{"timestamp":1548841632035,"speed":18200,"totalCount":5244000}
```


  - CPU占用：单核25%左右

- 内存占用：总物理内存0.4%左右，即0.256 GB左右（测试机器内存为64 GB）
- 网络带宽消耗：4000 KB/s左右



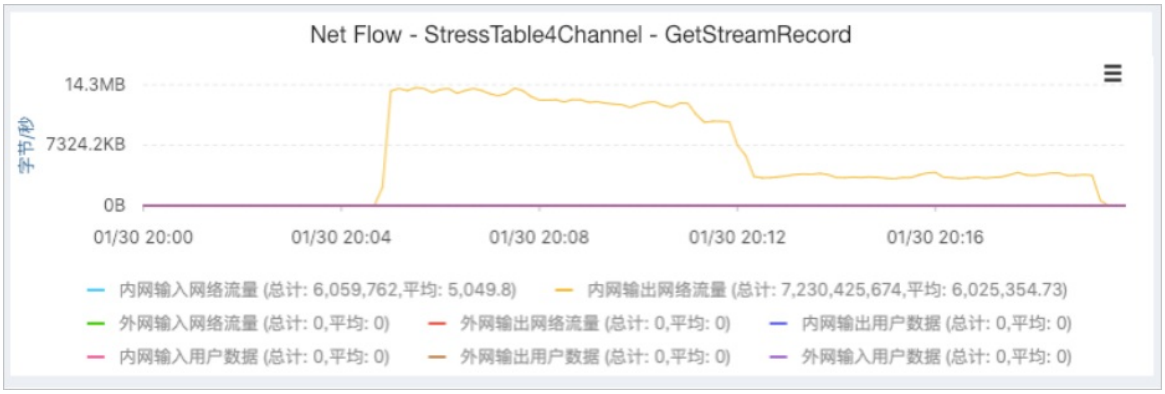
- 单机4分区：7万QPS

- 测试时间：2019/1/30 20:00
- QPS：稳定速率70000行/秒左右，峰值速度72400行/秒

- Latency：14.28ms/1000行左右


```
{"timestamp":1548849903425,"speed":68200,"totalCount":345000}
{"timestamp":1548849908451,"speed":69400,"totalCount":692000}
{"timestamp":1548849913454,"speed":71800,"totalCount":1051000}
{"timestamp":1548849918470,"speed":70600,"totalCount":1404000}
{"timestamp":1548849923479,"speed":69400,"totalCount":1751000}
{"timestamp":1548849928501,"speed":71000,"totalCount":2106000}
{"timestamp":1548849933544,"speed":70200,"totalCount":2457000}
{"timestamp":1548849938558,"speed":71400,"totalCount":2814000}
{"timestamp":1548849943585,"speed":71600,"totalCount":3172000}
{"timestamp":1548849948600,"speed":70600,"totalCount":3525000}
{"timestamp":1548849953609,"speed":71000,"totalCount":3880000}
{"timestamp":1548849958624,"speed":68000,"totalCount":4220000}
{"timestamp":1548849963645,"speed":69000,"totalCount":4565000}
{"timestamp":1548849968651,"speed":70200,"totalCount":4916000}
{"timestamp":1548849973661,"speed":70600,"totalCount":5269000}
{"timestamp":1548849978664,"speed":72400,"totalCount":5631000}
{"timestamp":1548849983676,"speed":68000,"totalCount":5971000}
{"timestamp":1548849988699,"speed":68000,"totalCount":6311000}
```


- CPU占用：单核70%左右
- 内存占用：物理内存1.9%左右，即1.1 GB左右。（测试机器内存为64 GB）
- 网络带宽消耗：13 MBps左右

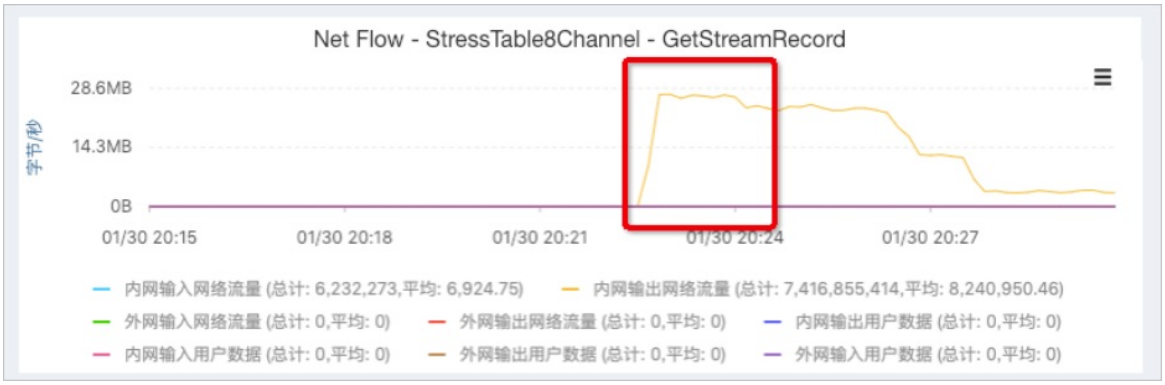


- 单机8分区：13万QPS

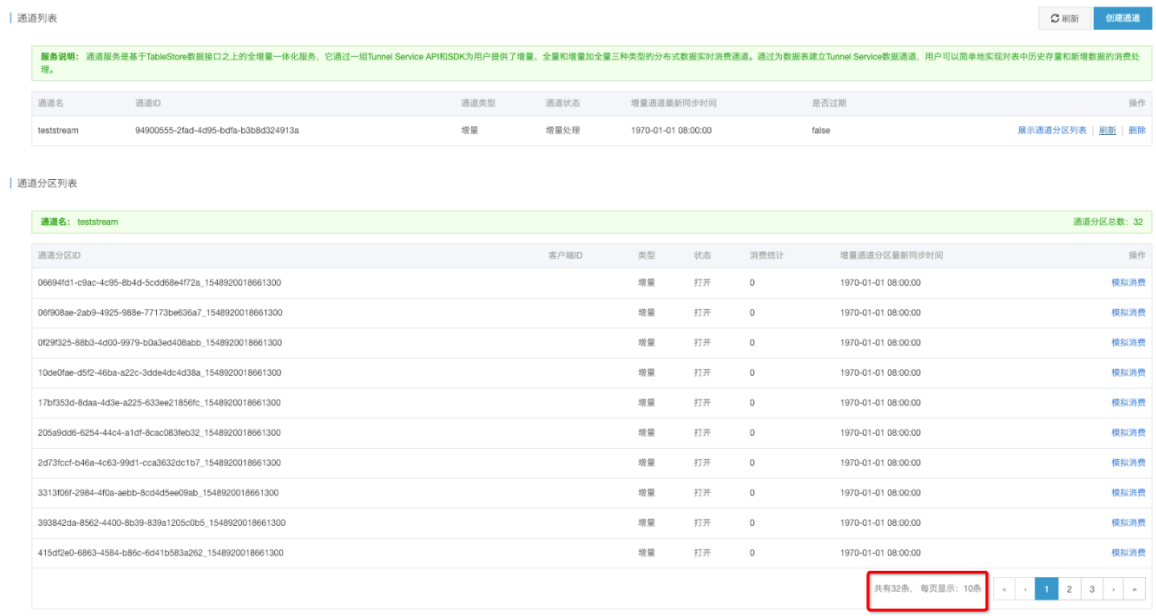
- 测试时间：2019/1/30 20:20
- QPS：稳定速率130000行/秒，峰值速率141644行/秒
- Latency：7.69ms/1000行左右


```
{"timestamp":1548850971326,"speed":136000,"totalCount":688000}
{"timestamp":1548850976329,"speed":137600,"totalCount":1376000}
{"timestamp":1548850981335,"speed":137800,"totalCount":2065000}
{"timestamp":1548850986351,"speed":139800,"totalCount":2764000}
{"timestamp":1548850991360,"speed":139200,"totalCount":3460000}
{"timestamp":1548850996362,"speed":134600,"totalCount":4133000}
{"timestamp":1548851001377,"speed":133800,"totalCount":4802000}
{"timestamp":1548851006389,"speed":137800,"totalCount":5491000}
{"timestamp":1548851011390,"speed":138000,"totalCount":6181000}
{"timestamp":1548851016412,"speed":137600,"totalCount":6869000}
{"timestamp":1548851021417,"speed":135600,"totalCount":7547000}
{"timestamp":1548851026418,"speed":134800,"totalCount":8221000}
{"timestamp":1548851031420,"speed":134400,"totalCount":8893000}
{"timestamp":1548851036430,"speed":136600,"totalCount":9576000}
{"timestamp":1548851041443,"speed":141400,"totalCount":10283000}
{"timestamp":1548851046452,"speed":141644,"totalCount":10991220}
{"timestamp":1548851051455,"speed":124928,"totalCount":11615860}
{"timestamp":1548851056456,"speed":122201,"totalCount":12226865}
{"timestamp":1548851061466,"speed":121944,"totalCount":12836585}
```


- CPU占用：单核120%左右
- 内存占用：物理总内存4.1%左右，即2.62 GB左右（测试机器内存为64 GB）
- 消耗网络带宽：27 MBps左右



- 单机32分区：42万QPS

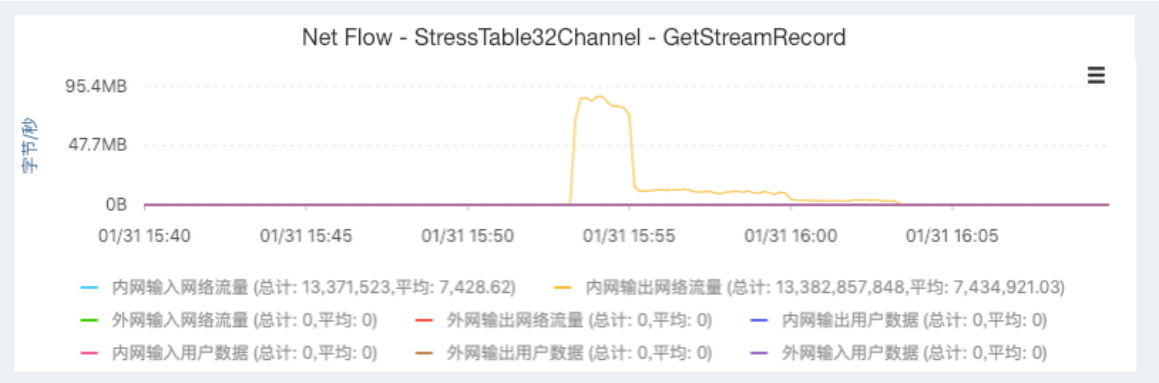


- 测试时间：2019/1/31 15:50
- QPS：稳定速率42万行/秒，峰值速率447600行/秒
- Latency：2.38ms/1000行

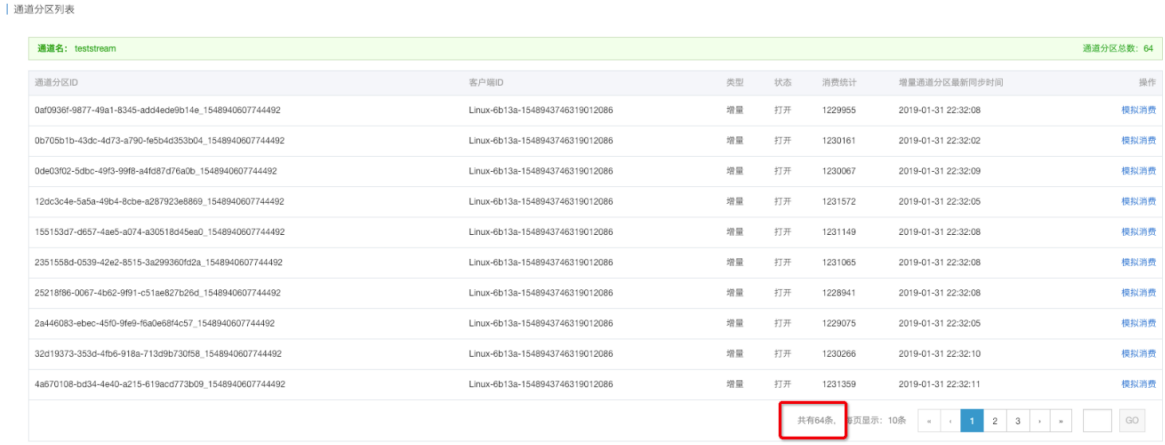
```
{"timestamp":1548921206560,"speed":401800,"totalCount":2016000}
{"timestamp":1548921211565,"speed":435600,"totalCount":4194000}
{"timestamp":1548921216569,"speed":440200,"totalCount":6397000}
{"timestamp":1548921221571,"speed":439000,"totalCount":8592000}
{"timestamp":1548921226573,"speed":440800,"totalCount":10796000}
{"timestamp":1548921231577,"speed":437400,"totalCount":12983000}
{"timestamp":1548921236579,"speed":421400,"totalCount":15090000}
{"timestamp":1548921241580,"speed":434400,"totalCount":17262000}
{"timestamp":1548921246581,"speed":445400,"totalCount":19489000}
{"timestamp":1548921251583,"speed":447600,"totalCount":21727000}
{"timestamp":1548921256591,"speed":447400,"totalCount":23964000}
{"timestamp":1548921261594,"speed":440800,"totalCount":26169000}
{"timestamp":1548921266595,"speed":425200,"totalCount":28295000}
{"timestamp":1548921271599,"speed":408600,"totalCount":30339000}
{"timestamp":1548921276603,"speed":403800,"totalCount":32358000}
{"timestamp":1548921281608,"speed":405000,"totalCount":34383000}
{"timestamp":1548921286610,"speed":403400,"totalCount":36400000}
{"timestamp":1548921291612,"speed":409479,"totalCount":38447399}
{"timestamp":1548921296617,"speed":400896,"totalCount":40452882}
{"timestamp":1548921301618,"speed":391936,"totalCount":42412564}
```

- CPU占用：单核450%左右
- 内存占用：8.2%左右，即5.25 GB左右（物理内存64 GB）

- 增量数据消耗网络带宽：86 MBps左右



- 单机64分区（千兆网卡被打满）：57万QPS

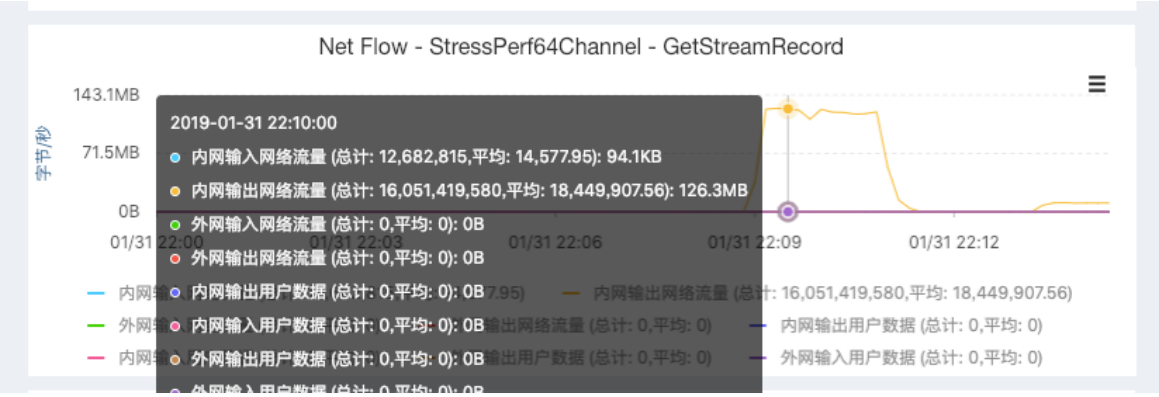


- 测试时间：2019/1/31 22:10
- QPS：稳定速率57万行/秒左右，峰值速率581400行/秒
- Latency：1.75ms/1000行左右

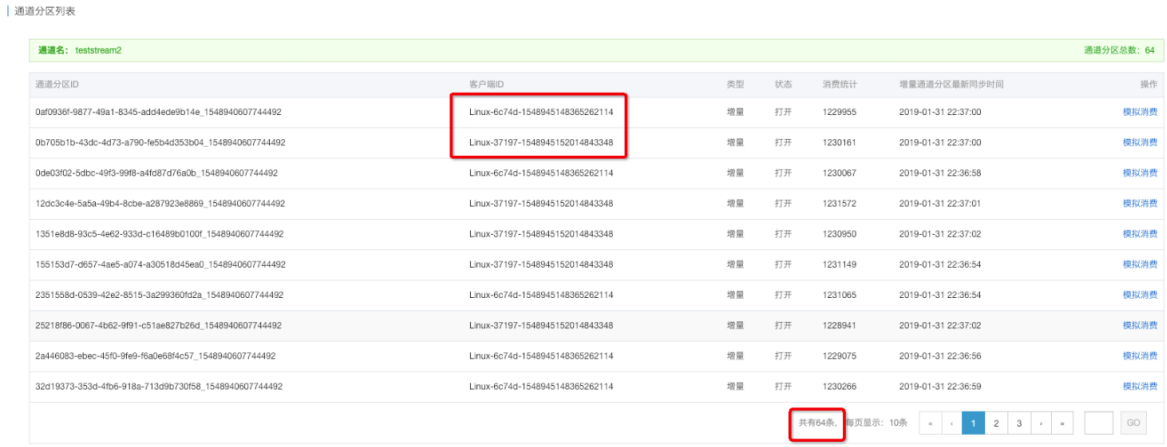

```
{"timestamp":1548943781849,"speed":536200,"totalCount":2688000}
{"timestamp":1548943786851,"speed":572000,"totalCount":5548000}
{"timestamp":1548943791852,"speed":578800,"totalCount":8442000}
{"timestamp":1548943796855,"speed":581800,"totalCount":11351000}
{"timestamp":1548943801857,"speed":576200,"totalCount":14232000}
{"timestamp":1548943806859,"speed":576200,"totalCount":17113000}
{"timestamp":1548943811860,"speed":581400,"totalCount":20020000}
{"timestamp":1548943816861,"speed":571600,"totalCount":22878000}
{"timestamp":1548943821864,"speed":555800,"totalCount":25657000}
{"timestamp":1548943826866,"speed":555000,"totalCount":28432000}
{"timestamp":1548943831869,"speed":577000,"totalCount":31317000}
{"timestamp":1548943836870,"speed":578800,"totalCount":34211000}
{"timestamp":1548943841871,"speed":559600,"totalCount":37009000}
{"timestamp":1548943846875,"speed":561400,"totalCount":39816000}
{"timestamp":1548943851878,"speed":551600,"totalCount":42574000}
{"timestamp":1548943856879,"speed":560600,"totalCount":45377000}
```


- CPU占用：单核640%左右
- 内存占用：8.4%左右，即5.376 GB左右

- 增量数据消耗网络带宽：125 MBps左右（达到千兆网卡的速率极限）



- 2台机器共同消费64分区：78万QPS



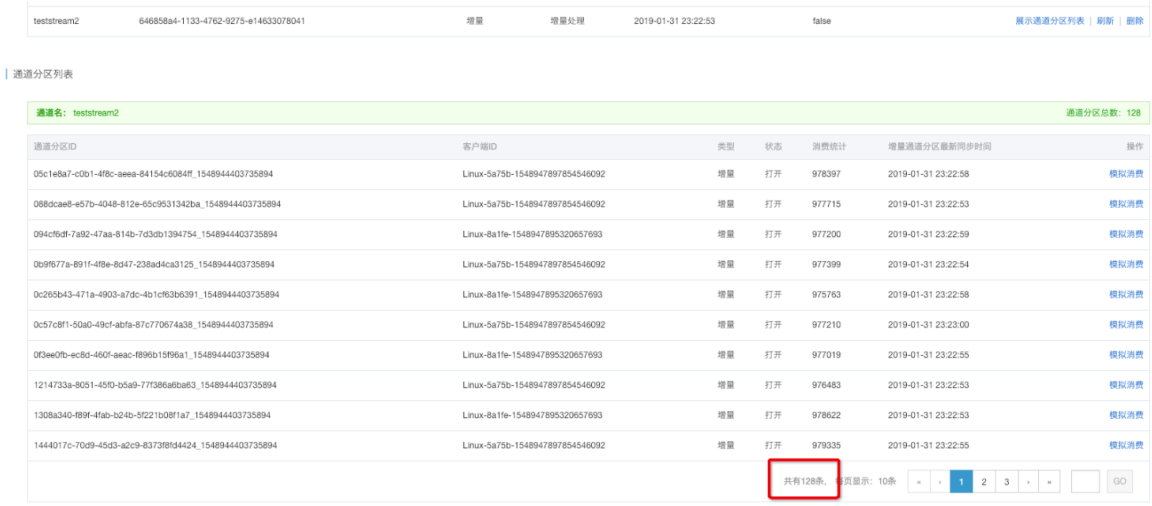
- 测试时间：2018/1/31 22:30
- QPS：每台稳定速率在39万行/秒左右，总的稳定速率在78万行/秒左右
- Latency：1.28ms/1000行

```
{"timestamp":1548945217504,"speed":380200,"totalCount":1902000}
{"timestamp":1548945222507,"speed":392400,"totalCount":3864000}
{"timestamp":1548945227509,"speed":392800,"totalCount":5828000}
{"timestamp":1548945232515,"speed":388200,"totalCount":7769000}
{"timestamp":1548945237517,"speed":394200,"totalCount":9740000}
{"timestamp":1548945242518,"speed":392800,"totalCount":11704000}
{"timestamp":1548945247521,"speed":391000,"totalCount":13660000}
{"timestamp":1548945252522,"speed":382200,"totalCount":15571000}
{"timestamp":1548945257523,"speed":383400,"totalCount":17488000}
{"timestamp":1548945262527,"speed":385600,"totalCount":19416000}
{"timestamp":1548945267528,"speed":385000,"totalCount":21341000}
{"timestamp":1548945272532,"speed":388600,"totalCount":23284000}
{"timestamp":1548945277538,"speed":385800,"totalCount":25213000}
{"timestamp":1548945282541,"speed":387400,"totalCount":27150000}
{"timestamp":1548945287546,"speed":392200,"totalCount":29111000}
```

- CPU占用：每台单核420%左右，共单核840%
- 内存占用：每台8.2%左右，共16.4%（10.5 GB）

- 增量数据消耗总网络带宽：169 MBps左右（和单机64分区对比，可以看出单机的网络已经成为瓶颈）

- 2台机器共同消费128分区（两台千兆机器的网卡近乎被打满）：100万QPS



- 测试时间：2018/1/31 23:20
- QPS：每台稳定速率在50万行/秒左右，总的稳定速率在100万行/秒左右
- Latency：1ms/1000行左右

```
$tail -f perf_128channel_2machine.txt
{"timestamp":1548948013375,"speed":492400,"totalCount":27363000}
{"timestamp":1548948018378,"speed":499800,"totalCount":29862000}
{"timestamp":1548948023383,"speed":499800,"totalCount":32361000}
{"timestamp":1548948028387,"speed":504400,"totalCount":34883000}
{"timestamp":1548948033389,"speed":504200,"totalCount":37404000}
{"timestamp":1548948038390,"speed":506800,"totalCount":39939000}
{"timestamp":1548948043391,"speed":500800,"totalCount":42443000}
{"timestamp":1548948048393,"speed":497400,"totalCount":44930000}
{"timestamp":1548948053394,"speed":511800,"totalCount":47490000}
{"timestamp":1548948058397,"speed":519600,"totalCount":50089000}
{"timestamp":1548948063398,"speed":518800,"totalCount":52683000}
{"timestamp":1548948068399,"speed":519600,"totalCount":55281000}
{"timestamp":1548948073401,"speed":503800,"totalCount":57800000}
```

- CPU占用：每台单核560%左右，两台共计单核1020%
- 内存占用：每台8.2%左右，共16.4%（10.5 GB）

- 增量数据消耗总网络带宽：220 MBps左右

总结

通过这次对于增量性能的实际测试，我们发现了单分区（或分区数较少）的速率主要取决于服务器端的读盘等延迟，本身机器资源的消耗很小。而随着分区数增长，Tunnel增量的整体吞吐也进行了线性的增长直至达到系统的瓶颈（本文中是网络带宽）。最后，在单机资源被打满的情况下，我们也可以通过添加新的机器资源进一步的提升系统整体的吞吐量，有效的验证了Tunnel具备良好的水平扩展性。

11.数据湖投递

11.1. 概述

表格存储数据湖投递可以全量备份或实时投递数据到数据湖OSS中存储，以满足更低成本的历史数据存储，以及更大规模的离线和准实时数据分析需求。

应用场景

利用数据湖投递可以实现如下场景需求：

- 冷热数据分层

  数据湖投递结合表格存储的数据生命周期功能，可以快速实现OSS低成本存储全量数据，表格存储提供热数据的低延迟查询和分析的需求。

- 全量数据备份

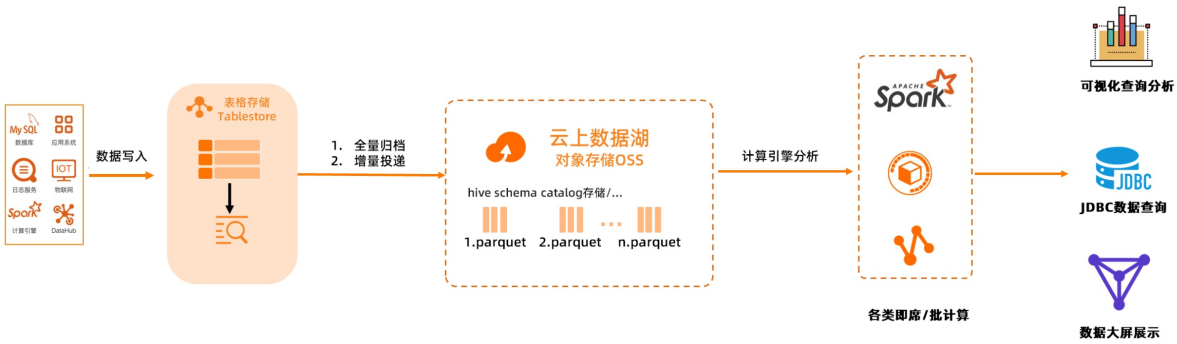
  数据湖投递可以自动将表格存储的全表数据投递到OSS Bucket中，作为备份归档数据。

- 大规模实时数据分析

  数据湖投递可以实时（每2分钟）投递增量的表格存储数据到OSS，投递的数据支持按系统时间分区、Parquet列存格式存储；再利用OSS的高读带宽和列存面向扫描场景优化实现高效实时数据分析。

- 加速SQL分析性能

  当表格存储数据未建立多元索引且查询条件中不包含主键列的过滤条件时，可以通过数据投递自动同步数据到OSS，再利用DLA+OSS数据扫描实现SQL分析加速。



功能特性

数据湖投递的主要功能特性如下：

- 数据湖投递会自动拉取表格存储的全量和增量数据，数据积累到合适大小或者投递超过2分钟后，持久化到OSS中存储。
- 支持配置增量、全量、全量&增量三种数据投递模式，投递的所有数据均按照Parquet列存格式存储。
- 支持监控实时投递的同步时间点，数据湖投递提供了DescribeDeliveryTask API，该API会返回任务已成功投递的实时数据位点。

核心优势

- 易于使用

只需在控制台完成简单配置，即可实现全托管的表格存储到OSS的自动投递。无需监控和运维，投递任务保证SLA内同步任务平稳执行和随吞吐规模扩展。

- 全增量一体

  提供全增量一体的数据投递能力。增量投递任务提供准实时体验，持续拉取新数据并缓存两分钟后写入OSS。

- 与计算生态无缝集成

  投递的数据兼容开源生态标准，按照Parquet列存格式存储，兼容Hive命名规范。使用数据湖分析和E-MapReduce可以直接对投递到OSS的数据进行外表分析。

- 数据分层的存储与访问体验

  数据投递到OSS后，表格存储提供数据表、索引表、投递OSS等分层数据。满足不同场景的使用分析需求。

11.2. 快速入门

通过表格存储控制台创建投递任务，将表格存储数据表中的数据投递到OSS Bucket中存储。

前提条件

已开通OSS服务且在表格存储实例所在地域创建Bucket，详情请参见开通OSS服务。

> **说明**
>
> 数据湖投递支持投递到和表格存储相同地域的任意OSS Bucket中。如需投递到其他数仓存储（例如MaxCompute），请提交工单申请。

注意事项

- 目前支持使用数据湖投递功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）和华北3（张家口）。
- 数据湖投递不支持同步删除操作，表格存储中的删除操作在数据投递时会被忽略，已投递到OSS中的数据不会被删除。
- 新建数据投递任务时存在最多1分钟的初始化时间。
- 数据同步存在延迟，写入速率稳定时，延迟在3分钟内。数据同步的P99延迟在10分钟内。

> **说明**
>
> P99延迟表示过去10秒内最慢的1%的请求的平均延迟。

创建投递任务

1. 登录表格存储控制台。
2. 选择地域，单击实例名称或者实例**操作**列的**实例管理**。
3. 在**实例管理**页面，单击**数据湖投递**。
4. （可选）创建服务关联角色AliyunServiceRoleForOTSDataDelivery。

首次配置数据湖投递时，需要创建表格存储服务关联角色AliyunServiceRoleForOTSDataDelivery，该角色用于授权表格存储服务写入OSS Bucket的权限，具体操作，请参见表格存储服务关联角色。

> **说明**
>
> 关于服务关联角色的更多信息，请参见服务关联角色。

i. 在**数据湖投递**页面，单击**数据湖投递关联角色说明**。

ii. 在**数据湖投递服务关联角色**对话框，查看相关说明，单击**确认创建**。

5. 创建投递任务。

i. 在**数据湖投递**页面，单击**创建投递任务**。

ii. 在**新建投递任务**对话框，配置投递参数。

参数	说明
任务名称	投递任务名称。 名称只能包含英文小写字母（a~z）、数字和短横线（-），开头和结尾必须为英文小写字母或数字，且长度为3~16字符。
目标region	表格存储实例和OSS Bucket所在地域。
目标Table	表格存储数据表名称。
目标OSS Bucket	OSS Bucket名称。 **注意** 该OSS Bucket必须已存在且与表格存储实例在同一地域。
投递路径（前缀）	OSS Bucket中的目录前缀，将表格存储的数据投递到该OSS Bucket目录中。投递路径中支持引用$yyyy、$MM、$dd、$HH、$mm五种时间变量。更多信息，请参见按时间分区。 ■ 当投递路径中引用时间变量时，可以按数据的写入时间动态生成OSS目录，实现hive partition naming style的数据时间分区，从而按照时间分区组织OSS中的文件分布。 ■ 当投递路径中不引用时间变量时，所有文件会被投递到固定的OSS前缀目录中。

参数	说明
投递类型	投递任务的类型，包括如下选项： ■ 增量：只同步增量数据。 ■ 全量：一次性全表扫描数据同步。 ■ 全量&增量：全量数据同步完成后，再同步增量数据 其中增量数据同步时可以获取最新投递时间和了解当前投递状态。
投递文件格式	投递的数据以Parquet列存格式存储，数据湖投递默认使用PLAIN编码方式，PLAIN编码方式支持任意类型数据。
Schema生成方式	指定需要投递的数据列，可以选择任意字段以任意顺序、名称写入列存文件，OSS的列存数据会按Schema数组中的数据列先后顺序分布。
Schema配置	根据选择的Schema生成方式配置投递Schema。 ■ 当**Schema生成方式**配置为**手动录入**时，需要手动配置投递字段的源表字段、目标字段名和目标字段类型。 ■ 当**Schema生成方式**配置为**自动生成**时，系统会自动匹配识别投递字段。 **注意** 投递数据的字段类型必须与数据源的字段类型匹配，否则会作为脏数据丢弃。关于字段类型映射的更多信息，请参见数据格式映射。 配置投递Schema时，可以执行如下操作： ■ 单击**新增投递字段**，新增投递字段。 ■ 在**操作**列单击图标或图标，调整投递字段的顺序。 ■ 在**操作**列单击图标，删除投递字段。

6. 单击**确定**。

   在**建表语句**对话框，您可以查看自动生成的DLA外表和EMR外表的建表语句，直接复制建表语句可以在DLA和EMR中快速创建外表，便于访问OSS中的数据。

   创建投递任务后，您可以执行如下操作：

   - 查看数据投递的详细信息，例如任务名称、表名、投递OSS Bucket、前缀路径、最新同步时间、状态等。
   - 查看或复制建表语句。

     在**操作**列单击**建表语句**，可以查看或复制通过计算分析引擎（例如DLA、EMR）创建外表的建表语句，具体操作，请分别参见使用DLA和使用EMR。

- 查看投递的错误信息。

  当OSS Bucket、投递权限相关配置不正确时，数据投递会无法成功完成，此时在投递任务的状态界面可以查看相关的错误信息。错误信息的处理详情请参见错误处理。

- 删除投递任务。

  在**操作**列单击**删除**，可以删除投递任务。删除处于初始化阶段的投递任务时，系统会返回错误，请稍后重试删除。

查看OSS数据

投递任务初始化完成且有数据投递后，可以通过OSS的控制台、API或者SDK，计算分析引擎（例如EMR等）查看投递到OSS的数据，具体操作，请参见文件概览。

OSS Object的地址格式如下所示。

```
oss://BucketName/TaskPrefix/TaskName_ConcurrentID_TaskPrefix__SequenceID
```

其中BucketName为Bucket名称，TaskPrefix为目录前缀，TaskName为投递任务名称，ConcurrentID为投递系统内部的并发编号，从0开始流量增大时任务并发会自动增加，TaskPrefix为任务的前缀信息，SequenceID为投递的文件编号，从1开始递增。

按时间分区

数据投递支持提取数据写入表格存储的时间，写入时间$yyyy（年份数字）、$MM（两位月份数字）、$dd（两位日期数字）、$HH（两位小时数字）、$mm（两位分钟数字）转化后可以作为投递到OSS Bucket中的目录前缀。

> **说明**
>
> OSS中的文件大小不宜过小，推荐4 MB或者更大，同时计算分析引擎加载OSS时，分区越多，加载事务的执行时间也会越长，因此时间分区粒度不宜过细，在多数实时写入流量场景中，宜按天或者按小时分区，不需要到分钟的分区粒度。

以2020年08月31日16点03分写入表格存储的数据投递为例，OSS中该日志的第一个数据文件目录，根据投递前缀配置，不同的文件路径请参见下表。

OSS Bucket	TaskName	投递前缀	OSS实际文件路径
myBucket	testTask	myPrefix	oss://myBucket/myPrefix/testTask_0_myPrefix__1
myBucket	testTaskTimeParitioned	myPrefix/$yyyy/$MM/$dd/$HH/$mm	oss://myBucket/myPrefix/2020/08/31/16/03/testTaskTimeParitioned_0_myPrefix_2020_08_31_16_03__1
myBucket	testTaskTimeParitionedHiveNamingStyle	myPrefix/year=$yyyy/month=$MM/day=$dd	oss://myBucket/myPrefix/year=2020/month=08/day=31/testTaskTimeParitionedHiveNamingStyle_0_myPrefix_year=2020_month=08

OSS Bucket	TaskName	投递前缀	OSS实际文件路径
myBucket	testTaskDs	ds=$yyyy$MM$dd	oss://myBucket/ds=20200831/testTaskDs_0_ds=20200831__0

数据格式映射

Parquet Logical Type	表格存储数据类型
Boolean	Boolean
Int64	Int64
Double	Double
UTF8	String

错误处理

错误信息	错误原因	处理方法
UnAuthorized	没有权限。	确认RAM中的服务关联角色AliyunServiceRoleForOTSDataDelivery是否存在。 当服务关联角色不存在时需要重新新建投递任务触发表格存储为用户创建该角色。
InvalidOssBucket	OSS Bucket不存在。	• 确认OSS Bucket所在地域是否与表格存储实例相同。 • 确认OSS Bucket是否存在。 当OSS Bucket重新创建后，所有数据会重试写入OSS Bucket，投递进度也会正常更新。

11.3. 使用SDK

使用SDK进行数据投递前，您需要了解使用数据湖投递功能的注意事项、接口等信息。创建投递任务后，表格存储数据表中的数据会自动投递到OSS Bucket中存储。

注意事项

- 目前支持使用数据湖投递功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）和华北3（张家口）。

- 数据湖投递不支持同步删除操作，表格存储中的删除操作在数据投递时会被忽略，已投递到OSS中的数据不会被删除。
- 新建数据投递任务时存在最多1分钟的初始化时间。
- 数据同步存在延迟，写入速率稳定时，延迟在3分钟内。数据同步的P99延迟在10分钟内。

> **说明**
>
> P99延迟表示过去10秒内最慢的1%的请求的平均延迟。

接口

接口	说明
CreateDeliveryTask	创建一个投递任务。
ListDeliveryTask	列出一个数据表所有的投递任务信息。
DescribeDeliveryTask	查询投递任务描述信息。
DeleteDeliveryTask	删除一个投递任务。

使用

您可以使用如下语言的SDK实现数据湖投递功能。

- Java SDK
- Go SDK

参数

参数	说明
tableName	数据表名称。
taskName	投递任务名称。 名称只能包含英文小写字母（a~z）、数字和短横线（-），开头和结尾必须为英文小写字母或数字，且长度为3~16字符。

参数	说明

taskConfig	投递任务配置，包括如下选项：
	• ossPrefix：OSS Bucket中的目录前缀，将表格存储的数据投递到该OSS Bucket目录中。投递路径中支持引用$yyyy、$MM、$dd、$HH、$mm五种时间变量。
	  ○ 当投递路径中引用时间变量时，可以按数据的写入时间动态生成OSS目录，实现hive partition naming style的数据时间分区，从而按照时间分区组织OSS中的文件分布。
	  ○ 当投递路径中不引用时间变量时，所有文件会被投递到固定的OSS前缀目录中。
	• ossBucket：OSS Bucket名称。
	• ossEndpoint：OSS Bucket所在地域的服务地址。
	• ossStsRole：表格存储服务关联角色的ARN信息。
	• format：投递的数据的存储以Parquet列存格式存储，数据湖投递默认使用PLAIN编码方式，PLAIN编码方式支持任意类型数据。
	• eventTimeColumn：事件时间列，用于指定按某一列数据的时间进行分区。如果不设置此参数，则按数据写入表格存储的时间进行分区。
	• parquetSchema：指定需要投递的数据列，必须手动配置投递字段的源表字段、目标字段和目标字段类型。
	您可以选择任意字段以任意顺序、名称写入列存文件，OSS的列存数据会按Schema数组中的数据列先后顺序分布。
	注意
	投递数据的字段类型必须与数据源的字段类型匹配，否则会作为脏数据丢弃。字段类型映射详情请参见数据格式映射。

taskType	投递任务的类型，包括如下选项： • INC：表示增量数据投递模式，只同步增量数据。 • BASE：表示全量数据投递模式，一次性全表扫描数据同步。 • BASE_INC（默认）：表示全量&增量数据投递模式，全量数据同步完成后，再同步增量数据。 其中增量数据同步时可以获取最新投递时间和了解当前投递状态。

示例

```
import com.alicloud.openservices.tablestore.ClientException;
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.TableStoreException;
import com.alicloud.openservices.tablestore.model.delivery.*;
public class DeliveryTask {

        public static void main(String[] args) {
            final String endPoint = "https://yourinstancename.cn-hangzhou.ots.aliyuncs.com"
;
```

```
            final String accessKeyId = "LT********************g5";

            final String accessKeySecret = "Er**************************Yc";

            final String instanceName = "yourinstancename";

            SyncClient client = new SyncClient(endPoint, accessKeyId, accessKeySecret, inst
anceName);
            try {
                createDeliveryTask(client);
                System.out.println("end");
            } catch (TableStoreException e) {
                System.err.println("操作失败，详情：" + e.getMessage() + e.getErrorCode() + e
.toString());
                System.err.println("Request ID:" + e.getRequestId());
            } catch (ClientException e) {
                System.err.println("请求失败，详情：" + e.getMessage());
            } finally {
                client.shutdown();
            }
        }

        private static void createDeliveryTask(SyncClient client){
            String tableName = "sampleTable";
            String taskName = "sampledeliverytask";
            OSSTaskConfig taskConfig = new OSSTaskConfig();
            taskConfig.setOssPrefix("sampledeliverytask/year=$yyyy/month=$MM");
            taskConfig.setOssBucket("datadeliverytest");
            taskConfig.setOssEndpoint("oss-cn-hangzhou.aliyuncs.com");
            taskConfig.setOssStsRole("acs:ram::17************45:role/aliyunserviceroleforot
sdatadelivery");
            //eventColumn为可选配置，指定按某一列数据的时间进行分区。如果不设置此参数，则按数据写入表格存储的时间进行分区。
            EventColumn eventColumn = new EventColumn("Col1", EventTimeFormat.RFC1123);
            taskConfig.setEventTimeColumn(eventColumn);
            taskConfig.addParquetSchema(new ParquetSchema("PK1", "PK1", DataType.UTF8));
            taskConfig.addParquetSchema(new ParquetSchema("PK2", "PK2", DataType.BOOL));
            taskConfig.addParquetSchema(new ParquetSchema("Col1", "Col1", DataType.UTF8));
            CreateDeliveryTaskRequest request = new CreateDeliveryTaskRequest();
            request.setTableName(tableName);
            request.setTaskName(taskName);
            request.setTaskConfig(taskConfig);
            request.setTaskType(DeliveryTaskType.BASE_INC);
            CreateDeliveryTaskResponse response = client.createDeliveryTask(request);
            System.out.println("resquestID: "+ response.getRequestId());
            System.out.println("traceID: " + response.getTraceId());
            System.out.println("create delivery task success");
        }
}
```

11.4. 数据湖计算分析

11.4.1. 使用DLA

配置数据湖投递任务后，表格存储的数据会持续投递到对应的OSS Bucket，当使用DLA执行SQL语句分析OSS中的数据前，需要使用元数据爬取或者手动创建指向OSS目录的外表。

前提条件

- 已开通云原生数据湖分析服务，详情请参见开通云原生数据湖分析服务。
- 已创建数据投递任务，详情请参见快速入门。

操作步骤

您可以通过向导式创建元数据爬取任务或者使用SQL手动创建方式使用DLA分析OSS中的数据，推荐使用向导式创建元数据爬取任务方式，请根据实际需要选择。

- （推荐）向导式创建元数据爬取任务
  i. 通过DLA的控制台向导创建OSS目录中的元数据爬取任务，详情请参见元数据爬取。

     元数据爬取任务可以在单次运行中自动为OSS中的数据文件创建和更新数据湖元数据（一张或多张表），具有自动探索文件数据字段及类型、自动映射目录和分区、自动感知新增列及分区、自动对文件进行分组建表的能力。

  ii. 查询数据，详情请参见执行SQL语句。
- 使用SQL手动创建
  i. 在**SQL执行**页面，使用SQL创建Schema和指向OSS数据目录的外表，详情请参见创建Schema和外表。

     > **说明**
     >
     > 您也可以通过向导式创建Schema或者直接选择已创建的Schema。

     创建指向OSS数据目录的外表时使用的SQL语句可以通过表格存储控制台获取，获取方法如下：

     在实例的**数据湖投递**页面，单击投递任务**操作**列的**建表语句**，可以查看和复制SQL语句，如下图所示。

ii. 同步OSS数据源中实际的数据分区信息到元数据中，详情请参见同步数据分区。

iii. 查询数据，详情请参见执行SQL语句。

11.4.2. 使用EMR

使用EMR的JindoFS缓存模式连接OSS数据湖。

背景信息

您可以使用EMR的JindoFS缓存模式或者JindoFS块模式连接OSS数据湖。

- 缓存模式（Cache）主要兼容原生OSS存储方式，文件以对象的形式存储在OSS上，每个文件根据实际访问情况会在本地进行缓存，提升EMR集群内访问OSS的效率，同时兼容了OSS原有文件形式，数据访问上能够与其他OSS客户端完全兼容。详情请参见JindoFS缓存模式使用说明。
- 块存储模式（Block）提供了最为高效的数据读写能力和元数据访问能力。数据以Block形式存储在后端存储OSS上，本地提供缓存加速，元数据则由本地Namespace服务维护，提供高效的元数据访问性能。详情请参见JindoFS块存储模式使用说明。

前提条件

- 已创建EMR集群，详情请参见创建集群。

  创建集群时，请注意如下事项：

  - 创建EMR集群和OSS属于同一个阿里云账号，且建议EMR集群和OSS Bucket处于同一地域。
  - 创建集群时，请打开**挂载公网**和**远程登录**开关，将集群挂载到公网，用于Shell远程登录服务器。
  - bigboot与smartdata为后续配置相关服务，如果默认未选中，请选中bigboot与smartdata服务。
- 已创建数据投递任务，详情请参见快速入门。

操作步骤

1. 使用EMR的jindoFS缓存模式连接OSS和启用缓存，详情请参见JindoFS缓存模式使用说明。

   启用缓存会利用本地磁盘对访问的热数据块进行缓存，默认状态为禁用，即所有OSS读取都直接访问OSS上的数据。缓存启用后，Jindo服务会自动管理本地缓存备份，通过水位清理本地缓存，请根据需求配置一定的比例用于缓存。

2. 启动spark SQL。

   i. 通过远程登录工具（例如PuTTY）登录EMR Header服务器。

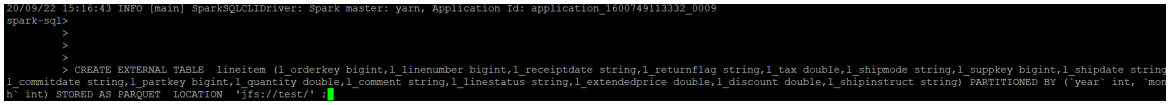
   ii. 执行如下命令运行Spark SQL。

   ```
   spark-sql --master yarn --num-executors 5 --executor-memory 1g --executor-cores 2
   ```

3. 使用SQL语句创建指向OSS数据目录的外表。

   请使用通过表格存储控制台获取的SQL语句，如下SQL语句示例仅供参考。

   ```
   CREATE EXTERNAL TABLE  lineitem (l_orderkey bigint,l_linenumber bigint,l_receiptdate st
   ring,l_returnflag string,l_tax double,l_shipmode string,l_suppkey bigint,l_shipdate str
   ing,l_commitdate string,l_partkey bigint,l_quantity double,l_comment string,l_linestatu
   s string,l_extendedprice double,l_discount double,l_shipinstruct string) PARTITIONED BY
   (`year` int, `month` int) STORED AS PARQUET  LOCATION  'jfs://test/' ;
   ```

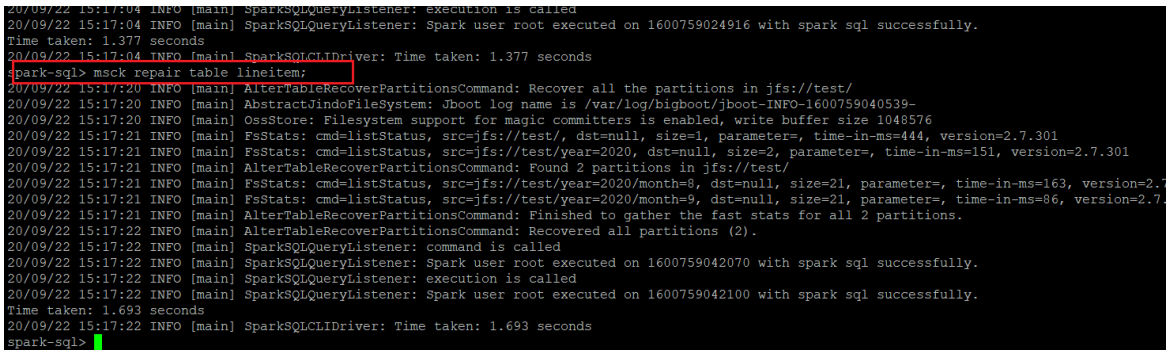
   通过表格存储控制台获取SQL语句的方法如下：

   在实例的**数据湖投递**页面，单击投递任务**操作**列的**建表语句**，可以查看和复制SQL语句，如下图所示。

4. 执行如下SQL语句，加载OSS数据源中实际的数据分区。

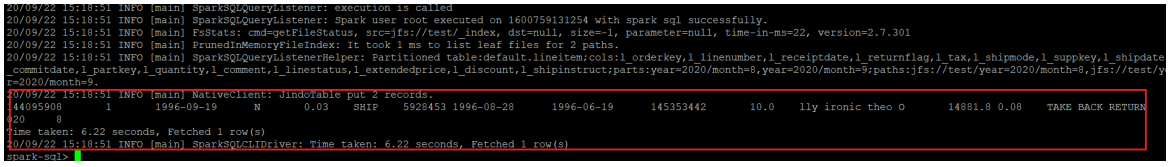
   其中lineitem为创建的外表名称。

```
msck repair table lineitem;
```



5. 查询数据。

```
select * from lineitem limit 1;
```



12.数据可视化

12.1. 数据可视化工具

本文介绍了表格存储支持对接的数据可视化工具。

工具	说明	文档链接
DataV	DataV数据可视化（简称DataV）可以将数据由单一的数字转化为各种动态的可视化图表。更多信息，请参见DataV数据可视化。 DataV可用于展示表格存储数据表或者二级索引表中的数据，一般用于构建复杂的大数据处理分析展现的企业应用系统。	对接DataV
Grafana	Grafana是一款开源的可视化和分析平台，支持Prometheus、Graphite、OpenTSDB、InfluxDB、Elasticsearch、MySQL、PostgreSQL等多种数据源的数据查询、可视化等。更多信息，请参见Grafana官方文档。 Grafana可用于展示表格存储数据表或者时序表中的数据。	对接Grafana

12.2. 对接Grafana

表格存储对接Grafana后，您可以通过Grafana可视化展示表格存储中的数据。

前提条件

- 首次使用表格存储时，请开通表格存储服务以及创建实例和数据表等。具体操作，请参见通过控制台使用或者通过命令行工具使用。
- 已为表格存储的数据表或者时序表创建映射关系。具体操作，请分别参见创建表的映射关系和创建多值模型映射关系。
- 首次使用时，请自行安装开源Grafana，且Grafana版本必须大于8.0.0。关于安装Grafana的具体操作，请参见Grafana官方文档。
- 已获取AccessKey（包括AccessKey ID和AccessKey Secret）。具体操作，请参见获取AccessKey。

背景信息

Grafana是一款开源的可视化和分析平台，支持Prometheus、Graphite、OpenTSDB、InfluxDB、Elasticsearch、MySQL、PostgreSQL等多种数据源的数据查询、可视化等。更多信息，请参见Grafana官方文档。

表格存储的表数据接入Grafana后，Grafana可以根据表数据生成大盘面板，将数据实时展示给需要的用户。

注意事项

目前表格存储支持使用Grafana实现数据可视化功能的地域有华东1（杭州）、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）和新加坡。

步骤一：安装表格存储插件

如果使用阿里云Grafana服务，请跳过此步骤。关于使用阿里云Grafana服务的具体操作，请参见添加并使用Tablestore数据源。

在Windows平台操作

1. 下载表格存储Grafana插件包。具体下载路径为表格存储Grafana插件包。
2. 解压表格存储Grafana插件包，并将表格存储Grafana插件包放到Grafana插件的*plugins-bundled*目录中。
3. 修改Grafana配置文件。
   i. 使用文本编辑器工具打开Grafana插件conf目录中的配置文件defaults.ini。
   ii. 在配置文件的**[plugins]**节点中，设置allow_loading_unsigned_plugins参数。

   ```
   allow_loading_unsigned_plugins = aliyun-tablestore-grafana-datasource
   ```
4. 在任务管理器中重启grafana-server.exe进程。

在Linux或者Mac平台操作

1. 执行以下命令下载表格存储Grafana插件包。

   ```
   wget https://tablestore-doc.oss-cn-hangzhou.aliyuncs.com/aliyun-tablestore-grafana-plugin/tablestore-grafana-plugin-1.0.0.zip
   ```
2. 将表格存储Grafana插件包解压到Grafana插件目录。

   根据Grafana的安装方式执行对应命令。
   - 使用YUM或RPM安装的Grafana（只适用于Linux平台）：**unzip tablestore-grafana-plugin-1.0.0.zip -d /var/lib/grafana/plugins**
   - 使用.tar.gz文件安装的Grafana：**unzip tablestore-grafana-plugin-1.0.0.zip -d {PATH_TO}/grafana-{VERSION}/data/plugins**
3. 修改Grafana配置文件。
   i. 进入文件目录打开配置文件。
      - 使用YUM或RPM安装的Grafana（只适用于Linux平台）：*/etc/grafana/grafana.ini*
      - 使用.tar.gz文件安装的Grafana：*{PATH_TO}/grafana-{VERSION}/conf/defaults.ini*

        其中 `{PATH_TO}/grafana-{VERSION}` 为Grafana的安装路径，VERSION为Grafana的版本号。
   ii. 在配置文件的**[plugins]**节点中，设置allow_loading_unsigned_plugins参数。

   ```
   allow_loading_unsigned_plugins = aliyun-tablestore-grafana-datasource
   ```
4. 重启Grafana。
   i. 使用**kill**命令终止Grafana进程。
   ii. 执行以下命令启动Grafana。
      - 使用YUM或RPM安装的Grafana（只适用于Linux平台）：**systemctl restart grafana-server**
      - 使用.tar.gz文件安装的Grafana：**./bin/grafana-server web**

步骤二：配置数据源

1. 登录Grafana。
   i. 在浏览器中输入 `http://localhost:3000/` ，进入Grafana登录界面。
   ii. 输入Email or username和Password，单击**Log in**。

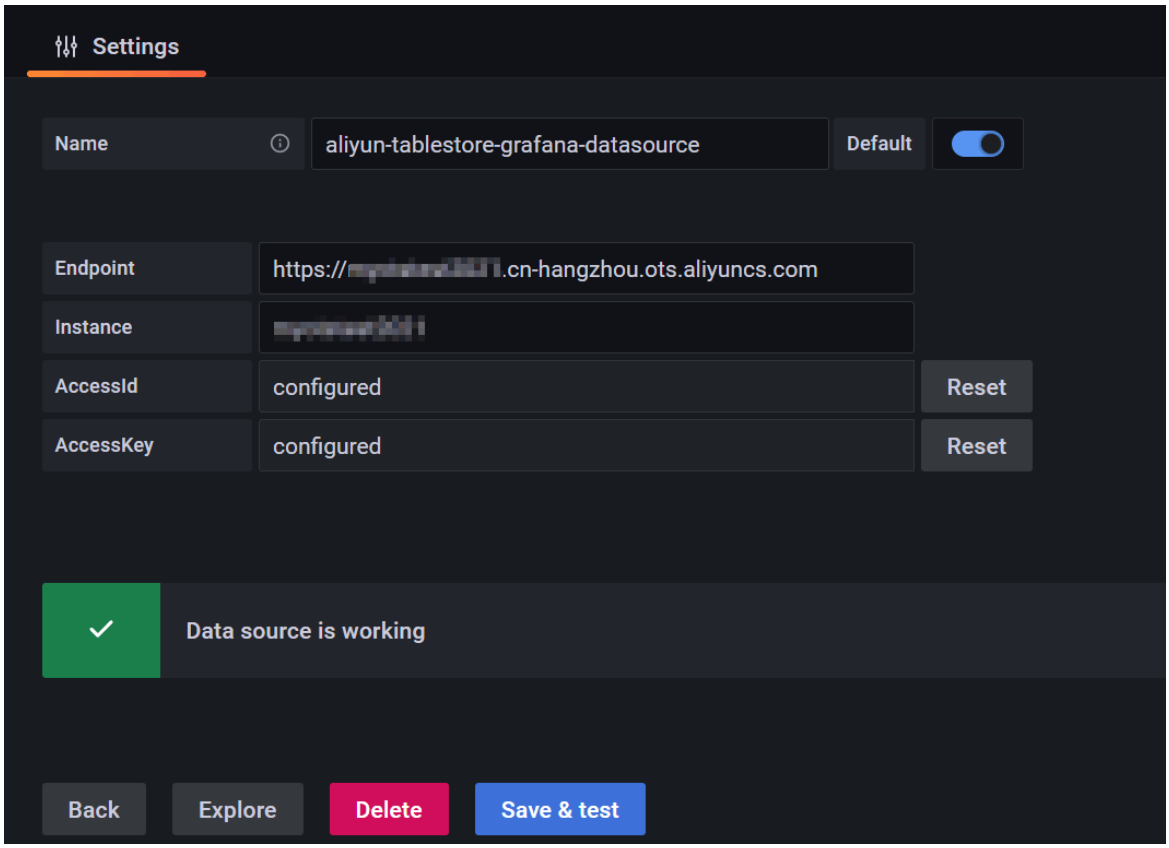
   Grafana默认初始登录用户名和密码均为admin。首次登录时，请根据系统提示修改初始密码。

2. 将鼠标移动到左侧导航栏的  图标上，单击Data sources。
3. 在Data sources页签，单击Add data source。
4. 在Add data source页面的Others区域，单击aliyun-tablestore-grafana-datasource。
5. 在Settings页面，根据下表说明配置相关参数。

参数	示例值	说明
Name	aliyun-tablestore-grafana-datasource	数据源名称，可自定义。默认为aliyun-tablestore-grafana-datasource。
Endpoint	https://myinstance.cn-hangzhou.ots.aliyuncs.com	表格存储实例的服务地址，请根据访问的表格存储实例填写。更多信息，请参见服务地址。
Instance	myinstance	Tablestore实例名称。
AccessId	**********	拥有表格存储访问权限的阿里云账号或者RAM用户的AccessKey ID。
AccessKey	**********	拥有表格存储访问权限的阿里云账号或者RAM用户的AccessKey Secret。

6. 单击Save & test。

   连接成功后，界面会显示Data source is working信息。



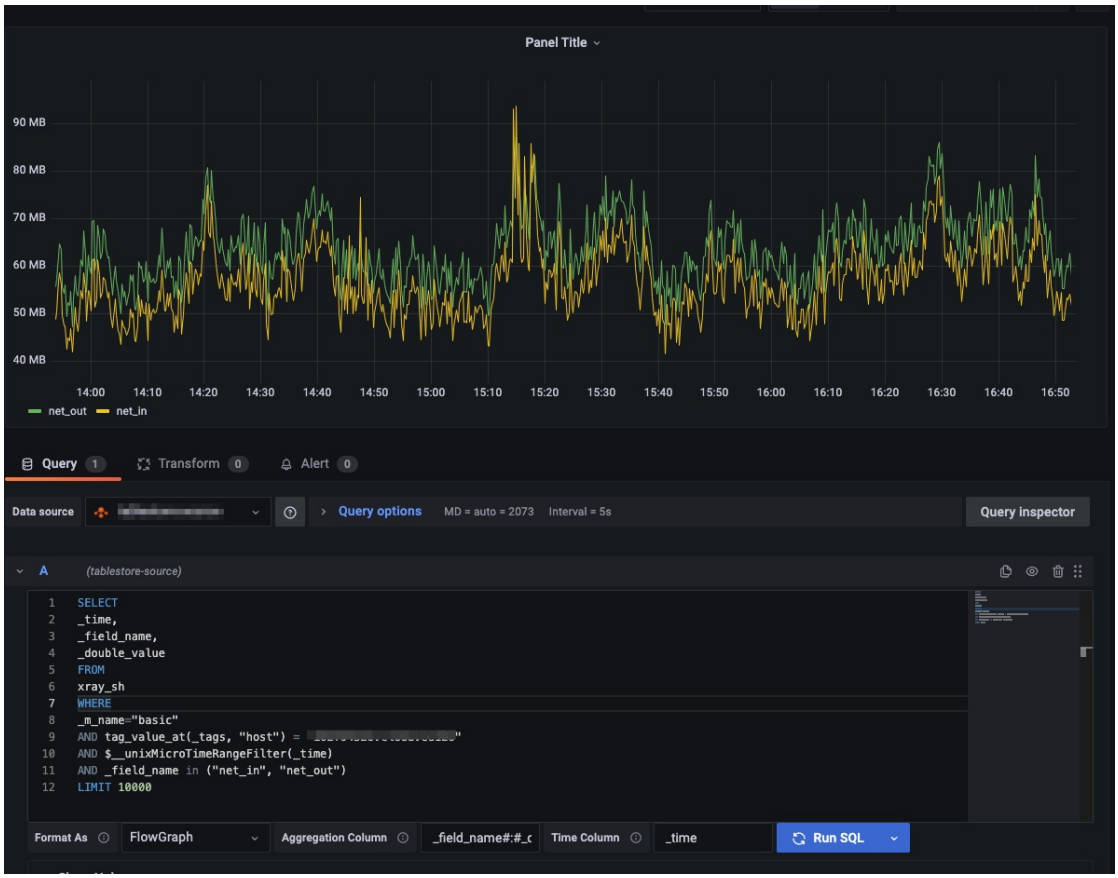
步骤三：创建大盘面板

1. 在Grafana控制台界面，将鼠标移动到左侧导航栏的  图标上，单击**Dashboard**。
2. 在**New dashboard**页面单击  图标。
3. 在**Add panel**区域，单击**Add a new panel**。
4. 在**Edit Panel**页面**Query**区域配置数据源查询条件。
   i. 在Data source下拉列表中选择tablestore数据源。
   ii. 配置数据源参数。

参数	示例	示例
Query	`SELECT * FROM your_table WHERE $__unixMicroTimeRangeFilter(_time)AND _m_name = "your_measurement" AND tag value_at(_tags, "your_tag")="your_tag_value"LIMIT 1000`	SQL查询语句。更多信息，请参见查询数据。 **注意** ■ 在WHERE子句中要通过预定义宏过滤时间范围，即示例中的 `$__unixMicroTimeRangeFilter` 。更多的时间宏函数请单击配置页面中的“Show Help”查看。 ■ 如果以时序图形式展示，则需要返回以数字时间戳形式表示的时间列，并配置时间列的列名。
Format As	Timeseries	结果处理形式。取值范围如下： ■ Timeseries（默认）：普通时序图。 ■ FlowGraph：多维图表展示。 ■ Table：普通表格形式。
Time Column	_time	返回数据中时间列的列名，时间列会作为时序图的横坐标。当选择Format As为**Timeseries**或者**FlowGraph**时可配置。
Aggregation Column	_field_name#:#_double_value	将同一时间点的多行单列数据转换为同一时间点的单行多列数据，适用于将Tablestore时序SQL产生的单值模型数据转换为多值模型数据。当选择Format As为**FlowGraph**时可配置。格式为 `<数据点名称列>#:#<数值列>` 。

5. 单击**Run SQL**，执行SQL语句查看数据和调试。
6. 设置并保存大盘面板。

i. 在右侧设置监控图表的名称、类型、展示样式等。



ii. 单击右上角的Apply。

iii. 单击右上角的图标，在Save dashboard as...对话框，设置Dashboard name和大盘归属的Folder后，单击Save。

步骤四：查看监控数据

1. 在Grafana控制台界面，将鼠标移动到左侧导航栏的图标上，单击Browse。
2. 在Browse页签，单击目标目录下的监控大盘，即可查看目标大盘上的所有监控图表。

12.3. 对接DataV

通过DataV控制台添加表格存储数据源后，您可以使用DataV可视化展现表格存储的数据。

前提条件

- 首次使用表格存储时，请开通表格存储服务以及创建实例和数据表等。具体操作，请参见通过控制台使用或者通过命令行工具使用。
- 首次使用DataV时，请开通DataV服务。具体操作，请参见开通DataV服务。
- 已获取AccessKey（包括AccessKey ID和AccessKey Secret）。具体操作，请参见获取AccessKey。

背景信息

DataV数据可视化（简称DataV）可以将数据由单一的数字转化为各种动态的可视化图表。更多信息，请参见DataV数据可视化。

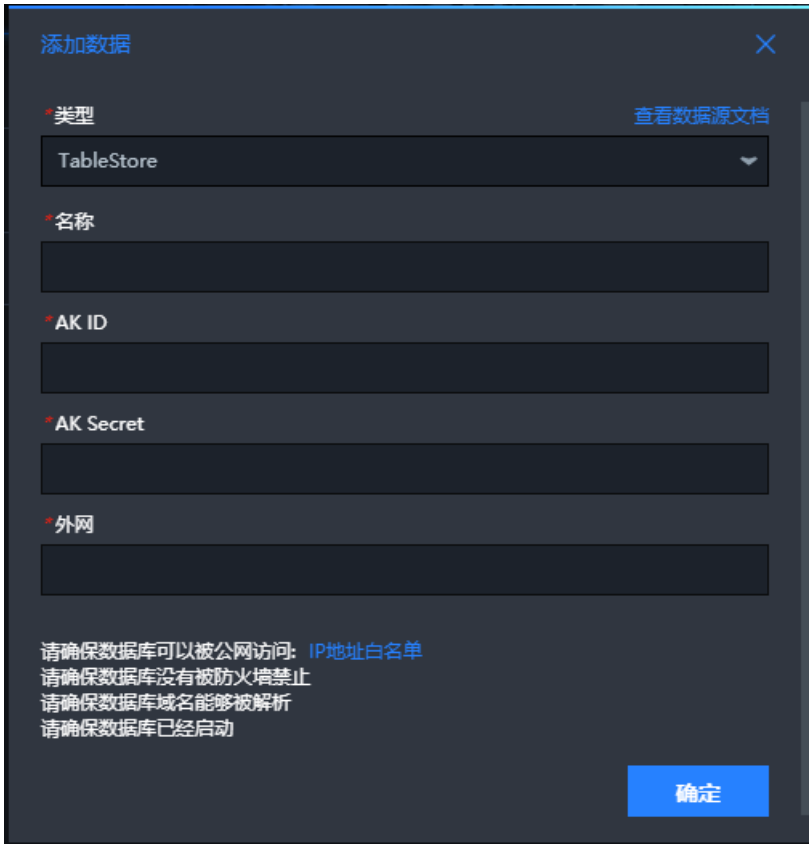
表格存储的表数据接入DataV后，DataV可以根据表数据生成数据看板，将数据实时展示给需要的用户。

注意事项

数据源只能是表格存储数据表或者二级索引表。

步骤一：添加表格存储数据源

1. 登录DataV控制台。
2. 在**我的数据**页签，单击**数据源管理**。
3. 在**数据源管理**页面，单击**添加数据**。



4. 在**添加数据**对话框，选择类型为TableStore并配置表格存储相关信息，详细参数说明请参见下表。

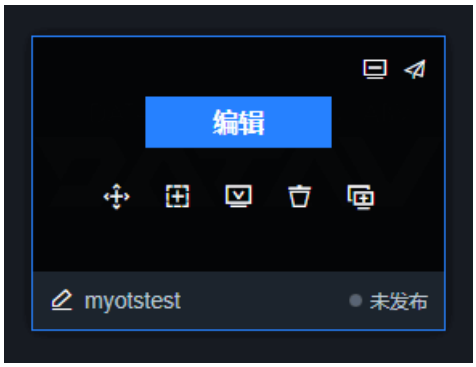
参数	说明
名称	数据源的显示名称。
AK ID	拥有表格存储访问权限的阿里云账号或者RAM用户的AccessKey ID。
AK Secret	拥有表格存储访问权限的阿里云账号或者RAM用户的AccessKey Secret。
外网	表格存储实例的服务地址，请根据访问的表格存储实例填写。更多信息，请参见服务地址。

5. 单击**确定**。

添加的数据源会自动显示在数据源列表中。

步骤二：配置表格存储数据源

1. 登录DataV控制台。
2. 根据实际场景创建可视化项目或者直接选择可视化项目进行操作。
   - 如果使用已有可视化项目，请将鼠标移动到需要使用的可视化项目上，单击**编辑**即可。

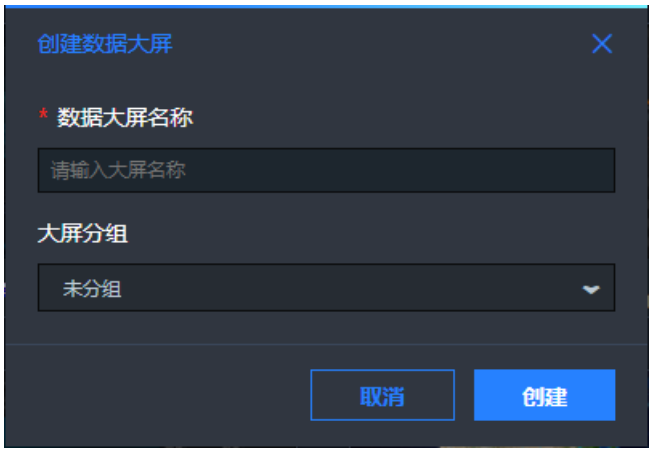


   - 如果是首次使用或者已有可视化项目不满足使用需求，请为可视化应用创建项目。

> **说明** 系统支持PC端创建、移动端创建以及识图创建三种方式，此处以PC端创建为例介绍。

     a. 在**我的可视化**页面，单击**PC端创建**。
     b. 将鼠标移动到需要使用的可视化应用上，单击**创建项目**。

        DataV支持基于模板或者使用空白画布制作可视化应用，请根据实际选择。
     c. 在**创建数据大屏**对话框，填写数据大屏名称以及选择大屏分组。
     d. 单击**创建**。



3. 配置表格存储数据源。
   i. 在画布编辑页面，单击画布中的某一组件。

> **说明** 如果画布中无组件，请先添加组件。具体操作，请参见添加资产。

   ii. 在画布右侧的组件配置面板中，单击[–]图标。

iii. 在**数据**页签，单击**配置数据源**。

iv. 在**设置数据源**面板，选择数据源类型为TableStore并选择已有数据源。

v. 选择对数据源的操作方式并填写查询语句。

系统支持getRow和getRange两种操作方式，分别对应于表格存储的GetRow和GetRange API。

- 当选择操作为getRow时，可读取指定主键的一行数据。查询语句的格式和参数说明如下：

```
{
"table_name": "test",
"rows": {
"id": 2
},
"columns": [
"id",
"test"
]
}
```

参数	说明
table_name	表格存储的表名称。
rows	行的主键。 **注意** 当表中存在多个主键列时，设置的主键列个数与数据类型必须和数据表的主键列个数与数据类型一致。
columns	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。

- 当选择操作为getRange时，可读取指定主键范围内的所有数据。查询语句的格式和参数说明如下：

```
{
"table_name": "test",
"direction": "FORWARD",
"columns": [
"id",
"test"
],
"range": {
"limit": 4,
"start": {
"id": "InfMin"
},
"end": {
"id": 3
}
}
}
```

参数	说明
table_name	表格存储的表名称。

参数	说明
direction	读取方向。 ■ 如果值为正序（FORWARD），则起始主键必须小于结束主键，返回的行按照主键值由小到大的顺序进行排列。 ■ 如果值为逆序（BACKWARD），则起始主键必须大于结束主键，返回的行按照主键值由大到小的顺序进行排列。 例如同一表中有两个主键值A和B，A<B。如正序读取[A, B)，则按从A至B的顺序返回主键值大于等于A、小于B的行；逆序读取[B, A)，则按从B至A的顺序返回大于A、小于等于B的数据。
columns	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 如果某行数据的主键属于读取范围，但是该行数据不包含指定返回的列，那么返回结果中不包含该行数据。
limit	数据的最大返回行数，此值必须大于0。 表格存储按照正序或者逆序返回指定的最大返回行数后即结束该操作的执行，即使该区间内仍有未返回的数据。
start	本次范围读取的起始主键和结束主键，起始主键和结束主键需要是有效的主键或者是由InfMin和InfMax类型组成的虚拟点，虚拟点的列数必须与主键相同。 其中InfMin表示无限小，任何类型的值都比它大；InfMax表示无限大，任何类型的值都比它小。 **注意** 当表中存在多个主键列时，设置的主键列个数与数据类型必须和数据表的主键列个数与数据类型一致。 ■ start表示起始主键，如果该行存在，则返回结果中一定会包含此行。 ■ end表示结束主键，无论该行是否存在，返回结果中都不会包含此行。
end	

vi. 单击**数据响应结果**后的图标，获取数据响应结果。

> **说明** 获取数据响应结果后，再次查询数据时可以直接单击**预览数据源返回结果**查看数据返回结果。

4. 预览和发布可视化应用。

i. 单击画布编辑器右上角的图标，预览可视化应用。

ii. 单击画布编辑器右上角的图标。

iii. 在**发布**对话框，单击**发布大屏**。

iv. 在**发布成功**对话框，单击**取消**。

> **说明** 发布成功后，发布页内容已锁定，每次编辑后可通过“快照管理”快速同步发布内容。您也可以单击**前往快照管理**，查看已创建的快照信息。

v. 单击**分享链接**对应链接右侧的图标，复制链接。

vi. 打开浏览器，将复制的链接粘贴到地址栏中，即可在线观看发布成功的可视化应用。

13.监控与报警

13.1. 概述

通过查看表格存储资源的监控信息，您可以了解资源的使用情况。通过为资源的重要监控指标设置报警规则，您还可以在第一时间得知指标异常并快速处理异常。本文介绍了支持的监控项以及支持为监控指标设置的报警规则信息。

背景信息

云监控（CloudMonitor）是一项针对阿里云资源和互联网应用进行监控的服务。云监控可用于监控各云服务资源的监控指标，并针对指定监控指标设置报警。使您全面了解阿里云上资源的使用情况和业务运行状况，并及时对故障资源进行处理，保证业务正常运行。更多信息，请参见什么是云监控。

监控项说明

通过云监控控制台支持对表格存储的实例进行数据监控，通过云监控API支持对表格存储的实例或者表进行数据监控，通过表格存储控制台支持对表格存储实例、表、多元索引进行数据监控。

通过云监控控制台操作

资源类型	监控项	说明
实例	VCU	预留模式下每分钟VCU的使用情况。支持按照平均值、最大值、最小值和求和值查看实例VCU个数。 监控指标包括已使用的VCU、正在使用VCU和实例预留VCU个数。
	InstanceCU	按量模式下每分钟平均消耗的CU个数。 监控指标包括内部读CU、内部写CU、实例按量读CU、实例按量写CU、操作读CU和操作写CU。
	RequestCount	每分钟的平均请求次数。 监控指标包括总请求、2xx请求、429请求、4xx请求和5xx请求。
	InstanceNet	每分钟平均使用的流量。 监控指标包括外网流入流量、外网流出流量、免费外网流出流量、内网流入流量和内网流出流量。
	RowCount	每分钟平均操作的行数统计。 监控指标包括总行数和失败行数。

通过云监控SDK操作

资源类型	监控项	说明

资源类型	监控项	说明
实例	InstanceCount2xxNumber	2xx请求次数。单位为countSecond。
	InstanceCount429Number	429请求次数。单位为countSecond。
	InstanceCount4xxNumber	4xx请求次数。单位为countSecond。
	InstanceCount5xxNumber	5xx请求次数。单位为countSecond。
	InstanceElasticVCU	预留模式下实例弹性VCU。单位为countSecond。
	InstanceExtranetIn	外网流入流量。单位为Bytes/s。
	InstanceExtranetOut	外网流出流量。单位为Bytes/s。
	InstanceFailRowCount	操作失败行数。单位为Row/s。
	InstanceFreeExtranetOut	免费外网流出流量。单位为Bytes/s。
	InstanceFreeReadCU	内部读CU。单位为CU/s。
	InstanceFreeWriteCU	内部写CU。单位为CU/s。
	InstanceIntranetIn	内网流入流量。单位为Bytes/s。
	InstanceIntranetOut	内网流出流量。单位为Bytes/s。
	InstanceOverReadCU	实例超量读CU。单位为CU/s。
	InstanceOverWriteCU	实例超量写CU。单位为CU/s。
	InstanceReadCU	实例操作读CU。单位为CU/s。
	InstanceRequestNumber	总请求次数。单位为countSecond。
	InstanceReservedVCU	预留模式下实例的预留VCU个数。单位为countSecond。
	InstanceRowCount	操作总行数统计。单位为Row/s。
	InstanceVCU	实例VCU。单位为countSecond。
	AverageLatency	操作平均延时。单位为微秒（us）。
	Count2xxNumber	2xx请求次数。单位为countSecond。
	Count429Number	429请求次数。单位为countSecond。
	Count4xxNumber	4xx请求次数。单位为countSecond。

资源类型	监控项	说明
表（包括数据表和二级索引表）	Count5xxNumber	5xx请求次数。单位为countSecond。
	ExtranetIn	外网流入流量。单位为Bytes/s。
	ExtranetOut	外网流出流量。单位为Bytes/s。
	FailRowCount	操作失败行数。单位为Row/s。
	FreeExtranetOut	免费外网流出流量。单位为Bytes/s。
	FreeReadCU	内部读CU。单位为CU/s。
	FreeWriteCU	内部写CU。单位为CU/s。
	IntranetIn	内网流入流量。单位为Bytes/s。
	IntranetOut	内网流出流量。单位为Bytes/s。
	OverReadCU	表超量读CU。单位为CU/s。
	OverWriteCU	表超量写CU。单位为CU/s。
	ReadCU	操作读CU个数。单位为CU/s。
	RequestNumber	总请求。单位为countSecond。
	RowCount	操作总行数。单位为Row/s。
	WriteCU	操作写CU。单位为CU/s。

通过表格存储控制台操作

资源类型	监控项	说明
实例、表（包括数据表和二级索引表）、多元索引	每秒请求次数	每分钟平均的请求次数。 监控指标包括总QPS、2xx QPS、4xx QPS、5xx QPS和429 QPS。
	行数统计	每分钟平均操作的行数。 监控指标包括总行数和失败行数。
	流量统计	每分钟平均使用的流量。 监控指标包括外网流入、外网流出、内网流入、内网流出和免费外网流出。
	CapacityUnit	每分钟平均消耗的CU个数。 监控指标包括读CU、写CU、内部读CU和内部写CU。

资源类型	监控项	说明
	请求状态统计	2xx、4xx、429、5xx等请求类型的统计值以及百分比信息。
表、多元索引	平均访问延迟	每分钟的平均访问延迟。 监控指标包括请求延时。
表	表大小	每分钟的数据量大小。
多元索引	存储	多元索引存储的数据量大小。单位为字节。
	行数	多元索引同步数据表的行数。
	预留读CU	多元索引的预留CU个数。

报警规则说明

通过云监控控制台，您可以为不同监控指标设置报警规则。当资源的监控指标达到报警条件时，云监控会自动发送报警通知。下表列出了报警规则的报警级别、通知机制以及报警条件信息。

报警等级	通知机制	报警条件
紧急Critical	电话+短信+邮件+钉钉机器人	连续N个周期监控指标的平均值与指定阈值满足所设置的判断条件。 其中N值请根据报警等级进行区分设置。 **说明** 当所选的指标类型不同时，报警条件会存在差异，请以实际界面为准。
警告Warn	短信+邮件+钉钉机器人	
普通Info	邮件+钉钉机器人	

13.2. 通过表格存储控制台查看监控数据

通过表格存储控制台，您可以查看实例、数据表、二级索引表或者多元索引的监控数据。

注意事项

- 不同资源类型支持的监控项不同，请以实际为准。关于监控项的更多信息，请参见监控项说明。
- 监控数据统计可能会存在一定延迟。

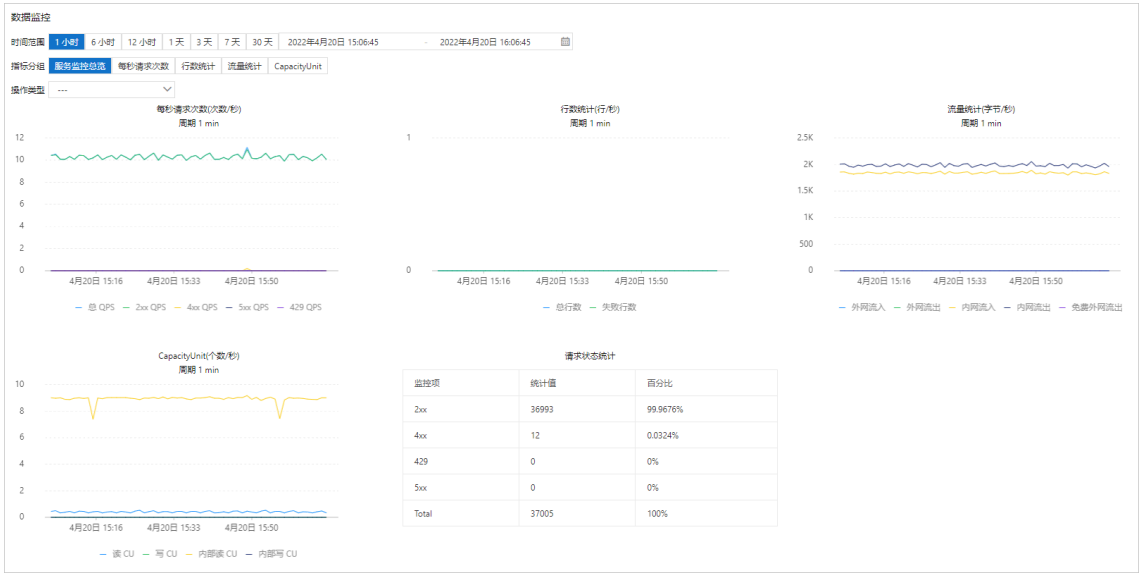
查看实例监控数据

通过表格存储控制台，您可以查看实例的每秒请求次数、行数统计、流量统计、CapacityUnit等监控数据。

1. 进入**实例管理**页面。
   i. 登录表格存储控制台。
   ii. 在页面上方，选择地域。
   iii. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
2. 在**实例管理**页面，单击**实例监控**页签后，设置时间范围。

您可以直接单击选择预设时间段或者自行选择所需日期范围。日期范围不能超过30天。

3. 选择要查看的监控数据。
   - 指标分组默认为**服务监控总览**，您可选择操作类型查看所有监控分组的监控数据。



   - 如果要根据监控分组查看所有操作类型的监控数据，请设置指标分组为所需的指标类型（例如每秒请求次数等）。此处以每秒请求次数为例介绍。

查看表或者索引监控数据

通过表格存储控制台，您可以查看数据表、二级索引表或者多元索引的平均访问延迟、每秒请求次数等监控数据。

> **注意**
> - 多元素引不支持表大小指标分组。
> - 不同资源的操作类型不同，请以实际界面为准。

1. 进入**实例管理**页面。
   i. 登录表格存储控制台。

ii. 在页面上方，选择地域。

iii. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。

2. 在**实例详情**页签的**数据表列表**区域，单击目标数据表监控列的 图标。

3. 在**指标监控**页签，选择数据表、二级索引表或者多元索引。

4. 设置时间范围。

   您可以直接单击选择预设时间段或者自行选择所需日期范围。日期范围不能超过30天。

5. 选择要查看的监控数据。

   - 指标分组默认为**服务监控总览**，您可选择操作类型查看所有监控分组的监控数据。

   - 如果要根据监控分组查看所有操作类型的监控数据，请设置指标分组为所需的指标类型（例如平均访问延迟等）。此处以平均访问延迟为例介绍。

查看多元索引计量数据

通过表格存储控制台，您可以查看多元索引的存储量、预留读CU、行数等计量数据。

1. 进入**实例管理**页面。
   i. 登录表格存储控制台。
   ii. 在页面上方，选择地域。
   iii. 在**概览**页面，单击实例名称或在操作列单击**实例管理**。
2. 在**实例详情**页签的**数据表列表**区域，单击目标数据表操作列的**索引管理**。
3. 在**索引管理**页签，单击目标多元索引操作列的**索引详情**。
4. 在**索引详情**对话框的**索引计量**区域，查看多元索引的存储大小、行数、预留读CU等信息。

13.3. 通过云监控控制台与SDK查看监控数据

表格存储通过云监控为您提供系统基本运行状态、性能以及计量等方面的监控数据指标，帮助您跟踪请求、分析使用情况、统计业务趋势，及时发现以及诊断系统的相关问题。本文介绍使用云监控服务提供的控制台或者DescribeMetricList接口来查询表格存储监控数据。

注意事项

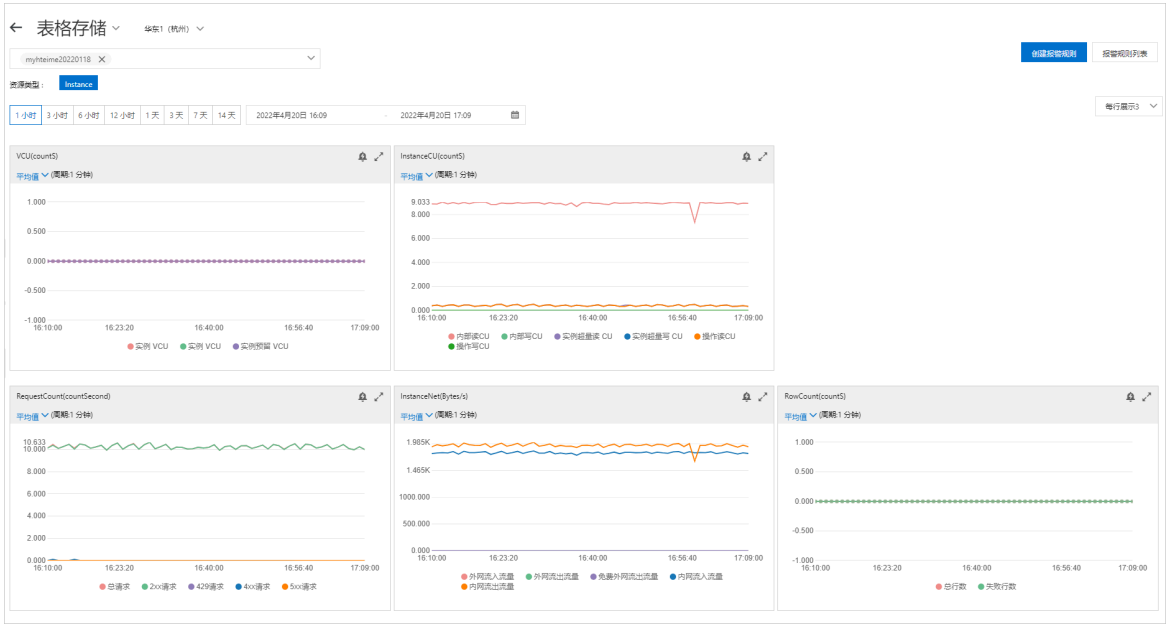
- 不同资源类型支持的监控项不同，请以实际为准。关于监控项的更多信息，请参见监控项说明。
- 监控数据统计可能会存在一定延迟。

通过云监控控制台查看

云监控会自动获取当前阿里云账号下所有云产品的资源。通过云监控控制台，您可以查看表格存储实例的VCU、InstanceCU、RequestCount、InstanceNet等监控数据。

1. 登录云监控控制台。
2. 单击**云产品监控**页签，然后选择**存储与CDN > 表格存储**。
3. 在**表格存储**页面，选择地域后，单击目标实例名称或在操作列单击**监控图表**。
4. 设置时间范围。

您可以直接单击选择预设时间段或者自行选择所需日期范围。日期范围不能超过30天。



通过云监控SDK查看

通过云监控SDK，您可以查看表格存储实例的InstanceCU、InstanceReadCU、InstanceCount2xxNumber等监控数据以及数据表或者二级索引表的AverageLatency、Count2xxNumber、ReadCU等监控数据。

> **说明** 云监控服务SDK示例，请参见Java SDK使用手册。关于DescribeMetricList接口的更多信息，请参见DescribeMetricList。

Namespace

Namespace用于指定监控的云服务。表格存储监控服务使用的Namespace为**acs_ots_new**。

通过Java SDK指定监控表格存储服务的示例代码如下：

```
DescribeMetricListRequest request = new DescribeMetricListRequest();
request.setNamespace("acs_ots_new");
```

StartTime和EndTime

StartTime和EndTime用于指定查询监控数据的时间范围。云监控的时间参数取值范围采用左开右闭的形式**(StartTime, EndTime]**，即可以查询StartTime到EndTime之间的数据（包含EndTime的数据）。

> **注意** StartTime和EndTime的时间间隔不能大于31天，且无法查询31天以前的数据。

通过Java SDK指定查询监控数据时间范围的示例代码如下：

```
//设置监控数据的结束时间。
request.setEndTime("2022-06-13 11:23:00");
//设置监控数据的开始时间。
request.setStartTime("2022-06-13 10:23:00");
```

您也可以使用毫秒单位的时间戳形式指定时间范围。示例代码如下：

```
//设置监控数据结束时间的时间戳，单位为毫秒。
request.setEndTime("1655090580000");
//设置监控数据开始时间的时间戳，单位为毫秒。
request.setStartTime("1655086860000");
```

Dimensions

Dimensions用于指定待查询的实例或者表。Dimensions传入时需要使用JSON字符串表示，例如 `{"userId":"1234567890****","region":"cn-hangzhou","instanceName":"myinstance","tableName":"mytable"}` 。

通过Java SDK查询实例的示例代码如下：

```
//填写待查询数据的实例名称。
request.setDimensions("{\"userId\":\"1234567890****\",\"region\":\"cn-hangzhou\",\"instanceName\":\"myinstance\"}");
```

如果要查询表级别的监控指标，则需要配置tableName参数。通过Java SDK查询表的示例代码如下：

```
//填写待查询数据的表名称。
request.setDimensions("{\"userId\":\"1234567890****\",\"region\":\"cn-hangzhou\",\"instanceName\":\"myinstance\",\"tableName\":\"mytable\"}");
```

如果要查询实例级别或者表表级别指定操作的监控指标，则需要配置operation参数。通过Java SDK查询表中PutRow操作的示例代码如下：

> **说明** operation参数的取值为API名称。更多信息，请参见API参考。

```
//填写待查询数据的表和操作名称。
request.setDimensions("{\"userId\":\"1234567890****\",\"region\":\"cn-hangzhou\",\"instanceName\":\"myinstance\",\"tableName\":\"mytable\",\"operation\":\"PutRow\"}");
```

Period

Period用于指定指标项的查询周期。监控的计量类指标查询周期为3600s，其他所有指标的查询周期均为60s。各指标项的说明，请参见监控项说明。

Metric

Metric用于指定查询的指标。

通过Java SDK设置指标名称的示例代码如下：

```
//设置Metric名称，此处以InstanceCount2xxNumber监控指标为例介绍。
request.setMetric("InstanceCount2xxNumber");
```

各监控项的名称请参见下表。

资源类型	Metric	Dimensions	说明

资源类型	Metric	Dimensions	说明
实例	InstanceCount2xxNumber	userId、region、instanceName、operation	2xx请求次数。单位为countSecond。
	InstanceCount429Number	userId、region、instanceName、operation	429请求次数。单位为countSecond。
	InstanceCount4xxNumber	userId、region、instanceName、operation	4xx请求次数。单位为countSecond。
	InstanceCount5xxNumber	userId、region、instanceName、operation	5xx请求次数。单位为countSecond。
	InstanceElasticVCU	userId、region、instanceName	预留模式下实例弹性VCU。单位为countSecond。
	InstanceExtranetIn	userId、region、instanceName、operation	外网流入流量。单位为Bytes/s。
	InstanceExtranetOut	userId、region、instanceName、operation	外网流出流量。单位为Bytes/s。
	InstanceFailRowCount	userId、region、instanceName、operation	操作失败行数。单位为Row/s。
	InstanceFreeExtranetOut	userId、region、instanceName、operation	免费外网流出流量。单位为Bytes/s。
	InstanceFreeReadCU	userId、region、instanceName、operation	内部读CU。单位为CU/s。
	InstanceFreeWriteCU	userId、region、instanceName、operation	内部写CU。单位为CU/s。
	InstanceIntranetIn	userId、region、instanceName、operation	内网流入流量。单位为Bytes/s。
	InstanceIntranetOut	userId、region、instanceName、operation	内网流出流量。单位为Bytes/s。

资源类型	Metric	Dimensions	说明
	InstanceOverReadCU	userId、region、instanceName、operation	实例超量读CU。单位为CU/s。
	InstanceOverWriteCU	userId、region、instanceName、operation	实例超量写CU。单位为CU/s。
	InstanceReadCU	userId、region、instanceName、operation	实例操作读CU。单位为CU/s。
	InstanceRequestNumber	userId、region、instanceName、operation	总请求次数。单位为countSecond。
	InstanceReservedVCU	userId、region、instanceName	预留模式下实例的预留VCU个数。单位为countSecond。
	InstanceRowCount	userId、region、instanceName、operation	操作总行数统计。单位为Row/s。
	InstanceVCU	userId、region、instanceName	实例VCU。单位为countSecond。
	AverageLatency	userId、region、instanceName、tableName、operation	操作平均延时。单位为微秒（us）。
	Count2xxNumber	userId、region、instanceName、tableName、operation	2xx请求次数。单位为countSecond。
	Count429Number	userId、region、instanceName、tableName、operation	429请求次数。单位为countSecond。
	Count4xxNumber	userId、region、instanceName、tableName、operation	4xx请求次数。单位为countSecond。
	Count5xxNumber	userId、region、instanceName、tableName、operation	5xx请求次数。单位为countSecond。

资源类型	Metric	Dimensions	说明
表（包括数据表和二级索引表）	ExtranetIn	userId、region、instanceName、tableName、operation	外网流入流量。单位为Bytes/s。
	ExtranetOut	userId、region、instanceName、tableName、operation	外网流出流量。单位为Bytes/s。
	FailRowCount	userId、region、instanceName、tableName、operation	操作失败行数。单位为Row/s。
	FreeExtranetOut	userId、region、instanceName、tableName、operation	免费外网流出流量。单位为Bytes/s。
	FreeReadCU	userId、region、instanceName、tableName、operation	内部读CU。单位为CU/s。
	FreeWriteCU	userId、region、instanceName、tableName、operation	内部写CU。单位为CU/s。
	IntranetIn	userId、region、instanceName、tableName、operation	内网流入流量。单位为Bytes/s。
	IntranetOut	userId、region、instanceName、tableName、operation	内网流出流量。单位为Bytes/s。
	OverReadCU	userId、region、instanceName、tableName、operation	表超量读CU。单位为CU/s。
	OverWriteCU	userId、region、instanceName、tableName、operation	表超量写CU。单位为CU/s。

资源类型	Metric	Dimensions	说明
	ReadCU	userId、region、instanceName、tableName、operation	操作读CU个数。单位为CU/s。
	RequestNumber	userId、region、instanceName、tableName、operation	总请求。单位为countSecond。
	RowCount	userId、region、instanceName、tableName、operation	操作总行数。单位为Row/s。
	WriteCU	userId、region、instanceName、tableName、operation	操作写CU。单位为CU/s。

13.4. 配置监控指标报警

通过云监控控制台为监控指标配置报警规则后，如果资源的监控指标达到报警条件，则云监控会自动发送报警通知提醒您关注异常监控数据，便于您及时采取措施处理异常。

前提条件

已创建报警联系人组。具体操作，请参见创建报警联系组。

批量创建报警规则

通过将报警模板应用到已添加产品资源的应用分组，您可以批量为应用分组内的表格存储实例创建报警规则。

步骤一：创建应用分组并添加产品资源

应用分组提供跨云产品、跨地域的资源分组管理功能。您可以根据业务管理需求创建应用分组，将业务相关的资源添加到同一应用分组中，从应用分组维度管理报警规则。

1. 登录云监控控制台。
2. 手动创建应用分组。
   i. 在左侧导航栏，单击**应用分组**。
   ii. 在**应用分组**页签，单击  图标或右上角的**创建应用分组**。
   iii. 在**创建应用分组**面板，选择**创建实例方法**为**手动创建**，并设置应用分组名称和报警联系人组，其他i保持默认即可。

   订阅事件通知默认为打开状态。当应用分组中的资源产生严重或警告级别的事件时，云监控会自动给报警联系人组中的报警联系人发送报警通知。

iv. 单击**确定**。

3. 为应用分组添加产品资源。
   i. 在目标应用分组的**组内资源**页面，单击**管理产品和资源**。
   ii. 在**添加/修改组资源**面板，选择目标应用分组的关联云产品为**表格存储**，并选择表格存储实例。
   iii. 单击**确定**。

   在目标应用分组的**组内资源**页面，您可以查看添加的云产品及其实例信息。

4. 在目标应用分组的**组内资源**页面，单击←图标，返回**应用分组列表**页面。

在**应用分组**页签，您可以查看创建的应用分组。

步骤二：创建报警模板并应用到分组

1. 登录云监控控制台。
2. 在左侧导航栏，选择**报警服务 > 报警模板**。
3. 在**报警模板**页面，单击**创建报警模板**。
4. 在**创建/修改报警模板**面板，配置报警模板信息。
   i. 输入模板名称，并选择云产品为**表格存储**。
   ii. 单击**添加规则**，根据下表说明为报警模板设置阈值报警规则，然后单击**确定**。

   表格存储只支持添加阈值报警规则。您可以根据需要添加多个规则。

参数	说明
规则名称	阈值报警规则的名称。
指标类型	阈值报警规则的指标类型。取值： ■ **单指标**：一条报警规则仅作用于一个监控指标。 ■ **多指标**：一条报警规则作用于多个监控指标。
指标名称	报警的监控指标名称。关于如何获取云产品的监控项，请参见云产品监控项。 **说明** 当**指标类型**选择**单指标**时，显示该参数。
阈值及报警级别	报警规则的报警条件、报警阈值和报警级别。 您可以设置多级报警，当阈值处于不同区间时，对应不同报警级别，云监控通过不同渠道给您发送报警通知。 **说明** 当**指标类型**选择**单指标**时，显示该参数。

参数	说明
报警级别	报警级别和报警通知方式。取值： ■ 紧急（Critical）电话+短信+邮件+钉钉机器人 ■ 警告（Warn）短信+邮件+钉钉机器人 ■ 普通（Info）邮件+钉钉机器人 **说明** 当**指标类型**选择**多指标**时，显示该参数。
指标类型	多个监控指标的指标类型。取值： ■ **标准创建**：直接选择多个监控指标及其报警条件。 ■ **表达式创建**：通过表达式设置多个监控指标及其报警条件。 **说明** 当**指标类型**选择**多指标**时，显示该参数。
多指标报警描述	多个监控指标的报警规则。 **说明** 当**指标类型**选择**多指标**，且为**标准创建**时，显示该参数。
监控图表预览	监控指标的监控图表预览效果。 **监控图表预览**开关默认关闭。 **说明** 当**指标类型**选择**多指标**，且为**标准创建**时，显示该参数。
多指标关系	多个监控指标之间的关系。取值： ■ **当所有指标都符合条件时候则报警** ■ **有一个满足条件就报警** **说明** 当**指标类型**选择**多指标**，且为**标准创建**时，显示该参数。
多指标报警表达式	多个监控指标的报警表达式。 **说明** 当**指标类型**选择**多指标**，且为**表达式创建**时，显示该参数。
发出报警需要满足达到阈值的次数	发送报警通知需要监控指标达到报警阈值的次数。取值：连续1个周期、连续3个周期、连续5个周期、连续10个周期、连续15个周期、连续30个周期、连续60个周期、连续70个周期、连续90个周期、连续120个周期和连续180个周期。

参数	说明
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**

iii. 单击**确定**。

5. 应用报警模板到分组。

i. 在**创建/修改报警模板完成**对话框，单击**确定**。

> **说明** 如果您单击**取消**，则取消将当前报警模板应用到应用分组的操作。关于如何将报警模板应用到应用分组，请参见应用报警模板到应用分组。

ii. 在**应用模板到分组**对话框，设置应用分组、通道沉默周期、生效时间、报警回调和模板应用方式，然后单击**确定**。

iii. 在**应用模板到分组**对话框，单击**确定**。

单个创建报警规则

当需要监控表格存储实例的使用情况时，您可以为单个表格存储实例创建报警规则。

1. 登录云监控控制台。
2. 在左侧导航栏，选择**报警服务 > 报警规则**。
3. 在**报警规则列表**页面，单击**创建报警规则**。
4. 在**创建报警规则**面板，根据下表说明设置报警规则相关参数，然后单击**确定**。

参数	说明
产品	云监控可管理的云产品名称。例如：云数据库RDS版。
资源范围	报警规则作用的资源范围。取值： ○ **全部资源**：报警规则作用于指定云产品的全部资源上。 ○ **应用分组**：报警规则作用于指定云产品的指定应用分组内的全部资源上。 ○ **实例**：报警规则作用于指定云产品的指定资源上。
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： i. 单击**添加规则**。 ii. 在**添加规则描述**面板，设置规则名称、监控指标类型、监控指标、阈值、报警级别和报警方式等。 iii. 单击**确定**。

参数	说明
通道沉默周期	报警发生后未恢复正常，间隔多久重复发送一次报警通知。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。报警联系人组是一组报警联系人，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。
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	如果您打开**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，请参见创建伸缩规则。
日志服务	如果您打开**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	如果您打开**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**

14.备份与恢复

14.1. 概述

通过混合云备份HBR（Hybrid Backup Recovery），您可以定期备份表格存储（Tablestore）实例中的数据，并在数据丢失或受损时及时恢复。HBR支持全量与增量数据备份，同时支持数据冗余机制，可以提高存储库的数据可靠性。

公测说明

使用HBR备份与恢复Tablestore数据功能当前正在华东2（上海）、华东1（杭州）和华北2（北京）地域开启公测。

如果使用过程中遇到问题，请通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

背景信息

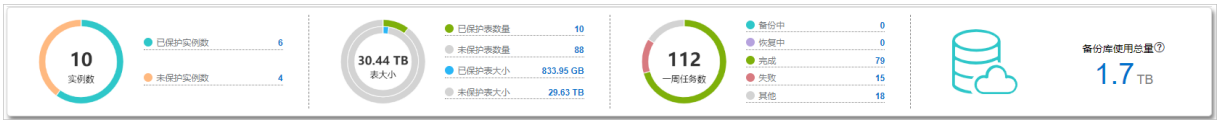
混合云备份HBR作为阿里云统一灾备平台，是一种简单易用、敏捷高效、安全可靠的公共云数据管理服务，可以为阿里云ECS整机、ECS数据库、文件系统、NAS、OSS、Tablestore以及自建机房内的文件、数据库、虚拟机、大规模NAS等提供备份、容灾保护以及策略化归档管理。更多信息，请参见什么是混合云备份HBR。

准备工作

- 已开通HBR服务。
- 已存在要备份的表格存储实例和数据。

备份情况总览

您可以通过备份大盘了解您名下表格存储资源的备份情况，包括已保护实例数、未保护实例数、已保护表数量及其大小、备份任务数等。



项目	说明
实例数	记录已备份和未备份的表格存储实例数量。
表大小	记录已备份和未备份的表格存储表数量和大小。
一周任务数	记录备份中、恢复中、完成和失败等的一周任务总数。
备份库使用总量	记录备份表格存储所使用的备份库总容量。HBR按照备份库存储容量计费。更多信息，请参见计费方式与计费项。

14.2. 备份Tablestore数据

本文介绍了通过混合云备份HBR（Hybrid Backup Recovery）如何定期备份表格存储（Tablestore）的全量数据或者增量数据。

注意事项

- 首次使用该功能，HBR将自动创建服务关联角色AliyunServiceRoleForHbrOtsBackup，用于获取您名下的Tablestore实例资源。请按照向导完成操作。更多信息，请参见HBR服务关联角色。
- HBR默认会读取Tablestore对应地域的实例，并自动加载，无需安装客户端。
- 只支持备份Tablestore数据到同一地域的HBR备份库中，不支持跨地域备份数据。
- 只支持备份与恢复Tablestore实例下的数据表，不支持备份索引和时序表。

操作步骤

您可以按如下步骤创建Tablestore备份计划。

1. 登录混合云备份管理控制台。
2. 在左侧导航栏，选择**备份 > Tablestore备份**。
3. 在顶部菜单栏左上角，选择所在地域。
4. 在目标Tablestore实例的操作列，单击**备份**。
5. 在**新建备份计划**面板，按照以下说明填写各项参数，然后单击**确定**。
   i. 配置备份内容，然后单击**下一步**。

参数	说明
备份计划名称	为备份计划命名。
备份对象	选中备份对象。备份对象精确到表，您可以同时选择多张表。

ii. 配置备份计划，然后单击**下一步**。

备份类型	参数	说明
全量备份	**全量备份**	打开**全量备份**开关，表示进行全量备份。
	时间粒度	仅当打开**全量备份**开关时，需要配置该参数。指定备份的周期，支持按天、星期和月进行周期备份。
	执行时间	仅当打开**全量备份**开关时，需要配置该参数。指定开始备份时间，精确到秒。
	备份间隔	仅当**时间粒度**参数取值为*按天（周期备份）*时，需要配置该参数。指定间隔多少天进行备份。
	指定星期	仅当**时间粒度**参数取值为*按星期（周期备份）*时，需要配置该参数。指定周几进行备份。
	指定月份	当**时间粒度**参数取值为*按月（周期备份）*时，需要配置该参数。指定进行备份的月份。
	指定天	当**时间粒度**参数取值为*按月（周期备份）*时，需要配置该参数。指定进行备份的月份的某一天。
增量备份	**增量备份**	打开**增量备份**开关，表示进行增量备份。相比全量备份，只对增加的数据进行备份，数据量较小，备份比较快。
	时间粒度	仅当打开**增量备份**开关时，需要配置该参数。指定备份的周期，支持按分钟、小时和天进行周期备份。
	执行时间	仅当打开**增量备份**开关且**时间粒度**取值为*按天*时，需要配置该参数。指定开始备份时间，精确到秒。
	备份间隔	仅当打开**增量备份**开关时，需要配置该参数。指定间隔30分钟、间隔多少小时或者多少天进行备份。

iii. 配置备份选项。

参数	说明
备份库配置	配置备份保存的备份库。 ■ *新建备份库*：新建备份库的名称命名。可不填，默认名字随机分配。 ■ *选择备份库*：从备份库名称下拉框中选择已有备份库。
备份库名称	指定备份库的名称。

参数	说明
备份库资源组	仅当**备份库配置**参数取值为*新建备份库*时，需要配置该参数。表示备份库属于哪个资源组。 资源组是在阿里云账号下进行资源分组管理的一种机制，资源组能够帮助您解决单个云账号内的资源分组和授权管理的复杂性问题。更多信息，请参见创建资源组。
数据冗余类型	仅当**备份库配置**参数取值为*新建备份库*时，需要配置该参数。表示存储库的数据冗余类型。通过数据冗余机制，可以提高存储库的数据可靠性。 ■ 本地冗余 采用数据冗余存储机制，将每个对象冗余存储在同一个可用区内多个设施的多个设备上，确保硬件失效时的数据持久性和可用性。 ■ 同城冗余 采用多可用区（AZ）机制，将您的数据分散存放在同一地域（Region）的3个可用区。当某个可用区不可用时，仍然能够保障数据的正常访问。 关于存储库类型的介绍，请参见存储库类型。
源端加密方式	仅当**备份库配置**参数取值为*新建备份库*时，需要配置该参数。表示备份库的加密方式。 ■ *HBR完全托管*（默认值）：使用备份服务默认加密方式。 ■ *KMS*：使用阿里云KMS服务自定义密钥加密。需指定**KMS密钥ID**参数。 **注意** ■ 使用KMS加密后，无法再修改KMS加密密钥。 ■ 使用KMS密钥加密备份库，您需要提前创建阿里云KMS服务的KMS加密密钥ID。更多信息，请参见创建密钥。
备份保留时间	指标备份保留时间。默认值为2年。时间单位：天、周、月、年。
使用流量控制	流量控制可以帮助您在业务高峰期，控制备份文件的流量，以免影响正常业务。若使用流量控制，您需要根据业务情况，选择**限流时间段**，输入限流时间段内可使用的**最大流量（MB）**，然后单击**添加**。

备份计划创建完成后，HBR将按照指定的备份起始时间、备份执行间隔进行Tablestore备份任务。您可以在**备份计划**页签中看到已创建的备份计划。

14.3. 恢复Tablestore数据

当表格存储（Tablestore）实例出现异常或者存在错误操作时，您可以将备份源中的数据恢复到源Tablestore实例或其他处于同一地域的Tablestore实例中。 本文介绍通过混合云备份HBR创建Tablestore恢复任务的操作方法。

前提条件

已创建Tablestore备份计划并完成备份。具体操作，请参见备份Tablestore数据。

注意事项

- 恢复有自增列的表时， 目前恢复方式仅支持重新生成自增列，并且只恢复putRow操作。在增量备份中，将忽略对表的updateRow和deleteRow操作。例如，原表中对同一行执行了多次put操作，在恢复该表后，由于自增列的值重新生成，每次put会新创建一行数据，因此会变成多行put结果。
- 恢复任务会覆盖目标表中相同Primary Key的行， 其他行不受影响。
- 选择恢复表时，如果目标表名不存在， 恢复任务会根据备份的表创建一张新表。如果恢复到一张已经存在的表，请注意目标表的Schema需要与原表保持一致，并且需要为目标表设置合理的数据有效版本偏差和数据生命周期，以防止恢复时写入失败或者数据恢复之后即过期。

操作步骤

1. 登录混合云备份管理控制台。
2. 在左侧导航栏，选择**备份 > Tablestore备份**。
3. 在顶部菜单栏左上角，选择所在地域。
4. 在目标Tablestore实例的操作列，单击**恢复**。
5. 在**新建恢复任务**面板，配置恢复内容和目标实例，然后单击**确定**。

i. 配置恢复内容，然后单击**下一步**。

参数	说明
可恢复表	从**可恢复表**列表中，选择可恢复的表名称。该恢复表名称来源您备份时指定的备份表名称，若您未备份某个表格，则不会出现在这里。 **注意** 暂时只支持一次恢复一张表。
可恢复时间	从**可恢复时间**列表中，选择可恢复时间段。该恢复时间点来源您备份时指定的备份计划执行时间。
可恢复时间点	选择可恢复时间点。

ii. 配置目标实例。

参数	说明
恢复到数据库	从**恢复到数据库**列表中，选择可恢复的数据库名称。该恢复数据库名称来源您名下的所有Tablestore资源。当指定恢复到其他Tablestore实例时，就完成跨表格实例的恢复。
恢复到新表名	恢复后新的表名称。 **注意** 该恢复表名不能和已经存在的表名重复。

恢复任务创建后，可以在**恢复任务**页签的**状态**栏查看恢复任务进度。

15.使用限制

15.1. 通用限制

本文介绍了表格存储的通用使用限制。为保证更好的性能，请合理设计表结构和单行数据大小。

实例限制

资源	限制值	说明
单个阿里云账号下可以保有实例数	10个	如果业务需求超过限制，请提交工单联系我们。
单实例中表的个数	64个	限制值包含数据表和索引表的数量。 如果业务需求超过限制，请提交工单联系我们。
实例名称长度	3~16 Bytes	实例名称需由a~z、A~Z、0~9和短划线（-）组成，首字符必须是字母且末尾字符不能为短划线（-）。

表限制

资源	限制值	说明
表名长度	1~255 Bytes	表名需由a~z、A~Z、0~9和下划线（_）组成。首字符必须是字母或下划线（_）。
单表的预留读写吞吐量	0~100000 CU	如果业务需求超过限制，请提交工单联系我们。
预定义列	0~32列	预定义列是为数据表预先定义一些非主键列以及其类型。使用二级索引时，预定义列可以作为索引表的索引列或者属性列。 **注意** 使用多元索引时，无需为数据表设置预定义列。

列限制

资源	限制值	说明
列名长度限制	1~255 Bytes	列名需由a~z、A~Z、0~9和下划线（_）组成。首字符必须是字母或下划线（_）。
主键包含的列数	1~4列	最少1列，最多4列。
String类型主键列列值大小	1 KB	单一主键列String类型的列值大小上限1 KB。
String类型属性列列值大小	2 MB	单一属性列String类型的列值大小上限2 MB。

资源	限制值	说明
Binary类型主键列列值大小	1 KB	单一主键列Binary类型的列值大小上限1 KB。
Binary类型属性列列值大小	2 MB	单一属性列Binary类型的列值大小上限2 MB。

行限制

资源	限制值	说明
一行中属性列的个数	无限制	无。
单行数据大小	无限制	不限制单一行中所有列名与列值总和大小。

操作限制

操作	限制值	说明
一次请求写入的属性列的个数	1024列	使用PutRow、UpdateRow或BatchWriteRow接口操作时，单行写入的属性列的个数不能超过1024列。
读请求中columns_to_get参数的列的个数	0~128个	读请求一行数据中获取的列的最大个数。
表元数据操作QPS	10次/秒	一个实例的表元数据操作每秒不超过10次，关于表元数据的具体操作，请参见表操作。
单表UpdateTable的次数	无限制	需要遵循单表的调整频率限制。
单表UpdateTable的频率	每2分钟1次	单表在2分钟之内，最多允许调整1次预留读或预留写能力值。
BatchGetRow一次操作请求读取的行数	100行	无。
BatchWriteRow一次操作请求写入行数	200行	无。
BatchWriteRow一次操作的数据大小	4 MB	无。
PutRow一次操作的数据大小	4 MB	无。
UpdateRow一次操作的数据大小	4 MB	无。
GetRange一次扫描的数据	5000行或者4 MB	一次扫描数据的最大行数为5000行或者数据大小最大为4 MB，超出上限的数据将会按行级别被截掉并返回下一行数据主键信息。
一次HTTP请求Request Body的数据大小	5 MB	无。
一次读请求时的过滤器个数	10个	无。

15.2. 二级索引限制

通过本文您可以了解二级索引的使用限制。

索引表限制

资源	限制值	说明
表名长度	1~255 Bytes	表名需由a~z、A~Z、0~9和下划线（_）组成。首字符必须是字母或下划线（_）。
单表二级索引个数	5	每张数据表最多创建5个索引表。
索引列个数	1～4	索引表中最多添加4个索引列，索引列为数据表主键和预定义列的任意组合。 索引表的主键由索引列和数据表自动补齐的主键列组成。
索引列的数据类型	String、Integer、Binary	索引列支持的数据类型包括String、Integer和Binary。
属性列个数	32	索引表中最多添加32个属性列，索引表属性列为数据表的预定义列的组合。
属性列的数据类型	String、Integer、Double、Boolean、Binary	属性列支持的数据类型包括String、Integer、Double、Boolean和Binary。

其他限制

资源	限制值	说明
索引列	非自增列	索引表的第一主键列不能为自增列。
索引表数据操作	只读不写	只能读取索引表中的数据，不能对索引表进行写操作。
数据多版本	不支持	在开启数据多版本的数据表上不支持创建二级索引。
数据生命周期（TTL）	支持	需要禁止数据表更新，保证索引表的TTL和数据表TTL一致。
Stream功能	不支持	无。
反查数据表	不支持	需要手动反查数据表。

15.3. 多元索引限制

通过本文您可以了解多元索引的使用限制。

Mapping

名称	最大值	说明
多元索引字段数量	500	可被索引的字段数。
数组长度	256	数组中最多包含的元素个数。
EnableSortAndAgg字段数量	100	可被排序和统计聚合的字段数。
Nested嵌套层数	5	最多支持5层Nested嵌套。
Nested字段的子行数量	256	嵌套字段的嵌套行最大数量，非子字段数量。
Nested字段数量	25	嵌套中子字段的个数。
表主键列长度之和	1000字节	所有主键列的长度累加后不超过1000字节。
表主键列中String长度	1000字节	当String类型的主键列要建立索引时，列值长度不能超过1000字节。
表属性列中String长度（索引成Keyword）	4 KB	无。
表属性列中String长度（索引成Text）	2 MB	与数据表中属性列的长度限制相同。
通配符查询的Query长度	32	不超过32字符。
前缀查询的Query长度	1000字节	不超过1000字节。

Search

类别	名称	最大值	说明
通用限制	offset+limit	10000	如果业务需求超过限制，请使用next_token。
	limit	100	• 当使用Search接口查询指定列的数据时，如果多元索引中包含了查询列的数据时，limit参数最大支持到1000，即一个请求最多返回1000条结果。 • 如果业务需求超过限制，请提交工单联系我们。
	timeout	10s	无。
	CU	10万	• 除扫描、分析类外。 • 如果业务需求超过限制，请提交工单联系我们。

类别	名称	最大值	说明
	QPS	10万	• 针对于轻量的事务型查询（TP），QPS上限为10万。 • 如果业务需求超过限制，请提交工单联系我们。
	一次Search查询中的Query个数	1024	如果Search查询中的Query嵌套太复杂，则会影响查询效率，请注意精简Query。
统计聚合	同层Aggregation个数	5个	SubGroupBy中添加Aggregation重新从0计算。
	同层GroupBy个数	5个	SubGroupBy中添加GroupBy重新从0计算。
	GroupBy嵌套层数	3层	层数计算包含本身的GroupBy。
	GroupByFilter中的Filter个数	10个	无。
	GroupByField返回的分组个数	2000个	无。
	GroupByRange中的Range个数	100个	无。
	GroupByGeodistance中的Range个数	10个	无。

ParallelScan

类别	名称	说明
通用限制	offset+limit	无法指定offset+limit，只能从最开始往后面遍历。
	limit	最大值为2000。
	CU	无限制。
	QPS	无限制。
	最大并发	ComputeSplits的返回值中的MaxParallel值。

Index

名称	最大值	说明
速率	5万行/s	• 初始写入或瞬间写入时会有分钟级别负载均衡时间。 • 文本类型的由于涉及分词会有较高CPU消耗，限制为1万行/s。 • 如果业务需求超过限制，请提交工单联系我们。

名称	最大值	说明
同步延迟	3s	• 99%情况下在3秒内。 • 新建索引最多会有1分钟的初始化时间。
行数	500亿	如果业务需求超过限制，请提交工单联系我们。
总大小	50 TB	如果业务需求超过限制，请提交工单联系我们。

其他限制

目前多元索引功能开放的地域包括华东1（杭州）、华东1 金融云、华东2（上海）、华北2（北京）、华北3（张家口）、华南1（深圳）、中国（香港）、新加坡、澳大利亚（悉尼）、印度尼西亚（雅加达）、日本（东京）、德国（法兰克福）、英国（伦敦）、美国（硅谷）、美国（弗吉尼亚）、印度（孟买）和菲律宾（马尼拉）。

说明 如果上述限制项不能满足您的业务需求，请在阿里云官网提交工单申请更高需求。在工单中请说明场景、限制项名称、限制项的数量需求、申请需求的原因，在后续功能开发中会优先考虑您的需求。

15.4. SQL使用限制

通过本文您可以了解SQL的使用限制。

基础限制

注意 数据库名称、表名和列名均不能为SQL中的保留字与关键字。关于保留字与关键字的更多信息，请参见保留字与关键字。

资源	取值范围	说明
数据库名称长度	3~16字节	对应于实例名称。 数据库名称需由a~z、A~Z、0~9和连词符（-）组成，首字符必须是字母且末尾字符不能为连词符（-）。
表名称长度	1~255字节	对应于数据表或者索引表名称。 表名称需由a~z、A~Z、0~9和下划线（_）组成。首字符必须是字母或下划线（_）。
列名长度	1~255字节	对应于数据表或者索引表中的列名。 列名需由a~z、A~Z、0~9和下划线（_）组成。首字符必须是字母或下划线（_）。
列的个数	1~32	如果业务需求超过限制，请提交工单联系我们。

资源	取值范围	说明
String类型主键列列值大小	1 KB	单一主键列String类型的列值大小上限1 KB。
String类型属性列列值大小	2 MB	单一属性列String类型的列值大小上限2 MB。
Binary类型（Blob）主键列列值大小	1 KB	单一主键列Binary类型（Blob）的列值大小上限1 KB。
Binary类型（Blob）属性列列值大小	2 MB	单一属性列Binary类型（Blob）的列值大小上限2 MB。

操作限制

资源	限制值	说明
单次扫描数据量	128 MB或者10万行	一次扫描数据的最大行数为10万行或者数据大小最大为128 MB。超出上限后，系统会返回错误。
单次执行时间	30 s	单次执行时间与SQL语句的复杂度以及表中数据量相关，最大时长为30秒。超过时长后，系统会返回错误。
列的数据类型	不支持修改	不支持修改列的数据类型及列位置。
大小写敏感	不敏感	由于表格存储中原表名和列名均是大小写敏感的，当使用SQL时，原表名和列名会统一转换为小写字母进行匹配，即如果要操作表格存储中的Aa列，在SQL中使用AA、aa、aA、Aa均可，因此在原表名或者列名不能同时为AA、aa、aA和Aa。

15.5. 时序模型限制

通过本文您可以了解时序模型的使用限制。

限制项	描述
时序表名	时序表名必须由字母、数字和下划线组成，且不能以数字开头。长度为1~128个字节。
时序列名	时序列名必须由小写字母、数字和下划线（_）组成，且不能以数字开头。长度不超过1~128个字节。 时序列名中不能包含_m_name、_data_source、_tags, _time、_meta_update_time和_attributes。
度量名称（measurement name）	度量名称为UTF-8字符串，不能出现不可见字符（包括空格）和#，长度为1~128个字节。
数据源（data source）	数据源为UTF-8字符串，长度为0~256个字节。

限制项	描述
标签（tags）	按照["k1=v1","k2=v2"]格式拼装，等号前为tagKey，等号后为tagValue。 tagKey必须为ASCII码中可见字符，tagValue可以为UTF-8字符串。tagKey和tagValue中都不能出现双引号和等号。标签长度不超过512个字节。
时间列	大于等于0，单位为微秒（us）。
一次写入列数限制	单行写入的属性列的个数不能超过1024列。
一次写入行数限制	一次写入的行数不能超过200行。
一次写入数据大小限制	一次写入数据大小上限为4 MB。
String类型列值大小	String类型的列值大小上限为2 MB。
Binary类型列值大小	Binary类型的列值大小上限为2 MB。

16.HBase支持

16.1. Tablestore HBase Client

除了使用现有的SDK以及Restful API来访问表格存储，表格存储还提供了Tablestore HBase Client。使用开源HBase API的Java应用可以通过Tablestore HBase Client来直接访问表格存储服务。

Tablestore HBase Client基于表格存储4.2.x以上版本的Java SDK，支持1.x.x版本以上的开源HBase API。

Tablestore HBase Client可以从以下三个途径获取：

- GitHub：tablestore-hbase-client项目
- 压缩包下载
- Maven

```
<dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
       </dependency>
   </dependencies>
```

当使用Tablestore HBase Client之后，您不再需要关心HBase Server的相关事项，只需要通过Client提供的接口进行表或者数据的操作即可。

相比自行搭建HBase服务，表格存储的优势请参见下表。

对比项	表格存储	自建HBase集群
成本	根据实际用量进行计费，提供高性能与容量型两种规格实例，适用于不同的应用场景。	需要根据业务峰值进行资源配置，空闲时段资源被闲置，租用及人工运维成本高。
安全	整合阿里云RAM资源权限管理系统，支持多种鉴权和授权机制以及VPC、阿里云账号、RAM用户功能，授权粒度达到表级别和API级别。	需要额外的安全机制。
可靠性	数据自动多重冗余备份，故障迁移自动完成，可用性不低于99.9%，数据可靠性达99.99999999%。	需要自行保障集群的可用性。
可扩展性	表格存储的自动负载均衡机制支持单表PB级数据，即使百万并发也无需任何人工扩容。	集群利用率到一定水位之后需要繁琐的机器上下线流程，影响在线业务。

16.2. Tablestore HBase Client 支持的功能

本文主要为您介绍Tablestore HBase Client 支持的功能和操作。

表格存储与 HBase 的 API 区别

作为 NoSQL 数据库服务，表格存储为您屏蔽了数据表分裂、Dump、Compact、Region Server 等底层相关的细节，您只需要关心数据的使用。因此，虽然与HBase在数据模型及功能上相近，Tablestore HBase Client 与原生的 HBase API 仍然有一些区别。

支持的功能

- CreateTable

  表格存储不支持列族（ColumnFamily），所有的数据可以认为是在同一个 ColumnFamily 之内，所以表格存储的 TTL 及 Max Versions 都是数据表级别的，支持如下相关功能：

功能	支持情况
family max version	支持表级别 max version，默认为 1
family min version	不支持
family ttl	支持表级别 TTL
is/set ReadOnly	通过 RAM 子账号支持
预分区	不支持
blockcache	不支持
blocksize	不支持
BloomFilter	不支持
column max version	不支持
cell ttl	不支持
控制参数	不支持

- Put

功能	支持情况
一次写入多列数据	支持
指定一个时间戳	支持
如果不写时间戳，默认用系统时间	支持
单行 ACL	不支持
ttl	不支持
Cell Visibility	不支持
tag	不支持

- Get

表格存储保证数据的强一致性，在数据写入 API 收到 HTTP 200 状态码（OK）的回复时，数据即被持久化到所有的备份上，这些数据能够马上被 Get 读到。

功能	支持情况
读取一行数据	支持
读取一个列族里面的所有列	支持
读取特定列的数据	支持
读取特定时间戳的数据	支持
读取特定个数版本的数据	支持
TimeRange	支持
ColumnfamilyTimeRange	不支持
RowOffsetPerColumnFamily	支持
MaxResultsPerColumnFamily	不支持
checkExistenceOnly	不支持
closestRowBefore	支持
attribute	不支持
cacheblock:true	支持
cacheblock:false	不支持
IsolationLevel:READ_COMMITTED	支持
IsolationLevel:READ_UNCOMMITTED	不支持
IsolationLevel:STRONG	支持
IsolationLevel:TIMELINE	不支持

- Scan

  表格存储保证数据的强一致性，在数据写入 API 收到 HTTP 200 状态码（OK）的回复时，数据即被持久化到所有的备份上，这些数据能够马上被 Scan 读到。

功能	支持情况
指定 start、stop 确定扫描范围	支持
如果不指定扫描范围，默认扫描全局	支持
prefix filter	支持

功能	支持情况
读取逻辑同 Get	支持
逆序读	支持
caching	支持
batch	不支持
maxResultSize，返回数据量大小的限制	不支持
small	不支持
batch	不支持
cacheblock:true	支持
cacheblock:false	不支持
IsolationLevel:READ_COMMITTED	支持
IsolationLevel:READ_UNCOMMITTED	不支持
IsolationLevel:STRONG	支持
IsolationLevel:TIMELINE	不支持
allowPartialResults	不支持

- Batch

功能	支持情况
Get	支持
Put	支持
Delete	支持
batchCallback	不支持

- Delete

功能	支持情况
删除整行	支持
删除特定列的所有版本	支持
删除特定列的特定版本	支持
删除特定列族	不支持

功能	支持情况
指定时间戳时，deleteColumn 会删除等于这个时间戳的版本	支持
指定时间戳时，deleteFamily 和 deleteColumns 会删除小于等于这个时间戳的所有版本	不支持
不指定时间戳时，deleteColumn 会删除最近的版本	不支持
不指定时间戳时，deleteFamily 和 deleteColumns 会删除当前系统时间的版本	不支持
addDeleteMarker	不支持

- checkAndXXX

功能	支持情况
CheckAndPut	支持
checkAndMutate	支持
CheckAndDelete	支持
检查列的值是否满足条件，满足则删除	支持
如果不指定值，则表示缺省	支持
跨行，检查 A 行，执行 B 行	不支持

- exist

功能	支持情况
判断一行或多行是否存在，不返回内容	支持

- Filter

功能	支持情况
ColumnPaginationFilter	不支持 columnOffset 和 count
SingleColumnValueFilter	支持： LongComparator，BinaryComparator，ByteArrayComparable 不支持： RegexStringComparator，SubstringComparator，BitComparator

不支持的方法

- Namespaces

  表格存储上使用__实例__对数据表进行管理。实例是表格存储最小的计费单元，用户可以在表格存储控制台上进行实例的管理，所以不再支持如下 Namespaces 相关的操作：

  - createNamespace(NamespaceDescriptor descriptor)
  - deleteNamespace(String name)
  - getNamespaceDescriptor(String name)
  - listNamespaceDescriptors()
  - listTableDescriptorsByNamespace(String name)
  - listTableNamesByNamespace(String name)
  - modifyNamespace(NamespaceDescriptor descriptor)

- Region 管理

  表格存储中数据存储和管理的基本单位为，表格存储会自动地根据数据分区的数据大小、访问情况进行分裂或者合并，所以不支持 HBase 中 Region 管理相关的方法。

- Snapshots

  表格存储目前不支持 Snapshots，所以暂时不支持 Snapshots 相关的方法。

- Table 管理

  表格存储会自动对 Table 下的数据分区进行分裂、合并及 Compact 等操作，所以不再支持如下方法：

  - getTableDescriptor(TableName tableName)
  - compact(TableName tableName)
  - compact(TableName tableName, byte[] columnFamily)
  - flush(TableName tableName)
  - getCompactionState(TableName tableName)
  - majorCompact(TableName tableName)
  - majorCompact(TableName tableName, byte[] columnFamily)
  - modifyTable(TableName tableName, HTableDescriptor htd)
  - split(TableName tableName)
  - split(TableName tableName, byte[] splitPoint)

- Coprocessors

  表格存储暂时不支持协处理器，所以不支持如下方法：

  - coprocessorService()
  - coprocessorService(ServerName serverName)
  - getMasterCoprocessors()

- Distributed procedures

  表格存储不支持 Distributed procedures，所以不支持如下方法：

  - execProcedure(String signature, String instance, Map props)
  - execProcedureWithRet(String signature, String instance, Map props)
  - isProcedureFinished(String signature, String instance, Map props)

- Increment 与 Append

暂不支持原子增减和原子 Append。

16.3. 表格存储和 HBase 的区别

Table Store HBase Client 的使用方式与 HBase 类似，但存在一些区别。本节内容介绍 Table Store HBase Client 的特点。

Table

不支持多列族，只支持单列族。

Row和Cell

- 不支持设置 ACL
- 不支持设置 Cell Visibility
- 不支持设置 Tag

GET

表格存储只支持单列族，所以不支持列族相关的接口，包括：

- setColumnFamilyTimeRange(byte[] cf, long minStamp, long maxStamp)
- setMaxResultsPerColumnFamily(int limit)
- setRowOffsetPerColumnFamily(int offset)

SCAN

类似于 GET，既不支持列族相关的接口，也不能设置优化类的部分接口，包括：

- setBatch(int batch)
- setMaxResultSize(long maxResultSize)
- setAllowPartialResults(boolean allowPartialResults)
- setLoadColumnFamiliesOnDemand(boolean value)
- setSmall(boolean small)

Batch

暂时不支持 BatchCallback。

Mutations 和 Deletions

- 不支持删除特定列族
- 不支持删除最新时间戳的版本
- 不支持删除小于某个时间戳的所有版本

Increment 和 Append

暂时不支持

Filter

- 支持 ColumnPaginationFilter
- 支持 FilterList

- 部分支持 SingleColumnValueFilter，比较器仅支持 BinaryComparator
- 其他 Filter 暂时都不支持

Optimization

HBase 的部分接口涉及到访问、存储优化等，这些接口目前没有开放：

- blockcache：默认为 true，不允许用户更改
- blocksize：默认为 64K，不允许用户更改
- IsolationLevel：默认为 READ_COMMITTED，不允许用户更改
- Consistency：默认为 STRONG，不允许用户更改

Admin

HBase 中的接口 `org.apache.hadoop.hbase.client.Admin` 主要是指管控类的 API，而其中大部分的 API 在表格存储中是不需要的。

由于表格存储是云服务，运维、管控类的操作都会被自动执行，用户不需要关注。其他一些少量接口，目前暂不支持。

- CreateTable

  表格存储只支持单列族，在创建表时只允许设置一个列族，列族中支持 MaxVersion 和 TimeToLive 两个参数。

- Maintenance task

  在表格存储中，下列的任务维护相关接口都会被自动处理：

  - abort(String why, Throwable e)
  - balancer()
  - enableCatalogJanitor(boolean enable)
  - getMasterInfoPort()
  - isCatalogJanitorEnabled()
  - rollWALWriter(ServerName serverName) -runCatalogScan()
  - setBalancerRunning(boolean on, boolean synchronous)
  - updateConfiguration(ServerName serverName)
  - updateConfiguration()
  - stopMaster()
  - shutdown()

- Namespaces

  在表格存储中，实例名称类似于 HBase 中的 Namespaces，因此不支持 Namespaces 相关的接口，包括：

  - createNamespace(NamespaceDescriptor descriptor)
  - modifyNamespace(NamespaceDescriptor descriptor)
  - getNamespaceDescriptor(String name)
  - listNamespaceDescriptors()
  - listTableDescriptorsByNamespace(String name)
  - listTableNamesByNamespace(String name)

- deleteNamespace(String name)

- Region

  表格存储中会自动处理 Region 相关的操作，因此不支持以下接口：

  - assign(byte[] regionName)
  - closeRegion(byte[] regionname, String serverName)
  - closeRegion(ServerName sn, HRegionInfo hri)
  - closeRegion(String regionname, String serverName)
  - closeRegionWithEncodedRegionName(String encodedRegionName, String serverName)
  - compactRegion(byte[] regionName)
  - compactRegion(byte[] regionName, byte[] columnFamily)
  - compactRegionServer(ServerName sn, boolean major)
  - flushRegion(byte[] regionName)
  - getAlterStatus(byte[] tableName)
  - getAlterStatus(TableName tableName)
  - getCompactionStateForRegion(byte[] regionName)
  - getOnlineRegions(ServerName sn)
  - majorCompactRegion(byte[] regionName)
  - majorCompactRegion(byte[] regionName, byte[] columnFamily)
  - mergeRegions(byte[] encodedNameOfRegionA, byte[] encodedNameOfRegionB, boolean forcible)
  - move(byte[] encodedRegionName, byte[] destServerName)
  - offline(byte[] regionName)
  - splitRegion(byte[] regionName)
  - splitRegion(byte[] regionName, byte[] splitPoint)
  - stopRegionServer(String hostnamePort)
  - unassign(byte[] regionName, boolean force)

Snapshots

不支持 Snapshots 相关的接口。

Replication

不支持 Replication 相关的接口。

Coprocessors

不支持 Coprocessors 相关的接口。

Distributed procedures

不支持 Distributed procedures 相关的接口。

Table management

表格存储自动执行 Table 相关的操作，用户无需关注，因此不支持以下接口：

- compact(TableName tableName)

- compact(TableName tableName, byte[] columnFamily)
- flush(TableName tableName)
- getCompactionState(TableName tableName)
- majorCompact(TableName tableName)
- majorCompact(TableName tableName, byte[] columnFamily)
- modifyTable(TableName tableName, HTableDescriptor htd)
- split(TableName tableName)
- split(TableName tableName, byte[] splitPoint)

限制项

表格存储是云服务，为了整体性能最优，对部分参数做了限制，且不支持用户通过配置修改，具体限制项请参见表格存储限制项。

16.4. 从HBase Client迁移到Tablestore HBase Client

Tablestore HBase Client是基于HBase Client的封装，使用方法和HBase Client基本一致，但仍存在一些差别。本文主要为您介绍如何从HBase Client迁移到Tablestore HBase Client 。

> **说明** 如果您需要将HBase中的全量数据迁移到表格存储，参见将HBase数据同步到表格存储。

依赖

Tablestore HBase Client 1.2.0版本依赖了HBase Client 1.2.0版本和Tablestore Java SDK 4.2.1版本。pom.xml配置如下：

```
<dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
       </dependency>
   </dependencies>
```

如果需要使用其他版本的HBase Client或Tablestore Java SDK，可以使用exclusion标签。下面示例中使用HBase Client 1.2.1版本和Tablestore4.2.0版本。

```
<dependencies>
     <dependency>
         <groupId>com.aliyun.openservices</groupId>
         <artifactId>tablestore-hbase-client</artifactId>
         <version>1.2.0</version>
         <exclusions>
             <exclusion>
                 <groupId>com.aliyun.openservices</groupId>
                 <artifactId>tablestore</artifactId>
             </exclusion>
             <exclusion>
                 <groupId>org.apache.hbase</groupId>
                 <artifactId>hbase-client</artifactId>
             </exclusion>
         </exclusions>
     </dependency>
     <dependency>
         <groupId>org.apache.hbase</groupId>
         <artifactId>hbase-client</artifactId>
         <version>1.2.1</version>
     </dependency>
     <dependency>
         <groupId>com.aliyun.openservices</groupId>
         <artifactId>tablestore</artifactId>
         <classifier>jar-with-dependencies</classifier>
         <version>4.2.0</version>
     </dependency>
 </dependencies>
```

HBase Client 1.2.x和其他版本（如 1.1.x）存在接口变化，而Tablestore HBase Client 1.2.x版本只能兼容HBase Client 1.2.x。

如果需要使用HBase Client 1.1.x版本，请使用Tablestore HBase Client 1.1.x版本。

如果需要使用HBase Client 0.x.x版本，请参考迁移较早版本的 HBase。

配置文件

从HBase Client迁移到Tablestore HBase Client，需要在配置文件中修改以下两点：

- HBase Connection类型

  Connection需要配置为TablestoreConnection。

  ```
  <property>
      <name>hbase.client.connection.impl</name>
      <value>com.alicloud.tablestore.hbase.TablestoreConnection</value>
  </property>
  ```

- 表格存储的配置项

  表格存储是云服务，提供了严格的权限管理。要访问表格存储，需要配置密钥等信息。

- 必须配置以下四个配置项才能成功访问表格存储：

```
<property>
            <name>tablestore.client.endpoint</name>
            <value></value>
        </property>
        <property>
            <name>tablestore.client.instancename</name>
            <value></value>
        </property>
        <property>
            <name>tablestore.client.accesskeyid</name>
            <value></value>
        </property>
        <property>
            <name>tablestore.client.accesskeysecret</name>
            <value></value>
        </property>
```

- 下面为可选配置项：

```
        <property>
            <name>hbase.client.tablestore.family</name>
            <value>f1</value>
        </property>
        <property>
            <name>hbase.client.tablestore.family.$tablename</name>
            <value>f2</value>
        </property>
        <property>
            <name>tablestore.client.max.connections</name>
            <value>300</value>
        </property>
        <property>
            <name>tablestore.client.socket.timeout</name>
            <value>15000</value>
        </property>
        <property>
            <name>tablestore.client.connection.timeout</name>
            <value>15000</value>
        </property>
        <property>
            <name>tablestore.client.operation.timeout</name>
            <value>2147483647</value>
        </property>
        <property>
            <name>tablestore.client.retries</name>
            <value>3</value>
        </property>
```

- hbase.client.tablestore.family与hbase.client.tablestore.family.$tablename
  - 表格存储只支持单列族，使用HBase API时，需要有一项family的内容，因此通过配置来填充此项family的内容。

    其中，`hbase.client.tablestore.family`为全局配置，`hbase.client.tablestore.family.$tablename`为单个表的配置。
  - 规则为：对表名为T的表，先找`hbase.client.tablestore.family.T`，如果不存在则找`hbase.client.tablestore.family`，如果依然不存在则取默认值f。
- tablestore.client.max.connections

  最大链接数，默认是300。
- tablestore.client.socket.timeout

  Socket超时时间，默认是15秒。
- tablestore.client.connection.timeout

  连接超时时间，默认是15秒。
- tablestore.client.operation.timeout

  API超时时间，默认是Integer.MAX_VALUE，类似于永不超时。
- tablestore.client.retries

  请求失败时，重试次数，默认是3次。

16.5. 如何兼容Hbase 1.0以前的版本

Table Store HBase Client 目前支持 HBase Client 1.0.0 及以上版本的 API。本文主要为您介绍Table Store HBase Client 如何兼容Hbase 1.0以前的版本的API。

HBase Client 1.0.0 版本相对于之前版本有一些较大的变化，这些变化是不兼容的。

为了协助一些使用老版本 HBase 的用户能方便地使用表格存储，本节我们将介绍 HBase 1.0 相较于旧版本的一些较大变化，以及如何使其兼容。

Connection 接口

HBase 1.0.0 及以上的版本中废除了 HConnection 接口，并推荐使用`org.apache.hadoop.hbase.client.ConnectionFactory`类，创建一个实现 Connection 接口的类，用 ConnectionFactory 取代已经废弃的 ConnectionManager 和 HConnectionManager。

创建一个 Connection 的代价比较大，但 Connection 是线程安全的。使用时可以在程序中只生成一个 Connection 对象，多个线程可以共享这一个对象。

HBase 1.0.0 及以上的版本中，您需要管理 Connection 的生命周期，并在使用完以后将其它 close。

最新的代码如下所示：

```
Connection connection = ConnectionFactory.createConnection(config);
// ...
connection.close();
```

TableName 类

在HBase 1.0.0 之前的版本中，创建表时可以使用 String 类型的表名，但是 HBase 1.0.0 之后需要使用类 `org.apache.hadoop.hbase.TableName` 。

最新的代码如下所示：

```
String tableName = "MyTable";
// or byte[] tableName = Bytes.toBytes("MyTable");
TableName tableNameObj = TableName.valueOf(tableName);
```

Table、BufferedMutator 和 RegionLocator 接口

从 HBase Client 1.0.0 开始，HTable 接口已经废弃，取而代之的是 Table、BufferedMutator 和 RegionLocator 三个接口。

- `org.apache.hadoop.hbase.client.Table` ：用于操作单张表的读写等请求。
- `org.apache.hadoop.hbase.client.BufferedMutator` ：用于异步批量写，对应于旧版本 HTableInterface 接口中的 `setAutoFlush(boolean)` 。
- `org.apache.hadoop.hbase.client.RegionLocator` ：表分区信息。

Table、BufferedMutator 和 RegionLocator 三个接口都不是线程安全的，但比较轻量，可以为每个线程创建一个对象。

Admin 接口

从 HBase Client 1.0.0 开始，新接口 `org.apache.hadoop.hbase.client.Admin` 取代了HBaseAdmin 类。由于表格存储是一个云服务，大多数运维类接口都是自动处理的，所以 Admin 接口中的众多接口都不会被支持，具体区别请参见表格存储和 HBase 的区别。

通过 Connection 实例创建 Admin 实例：

```
Admin admin = connection.getAdmin();
```

16.6. 快速入门

本文描述如何使用Tablestore HBase Client实现一个简单的程序。

> **说明** 当前示例程序使用了HBase API访问表格存储服务，完整的示例程序位于Github的 hbase项目中，目录位置为*src/test/java/samples/HelloWorld.java*。

操作步骤

1. 配置项目依赖。

   Maven的依赖配置如下：

   ```
   <dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
       </dependency>
   </dependencies>
   ```

关于配置项目依赖的更多信息，请参见从HBase迁移到表格存储。

2. 配置文件。

   hbase-site.xml中增加以下配置项：

```
<configuration>
    <property>
        <name>hbase.client.connection.impl</name>
        <value>com.alicloud.tablestore.hbase.TablestoreConnection</value>
    </property>
    <property>
        <name>tablestore.client.endpoint</name>
        <value>endpoint</value>
    </property>
    <property>
        <name>tablestore.client.instancename</name>
        <value>instance_name</value>
    </property>
    <property>
        <name>tablestore.client.accesskeyid</name>
        <value>access_key_id</value>
    </property>
    <property>
        <name>tablestore.client.accesskeysecret</name>
        <value>access_key_secret</value>
    </property>
    <property>
        <name>hbase.client.tablestore.family</name>
        <value>f1</value>
    </property>
    <property>
        <name>hbase.client.tablestore.table</name>
        <value>ots_adaptor</value>
    </property>
</configuration>
```

   关于配置的更多信息，请参见从HBase迁移到表格存储。

3. 连接表格存储。

   通过创建一个TableStoreConnection对象来链接表格存储服务。

```
Configuration config = HBaseConfiguration.create();
// 创建一个Tablestore Connection。
Connection connection = ConnectionFactory.createConnection(config);
// Admin负责创建、管理、删除等。
Admin admin = connection.getAdmin();
```

4. 创建表。

   通过指定表名创建一张表，MaxVersion和TimeToLive使用默认值。

```
// 创建一个HTableDescriptor，只有一个列族。
HTableDescriptor descriptor = new HTableDescriptor(TableName.valueOf(TABLE_NAME));
// 创建一个列族，MaxVersion和TimeToLive使用默认值，MaxVersion默认值为1，TimeToLive默认值为Int
eger.INF_MAX。
descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));
// 通过Admin的createTable接口创建表。
System.out.println("Create table " + descriptor.getNameAsString());
admin.createTable(descriptor);
```

5. 写数据。

   写入一行数据到表格存储。

```
// 创建一个TablestoreTable，用于单个表上的读写更新删除等操作。
Table table = connection.getTable(TableName.valueOf(TABLE_NAME));
// 创建一个Put对象，主键为row_1。
System.out.println("Write one row to the table");
Put put = new Put(ROW_KEY);
// 增加一列，表格存储只支持单列族，列族名称在hbase-site.xml中配置。如果未配置则默认为f，所以写入
数据时COLUMN_FAMILY_NAME可以为空值。
put.addColumn(COLUMN_FAMILY_NAME, COLUMN_NAME, COLUMN_VALUE);
// 执行Table的put操作，使用HBase API将这一行数据写入表格存储。
table.put(put);
```

6. 读数据。

   读取指定行的数据。

```
// 创建一个Get对象，读取主键为ROW_KEY的行。
Result getResult = table.get(new Get(ROW_KEY));
Result result = table.get(get);
// 打印结果。
String value = Bytes.toString(getResult.getValue(COLUMN_FAMILY_NAME, COLUMN_NAME));
System.out.println("Get one row by row key");
System.out.printf("\t%s = %s\n", Bytes.toString(ROW_KEY), value);
```

7. 扫描数据。

   范围读取数据。

```
// 扫描全表所有行数据.
System.out.println("Scan for all rows:");
Scan scan = new Scan();
ResultScanner scanner = table.getScanner(scan);
// 循环打印结果.
for (Result row : scanner) {
    byte[] valueBytes = row.getValue(COLUMN_FAMILY_NAME, COLUMN_NAME);
    System.out.println('\t' + Bytes.toString(valueBytes));
}
```

8. 删表。

   使用Admin API删除一张表。

```
System.out.println("Delete the table");
admin.disableTable(table.getName());
admin.deleteTable(table.getName());
```

完整代码

```
package samples;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
import java.io.IOException;
public class HelloWorld {
    private static final byte[] TABLE_NAME = Bytes.toBytes("HelloTablestore");
    private static final byte[] ROW_KEY = Bytes.toBytes("row_1");
    private static final byte[] COLUMN_FAMILY_NAME = Bytes.toBytes("f");
    private static final byte[] COLUMN_NAME = Bytes.toBytes("col_1");
    private static final byte[] COLUMN_VALUE = Bytes.toBytes("col_value");
    public static void main(String[] args) {
        helloWorld();
    }
    private static void helloWorld() {
        try  {
            Configuration config = HBaseConfiguration.create();
            Connection connection = ConnectionFactory.createConnection(config);
            Admin admin = connection.getAdmin();
            HTableDescriptor descriptor = new HTableDescriptor(TableName.valueOf(TABLE_NAME
));
            descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));
            System.out.println("Create table " + descriptor.getNameAsString());
            admin.createTable(descriptor);
            Table table = connection.getTable(TableName.valueOf(TABLE_NAME));
            System.out.println("Write one row to the table");
            Put put = new Put(ROW_KEY);
            put.addColumn(COLUMN_FAMILY_NAME, COLUMN_NAME, COLUMN_VALUE);
            table.put(put);
            Result getResult = table.get(new Get(ROW_KEY));
            String value = Bytes.toString(getResult.getValue(COLUMN_FAMILY_NAME, COLUMN_NAM
E));
            System.out.println("Get a one row by row key");
            System.out.printf("\t%s = %s\n", Bytes.toString(ROW_KEY), value);
            Scan scan = new Scan();
            System.out.println("Scan for all rows:");
            ResultScanner scanner = table.getScanner(scan);
            for (Result row : scanner) {
                byte[] valueBytes = row.getValue(COLUMN_FAMILY_NAME, COLUMN_NAME);
                System.out.println('\t' + Bytes.toString(valueBytes));
            }
            System.out.println("Delete the table");
            admin.disableTable(table.getName());
            admin.deleteTable(table.getName());
            table.close();
            admin.close();
            connection.close();
        } catch (IOException e) {
```

```
            System.err.println("Exception while running HelloTablestore: " + e.toString());
            System.exit(1);
        }
    }
}
```

17.授权管理

17.1. RAM和STS介绍

阿里云权限管理机制包括访问控制（Resource Access Management，简称RAM）和安全凭证管理（Security Token Service，简称STS），可以根据需求使用不同权限的RAM用户来访问表格存储，也支持为用户提供访问的临时授权。使用RAM和STS能极大地提高管理的灵活性和安全性。

RAM的主要作用是控制账号系统的权限。通过使用RAM可以在阿里云账号的权限范围内创建RAM用户，给不同的RAM用户分配不同的权限，从而达到授权管理的目的。更多信息，请参见访问控制产品帮助文档。

STS是一个安全凭证（Token）的管理系统，用来授予临时的访问权限，如此即可通过STS来完成对临时用户的访问授权。更多信息，请参见STS。

背景介绍

RAM和STS需要解决的一个核心问题是如何在不暴露阿里云账号AccessKey的情况下安全地授权别人访问。因为如果阿里云账号AccessKey暴露出去，则会带来极大的安全风险，别人可以随意操作该账号下所有的资源、盗取重要信息等。

RAM提供了一种长期有效的权限控制机制，通过分出不同权限的RAM用户，将不同的权限分给不同的用户，即使RAM用户泄露也不会造成全局的信息泄露。RAM用户在一般情况下是长期有效的。因此，RAM用户的AccessKey是不能泄露的。

相对于RAM提供的长效控制机制，STS提供的是一种临时访问授权，通过STS可以返回临时的AccessKey和Token，这些信息可以直接发给临时用户用来访问表格存储。一般来说，从STS获取的权限会受到更加严格的限制，并且拥有时间限制，因此即使这些信息泄露，对于系统的影响也很小。

基本概念

下表是一些基本概念的简单解释。

基本概念	描述
RAM用户	从阿里云账号中创建出来的RAM用户，在创建的时候可以分配独立的密码和权限，每个RAM用户拥有自己的AccessKey，可以和阿里云账号一样正常完成有权限的操作。一般来说，此处的RAM用户可以理解为具有某种权限的用户，可以被认为是一个具有某些权限的操作发起者。
角色（Role）	表示某种操作权限的虚拟概念，但是没有独立的登录密码和AccessKey。RAM用户可以扮演角色，扮演角色时的权限是该角色自身的权限。
授权策略（Policy）	用来定义权限的规则，例如允许用户读取或者写入某些资源。
资源（Resource）	代表用户可访问的云资源，例如表格存储所有的实例、某个实例或者实例中的某个表等。

RAM用户和角色可以类比为个人和其身份的关系，某人在公司的角色是员工，在家里的角色是父亲，在不同的场景扮演不同的角色，但是还是同一个人。在扮演不同角色的时候也就拥有对应角色的权限。单独的员工或者父亲概念并不能作为一个操作的实体，只有有人扮演了之后才是一个完整的概念。此处还可以体现一个重要的概念，那就是角色可以被多个不同的个人同时扮演。完成角色扮演之后，该个人就自动拥有该角色的所有权限。

使用示例：

某个阿里云用户名为alice，其在表格存储有alice_a和alice_b两个实例。alice对两个实例都拥有完全的权限。

为避免阿里云账号的AccessKey泄露而导致安全风险，alice使用RAM创建了两个RAM用户bob和carol。bob对alice_a拥有读写权限，carol对alice_b拥有读写权限。bob和carol都拥有独立的AccessKey，这样万一泄露了也只会影响其中一个实例，而且alice可以很方便地在控制台取消泄露用户的权限。

假设现在需要授权给其他人读取alice_a中的数据表。这种情况下不应该直接把bob的AccessKey透露出去，可以新建一个角色，例如AliceAReader，给该角色赋予读取alice_a的权限。请注意此时AliceAReader还是没法直接使用，因为并不存在对应AliceAReader的AccessKey，AliceAReader当前仅表示一个拥有访问alice_a权限的虚拟实体。

为了能获取临时授权，此时可以调用STS的AssumeRole接口，通知STS bob将要扮演AliceAReader角色。如果成功，则STS会返回一个临时的AccessKeyId、AccessKeySecret和SecurityToken作为访问凭证。将该凭证发给需要访问的临时用户即可获得访问alice_a的临时权限。临时访问凭证的过期时间在调用AssumeRole时指定。

为什么RAM和STS这么复杂

RAM和STS的概念之所以复杂，是为了权限控制的灵活性而牺牲了部分的易用性。

将RAM用户和角色分开，主要是为了将执行操作的实体和代表权限集合的虚拟实体分开。如果用户本身需要的权限很多，例如读写权限，但是实际上每次操作只需要其中的一部分权限，那么我们就可以创建两个角色，分别具有读和写权限，然后创建一个没有任何权限但是可以拥有扮演两个角色权限的用户。当用户需要读权限时就可以临时扮演其中拥有读权限的角色，写权限同理，以降低每次操作中权限泄露的风险。而且通过扮演角色可以将权限授予其他的阿里云用户，更加方便了协同使用。

当然，提供了灵活性并不代表一定要使用全部的功能，应该根据需求来使用其中的一个子集。例如，不需要带过期时间的临时访问凭证，可以只使用RAM的RAM用户功能而无需使用STS。

17.2. 配置用户权限

阿里云权限管理机制包括访问控制和临时安全令牌，可以根据需求使用不同权限的RAM用户来访问表格存储，也支持为用户提供访问的临时授权。使用RAM和STS能极大地提高管理的灵活性和安全性。

背景信息

RAM和STS解决的一个核心问题是如何在不暴露主账号的AccessKey的情况下安全地授权其他人访问。因为主账号的AccessKey泄露会带来极大的安全风险，其他人可以随意操作该账号下所有的资源、盗取重要信息等。

- 访问控制（RAM）是阿里云提供的管理用户身份与资源访问权限的服务。

  RAM允许在一个阿里云账号下创建并管理多个身份，并允许给单个身份或一组身份分配不同的权限，从而实现不同用户拥有不同资源访问权限的目的。更多信息，请参见什么是访问控制。

- 阿里云临时安全令牌（Security Token Service，STS）是阿里云提供的一种临时访问权限管理服务。

  通过STS服务，您所授权的身份主体（RAM用户或RAM角色）可以获取一个自定义时效和访问权限的临时访问令牌。更多信息，请参见什么是STS。

RAM提供了一种长期有效的权限控制机制，通过分出不同权限的RAM用户，将不同的权限分给不同的用户，即使RAM用户的AccessKey泄露也不会造成全局的信息泄露。RAM用户一般情况也是长期有效的。因此RAM用户的AccessKey也不能泄露。

相对于RAM提供的长效控制机制，STS提供的是一种临时访问授权，通过STS可以获取临时的AccessKey和Token，这些信息可以直接发给临时用户用来访问表格存储。一般来说，从STS获取的权限会受到更加严格的限制，并且拥有时间限制，因此即使这些信息泄露，对于系统的影响也很小。

配置RAM用户权限

1. 创建RAM用户。具体操作，请参见创建RAM用户。
2. 为RAM用户授权，根据实际配置RAM用户的权限。具体操作，请参见为RAM用户授权。
   - 如果需要管理表格存储，例如创建实例等，请授予RAM用户AliyunOTSFullAccess权限。
   - 如果需要只读访问表格存储，例如读取表中数据等，请授予RAM用户AliyunOTSReadOnlyAccess权限。
   - 如果需要只写访问表格存储，例如创建数据表等，请授予RAM用户AliyunOTSWriteOnlyAccess权限。

   > **说明** 如果需要更精细的权限控制，您可以创建自定义权限策略，进行策略权限的配置。更多信息，请参见自定义权限。

3. 为RAM用户设置多因素认证。具体操作，请参见为RAM用户启用多因素认证。

配置临时用户权限

1. 创建临时角色及授权。
   i. 创建可信实体为阿里云账号的RAM角色。具体操作，请参见创建可信实体为阿里云账号的RAM角色。

      创建名称分别为RamTestAppReadOnly和RamTestAppWrite两个角色，RamTestAppReadOnly用于读取等操作，RamTestAppWrite用于上传文件的操作。

   ii. 创建自定义策略。具体操作，请参见创建自定义权限策略。

      > **说明** 如果需要更精细的权限控制，您可以自定义策略的权限。更多信息，请参见自定义权限。

      创建名称分别为ram-test-app-readonly和ram-test-app-write的两个策略。

- ram-test-app-readonly策略

```
{
"Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ots:BatchGet*",
        "ots:Describe*",
        "ots:Get*",
        "ots:List*"
      ],
      "Resource": [
        "acs:ots:*:*:instance/ram-test-app",
        "acs:ots:*:*:instance/ram-test-app/table/*"
      ]
    }
],
"Version": "1"
}
```

- ram-test-app-write策略

```
    {
 "Statement": [
        {
          "Effect": "Allow",
          "Action": [
            "ots:Create*",
            "ots:Insert*",
            "ots:Put*",
            "ots:Update*",
            "ots:Delete*",
            "ots:BatchWrite*"
          ],
          "Resource": [
            "acs:ots:*:*:instance/ram-test-app",
            "acs:ots:*:*:instance/ram-test-app/table/*"
          ]
        }
 ],
 "Version": "1"
    }
```

iii. 为临时角色授权。具体操作，请参见为RAM角色授权。

为ramtestappreadonly角色赋予ram-test-app-readonly（只读访问表格存储）策略，为ramtestappwrite角色赋予ram-test-app-write（只写表格存储）策略。

授权完成后，记录角色的**ARN**，即需要扮演角色的ID，如下图所示。

2. 临时授权访问。

i. 创建自定义策略。具体操作，请参见创建自定义权限策略。

> **说明** 如果需要更精细的权限控制，您可以自定义策略的权限。更多信息，请参见自定义权限。

创建名称分别为AliyunSTSAssumeRolePolicy2016011401和AliyunSTSAssumeRolePolicy2016011402的两个策略。其中Resource为角色的ARN信息。

- AliyunSTSAssumeRolePolicy2016011401

```
{
"Version": "1",
"Statement": [
    {
        "Effect": "Allow",
        "Action": "sts:AssumeRole",
        "Resource": "acs:ram:198***237:role/ramtestappreadonly"
    }
]
}
```

- AliyunSTSAssumeRolePolicy2016011402

```
{
"Version": "1",
"Statement": [
    {
        "Effect": "Allow",
        "Action": "sts:AssumeRole",
        "Resource": "acs:ram:198***237:role/ramtestappwrite"
    }
]
}
```

ii. 授权RAM用户临时角色。具体操作，请参见为RAM用户授权。

将自定义策略AliyunSTSAssumeRolePolicy2016011401和AliyunSTSAssumeRolePolicy2016011402授权给名称为ram_test_app的RAM用户。

3. 从STS获取的临时访问凭证。具体操作，请参见AssumeRole。
4. 使用临时授权读写数据。

您可以使用临时授权调用不同语言的SDK访问表格存储。Java SDK请参考以下方式创建OTSClient对象，传入从STS获取的AccessKeyId、AccessKeySecret和SecurityToken等参数。

```
OTSClient client = new OTSClient(otsEndpoint, stsAccessKeyId, stsAccessKeySecret, instanceName, stsToken);
```

17.3. 表格存储服务关联角色

使用表格存储数据湖投递功能时，需要具有OSS资源的访问权限，此时可以通过表格存储控制台自动创建表格存储服务关联角色AliyunServiceRoleForOTSDataDelivery获取OSS资源的访问权限。

> 说明
>
> 关于服务关联角色的更多信息，请参见服务关联角色。

创建服务关联角色

使用表格存储数据湖投递功能时，通过表格存储控制台可以自动创建表格存储服务关联角色AliyunServiceRoleForOTSDataDelivery。

表格存储服务关联角色AliyunServiceRoleForOTSDataDelivery对应的权限策略为AliyunServiceRolePolicyForOTSDataDelivery，支持的OSS操作为PutObject、AbortMultipartUpload、PutObjectTagging、GetObject和DeleteObjectTagging。

删除服务关联角色

请确保当前账号下所有实例均未使用数据湖投递功能，才能删除表格存储服务关联角色（AliyunServiceRoleForOTSDataDelivery）。

> 注意
>
> 删除表格存储服务关联角色后，当前账号下的数据将无法投递到OSS。

删除服务关联角色的操作步骤如下：

1. 登录RAM控制台。
2. 在左侧导航栏中，选择**身份管理>角色**。
3. 在**角色**页面的搜索框中，输入AliyunServiceRoleForOTSDataDelivery，自动搜索到名称为AliyunServiceRoleForOTSDataDelivery的RAM角色。
4. 在右侧**操作**列，单击**删除**。
5. 在确认对话框，单击**确定**。
   - 如果当前账号下有实例使用数据湖投递功能，将无法删除该角色。请删除实例下的投递任务后重试删除操作。
   - 如果当前账号下所有实例均未使用数据湖投递功能，则可直接删除该角色。

常见问题

为什么使用RAM用户无法自动创建表格存储服务关联角色（AliyunServiceRoleForOTSDataDelivery）？

只有拥有指定权限的用户，才能自动创建或删除表格存储服务关联角色。当RAM用户无法自动创建表格存储服务关联角色时，需要为RAM用户添加如下权限策略。

使用时请将*主账号ID*替换为实际的阿里云账号（主账号）ID。

```
{
    "Statement": [
        {
            "Action": [
                "ram:CreateServiceLinkedRole"
            ],
            "Resource": "acs:ram:*:
主账号ID
:role/*",
            "Effect": "Allow",
            "Condition": {
                "StringEquals": {
                    "ram:ServiceName": [
                        "arms.aliyuncs.com"
                    ]
                }
            }
        }
    ],
    "Version": "1"
}
```

17.4. 自定义权限

本文为您介绍Action、Resource和Condition的定义以及应用场景。

Action定义

Action是API的名称，可以根据Action设置开放或限制用户可以访问的API。在创建表格存储的授权策略时，每个Action都需要添加 `ots:` 前缀，多个Action以逗号分隔，并且支持星号通配符（包括前缀匹配和后缀匹配）。

以下是一些典型的Action定义：

- 单个API

  ```
  "Action": "ots:GetRow"
  ```

- 多个API

  ```
  "Action": [
  "ots:PutRow",
  "ots:GetRow"
  ]
  ```

- 所有只读API

```
{
  "Version": "1",
  "Statement": [
    {
      "Action": [
        "ots:BatchGet*",
        "ots:Describe*",
        "ots:Get*",
        "ots:List*",
        "ots:Consume*",
        "ots:Search",
        "ots:ComputeSplitPointsBySize"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}
```

- 所有读写API

```
"Action": "ots:*"
```

- 所有SQL操作API

```
"Action": "ots:SQL*"
```

Resource定义

表格存储的资源由产品、地域、用户ID、实例名和表名多个字段组成。每个字段支持星号通配符（包括前缀匹配和后缀匹配），Resource格式如下：

```
acs:ots:[region]:[user_id]:instance/[instance_name]/table/[table_name]
```

其中，[xxx]表示变量，产品固定为ots，地域为英文缩写（例如cn-hangzhou），用户ID为阿里云账号ID。

> **说明** 表格存储中实例名称不区分大小写，上述Resource资源定义中的[instance_name]请用小写表示。

Tunnel涉及的Resource定义只能到实例级别，而不是表级别，即Tunnel的资源由产品、地域、用户ID和实例名组成，Resource格式如下：

```
acs:ots:[region]:[user_id]:instance/[instance_name]
```

以下是一些典型的Resource定义：

- 所有地域的所有用户的所有资源

```
"Resource": "acs:ots:*:*:*"
```

- 华东1（杭州）地域，用户123456的所有实例及其下所有的表

```
"Resource": "acs:ots:cn-hangzhou:123456:instance/*"
```

- 华东1（杭州）地域，用户123456的名称为abc的实例及其下所有的表

```
"Resource": [
"acs:ots:cn-hangzhou:123456:instance/abc",
"acs:ots:cn-hangzhou:123456:instance/abc/table/*"
]
```

- 所有以abc开头的实例及下的所有表

```
"Resource": "acs:ots:*:*:instance/abc*"
```

- 所有以abc开头的实例下的所有以xyz开头的表（不包括实例资源，不匹配acs:ots:*:*:instance/abc*）

```
"Resource": "acs:ots:*:*:instance/abc*/table/xyz*"
```

- 所有以abc结尾的Instance及其下的所有以xyz结尾的表

```
"Resource": [
"acs:ots:*:*:instance/*abc",
"acs:ots:*:*:instance/*abc/table/*xyz"
]
```

表格存储的API类型

目前表格存储包含以下三类API:

- 实例管理类API
- 表和数据读写类API
- 实时通道管理和读写类API

各API的类别详情请参见下表。

- 管理类API访问的资源

  管理类API主要为实例相关的操作，仅由控制台调用。对该类API的Action和Resource定义，将影响用户使用控制台。以下访问的资源省略了 `acs:ots:[region]:[user_id]:` 前缀，只描述实例和表部分。

API名称/Action	访问的资源
ListInstance	instance/*
InsertInstance	instance/[instance_name]
GetInstance	instance/[instance_name]
DeleteInstance	instance/[instance_name]

- 数据类API访问的资源

  数据类API主要为表和行相关的操作，控制台和SDK都会调用，对该类API的Action和Resource定义，将影响用户使用控制台。以下访问的资源省略了 `acs:ots:[region]:[user_id]:` 前缀，只描述实例和表部分。

API名称/Action	访问的资源
ListTable	instance/[instance_name]/table/*

API名称/Action	访问的资源
CreateTable	instance/[instance_name]/table/[table_name]
UpdateTable	instance/[instance_name]/table/[table_name]
DescribeTable	instance/[instance_name]/table/[table_name]
DeleteTable	instance/[instance_name]/table/[table_name]
GetRow	instance/[instance_name]/table/[table_name]
PutRow	instance/[instance_name]/table/[table_name]
UpdateRow	instance/[instance_name]/table/[table_name]
DeleteRow	instance/[instance_name]/table/[table_name]
GetRange	instance/[instance_name]/table/[table_name]
BatchGetRow	instance/[instance_name]/table/[table_name]
BatchWriteRow	instance/[instance_name]/table/[table_name]
ComputeSplitPointsBySize	instance/[instance_name]/table/[table_name]
StartLocalTransaction	instance/[instance_name]/table/[table_name]
CommitTransaction	instance/[instance_name]/table/[table_name]
AbortTransaction	instance/[instance_name]/table/[table_name]
CreateIndex	instance/[instance_name]/table/[table_name]
DropIndex	instance/[instance_name]/table/[table_name]
CreateSearchIndex	instance/[instance_name]/table/[table_name]
DeleteSearchIndex	instance/[instance_name]/table/[table_name]
ListSearchIndex	instance/[instance_name]/table/[table_name]
DescribeSearchIndex	instance/[instance_name]/table/[table_name]
Search	instance/[instance_name]/table/[table_name]
CreateTunnel	instance/[instance_name]/table/[table_name]
DeleteTunnel	instance/[instance_name]/table/[table_name]
ListTunnel	instance/[instance_name]/table/[table_name]
DescribeTunnel	instance/[instance_name]/table/[table_name]

API名称/Action	访问的资源
ConsumeTunnel	instance/[instance_name]/table/[table_name]
BulkImport	instance/[instance_name]/table/[table_name]
BulkExport	instance/[instance_name]/table/[table_name]
SQL_Select	instance/[instance_name]/table/[table_name]
SQL_Create	instance/[instance_name]/table/[table_name]
SQL_DropMapping	instance/[instance_name]/table/[table_name]

- Tunnel API访问的资源

  Tunnel API主要为通道相关的操作，控制台和SDK都会调用，对该类API的Action和Resource定义，将影响用户使用控制台。以下访问的资源省略了 `acs:ots:[region]:[user_id]:` 前缀，只描述实例和表部分。

API 名称/Action	访问的资源
ListTable	instance/[instance_name]
CreateTable	instance/[instance_name]
UpdateTable	instance/[instance_name]
DescribeTable	instance/[instance_name]
DeleteTable	instance/[instance_name]
GetRow	instance/[instance_name]
PutRow	instance/[instance_name]
UpdateRow	instance/[instance_name]
DeleteRow	instance/[instance_name]
GetRange	instance/[instance_name]
BatchGetRow	instance/[instance_name]
BatchWriteRow	instance/[instance_name]
ComputeSplitPointsBySize	instance/[instance_name]
StartLocalTransaction	instance/[instance_name]
CommitTransaction	instance/[instance_name]
AbortTransaction	instance/[instance_name]
CreateIndex	instance/[instance_name]

API 名称/Action	访问的资源
DropIndex	instance/[instance_name]
CreateSearchIndex	instance/[instance_name]
DeleteSearchIndex	instance/[instance_name]
ListSearchIndex	instance/[instance_name]
DescribeSearchIndex	instance/[instance_name]
Search	instance/[instance_name]
CreateTunnel	instance/[instance_name]
DeleteTunnel	instance/[instance_name]
ListTunnel	instance/[instance_name]
DescribeTunnel	instance/[instance_name]
ConsumeTunnel	instance/[instance_name]

- 常见问题说明
  - Policy中Action和Resource通过字符串匹配进行验证的，并且星号通配符区分前缀和后缀匹配。如果Resource定义为acs:ots:*:*:instance/*/，则无法匹配acs:ots:*:*:instance/abc。如果Resource定义为 acs:ots:*:*:instance/abc，则无法匹配 acs:ots:*:*:instance/abc/table/xyz。
  - 登录表格存储控制台管理实例资源，需要授予用户acs:ots:[region]:[user_id]:instance/*资源的读取权限，因为控制台需要获取实例的列表。
  - 对于批量操作API（例如BatchGetRow和BatchWriteRow），后端服务会对被访问的每张表分别鉴权，只有所有表都通过鉴权才能执行操作，否则会返回权限错误。

Condition定义

目前Policy支持访问IP限制、是否通过HTTPS访问、是否通过MFA（多因素认证）访问和访问时间限制等多种鉴权条件，表格存储所有API都已经支持这些条件。

- 访问IP限制

  访问控制RAM可以限制访问表格存储的源IP地址，并且支持根据网段进行过滤。以下是一些典型的使用场景：

- 限制多个IP地址。例如只允许IP地址为10.101.168.111和10.101.169.111的请求访问。

```
{
"Statement": [
    {
        "Effect": "Allow",
        "Action": "ots:*",
        "Resource": "acs:ots:*:*:*",
        "Condition": {
            "IpAddress": {
                "acs:SourceIp": [
                    "10.101.168.111",
                    "10.101.169.111"
                ]
            }
        }
    }
],
"Version": "1"
}
```

- 限制单个IP地址和IP网段。例如只允许IP地址为10.101.168.111或10.101.169.111/24网段的请求访问。

```
{
"Statement": [
    {
        "Effect": "Allow",
        "Action": "ots:*",
        "Resource": "acs:ots:*:*:*",
        "Condition": {
            "IpAddress": {
                "acs:SourceIp": [
                    "10.101.168.111",
                    "10.101.169.111/24"
                ]
            }
        }
    }
],
"Version": "1"
}
```

- HTTPS访问限制

  访问控制可以限制是否通过HTTPS访问。

  以下是典型的使用场景，限制请求必须通过HTTPS访问。

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "ots:*",
            "Resource": "acs:ots:*:*:*",
            "Condition": {
                "Bool": {
                    "acs:SecureTransport": "true"
                }
            }
        }
    ],
    "Version": "1"
}
```

- MFA访问限制

  访问控制可以限制是否通过MFA（多因素认证）访问。

  以下是典型的使用场景，限制请求必须通过MFA访问。

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "ots:*",
            "Resource": "acs:ots:*:*:*",
            "Condition": {
                "Bool": {
                    "acs:MFAPresent ": "true"
                }
            }
        }
    ],
    "Version": "1"
}
```

- 访问时间限制

  访问控制可以限制请求的访问时间，即只允许或拒绝在某个时间点范围之前的请求。以下是典型的使用场景。

  例如北京时间2016年1月1日零点之前用户可以访问，之后就不能再访问。

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "ots:*",
            "Resource": "acs:ots:*:*:*",
            "Condition": {
                "DateLessThan": {
                    "acs:CurrentTime": "2016-01-01T00:00:00+08:00"
                }
            }
        }
    ],
    "Version": "1"
}
```

典型使用场景

结合以上对Action、Resource和Condition的定义，以下列出一些典型使用场景的Policy定义和授权方法。

- 多种授权条件

  对于访问IP地址为10.101.168.111/24网段的用户，可以对所有名称为online-01和online-02的实例执行读或者写操作（包括实例下的所有表），且要求只能在2016-01-01 00:00:00之前访问和通过HTTPS访问。

  操作步骤如下：

  i. 使用阿里云账号登录访问控制RAM管理控制台（默认已开通访问控制服务）。
  ii. 在左侧导航栏，选择**权限管理 > 权限策略管理**。
  iii. 在**权限策略管理**页面，单击**新建权限策略**。
  iv. 在创建权限策略页面，填写**策略名称**，选择**脚本配置**，并将如下内容填写到**策略内容**栏。

```
{
"Statement": [
    {
        "Effect": "Allow",
        "Action": "ots:*",
        "Resource": [
            "acs:ots:*:*:instance/online-01",
            "acs:ots:*:*:instance/online-01/table/*",
            "acs:ots:*:*:instance/online-02",
            "acs:ots:*:*:instance/online-02/table/*"
        ],
        "Condition": {
            "IpAddress": {
                "acs:SourceIp": [
                    "10.101.168.111/24"
                ]
            },
            "DateLessThan": {
                "acs:CurrentTime": "2016-01-01T00:00:00+08:00"
            },
            "Bool": {
                "acs:SecureTransport": "true"
            }
        }
    }
],
"Version": "1"
}
```

v. 单击**确定**。

vi. 在左侧导航栏，选择**身份管理 > 用户**。找到需要授权的RAM用户，单击其右侧操作栏下面的**添加权限**按钮，进入编辑个人授权页面。

vii. 搜索刚才新建的策略名称，将该策略添加至已选择授权策略名称栏中，然后单击**确定**，完成对该账号的策略授权。

- 拒绝请求

  对于访问IP地址为10.101.169.111的用户，拒绝对华北2（北京）地域，名称以online和product开头的实例下的所有表执行写操作（不包括对实例的操作）。

  创建自定义授权策略和为RAM用户授权的步骤同上，只需将如下内容填写到**策略内容**栏即可。

```
{
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "ots:Create*",
                "ots:Insert*",
                "ots:Put*",
                "ots:Update*",
                "ots:Delete*",
                "ots:BatchWrite*"
            ],
            "Resource": [
                "acs:ots:cn-beijing:*:instance/online*/table/*",
                "acs:ots:cn-beijing:*:instance/product*/table/*"
            ],
            "Condition": {
                "IpAddress": {
                    "acs:SourceIp": [
                        "10.101.169.111"
                    ]
                }
            }
        }
    ],
    "Version": "1"
}
```

18.常见问题

18.1. 多元索引路由字段的使用

在创建多元索引时可以选择部分主键列作为路由字段，在进行索引数据写入时，会根据路由字段的值计算索引数据的分布位置，路由字段的值相同的记录会被索引到相同的数据分区中。

使用方法

1. 在创建索引时，指定一个或多个路由字段。

   您在创建多元索引时指定了路由字段后，索引数据的读写都会使用该路由字段进行定位，不能动态改变。如果想使用系统默认路由（即主键列路由）或者重新指定其他字段为路由字段，您需要重建索引。

   > **说明** 路由字段只能是表格存储的主键列。

2. 在索引查询时，在查询请求中指定路由字段值。

   查询时使用路由，定向搜索指定数据分区，可以减少长尾对延迟的影响。对于自定义路由的查询请求，都要求用户提供路由字段。如不指定，虽然查询结果一样，但查询时会访问无关的数据分区，浪费系统资源，增加访问延迟。

示例代码

```
private static void testRoute(SyncClient client) throws InterruptedException {
        //创建表
        TableMeta meta = new TableMeta("order");
        meta.addPrimaryKeyColumn("order_id",PrimaryKeyType.STRING);
        meta.addPrimaryKeyColumn("user_id",PrimaryKeyType.STRING);
        TableOptions options = new TableOptions();
        options.setMaxVersions(1);
        options.setTimeToLive(-1);
        CreateTableRequest request = new CreateTableRequest(meta,options);
        request.setReservedThroughput(new ReservedThroughput(new CapacityUnit(0, 0)));
        CreateTableResponse response = client.createTable(request);
        //创建多元索引并指定路由字段
        CreateSearchIndexRequest searchIndexRequest = new CreateSearchIndexRequest();
        searchIndexRequest.setTableName("order"); //订单表
        searchIndexRequest.setIndexName("order_index");  //订单表索引名
        IndexSchema indexSchema = new IndexSchema();
        IndexSetting indexSetting = new IndexSetting();
        indexSetting.setRoutingFields(Arrays.asList("user_id"));//设置商户id为路由字段
        indexSchema.setIndexSetting(indexSetting);
        //添加索引字段 这里只是给出示例，您可以根据业务需求添加索引字段
        indexSchema.setFieldSchemas(Arrays.asList(
                new FieldSchema("product_name",FieldType.KEYWORD).setStore(true).setIndex(t
rue),
                new FieldSchema("order_time",FieldType.LONG).setStore(true).setEnableSortAn
dAgg(true).setIndex(true),
                new FieldSchema("user_id",FieldType.KEYWORD).setStore(true).setIndex(true)
        ));
        searchIndexRequest.setIndexSchema(indexSchema);
        client.createSearchIndex(searchIndexRequest);
```

```
        Thread.sleep(6*1000); // 等待数据表加载
        //插入一些测试数据
        String[] productName = new String[]{"product a", "product b", "product c"};
        String[] userId = new String[]{"00001", "00002", "00003", "00004", "00005"};
        for (int i = 0; i < 100; i++){
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder
();
            primaryKeyBuilder.addPrimaryKeyColumn("order_id",PrimaryKeyValue.fromString(i+"
"));
            primaryKeyBuilder.addPrimaryKeyColumn("user_id",PrimaryKeyValue.fromString(user
Id[i%(userId.length)]));
            PrimaryKey primaryKey = primaryKeyBuilder.build();
            RowPutChange rowPutChange = new RowPutChange("order",primaryKey);
            //写入属性列
            rowPutChange.addColumn("product_name",ColumnValue.fromString(productName[i%(pro
ductName.length)]));
            rowPutChange.addColumn("order_time",ColumnValue.fromLong(System.currentTimeMill
is()));
            rowPutChange.setCondition(new Condition(RowExistenceExpectation.IGNORE));
            client.putRow(new PutRowRequest(rowPutChange));
        }
        Thread.sleep(20*1000);//等待数据同步到多元索引
        //带上路由字段的查询
        SearchRequest searchRequest = new SearchRequest();
        searchRequest.setTableName("order");
        searchRequest.setIndexName("order_index");
        MatchQuery matchQuery = new MatchQuery();
        matchQuery.setFieldName("user_id");
        matchQuery.setText("00002");
        SearchQuery searchQuery = new SearchQuery();
        searchQuery.setQuery(matchQuery);
        searchQuery.setGetTotalCount(true);
        SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        columnsToGet.setReturnAll(true);
        searchRequest.setColumnsToGet(columnsToGet);
        searchRequest.setSearchQuery(searchQuery);
        PrimaryKeyBuilder pkbuild = PrimaryKeyBuilder.createPrimaryKeyBuilder();
        pkbuild.addPrimaryKeyColumn("user_id",PrimaryKeyValue.fromString("00002"));
        PrimaryKey routingValue = pkbuild.build();
        searchRequest.setRoutingValues(Arrays.asList(routingValue));
        SearchResponse searchResponse = client.search(searchRequest);
        System.out.println(searchResponse.isAllSuccess());
        System.out.println("totalCount:"+ searchResponse.getTotalCount());
        System.out.println("RowCount:"+searchResponse.getRows().size());
    }
```

18.2. 全局二级索引和多元索引的选择

本文主要对原生Tablestore查询、全局二级索引（Global Secondary Index）和多元索引（Search Index）三种查询场景进行详细分析。

详细分析参见Tablestore存储和索引引擎详解。

原生Tablestore

数据查询依赖主键，主要是通过主键点查询（GetRow），主键范围查询（GetRange）。如需对属性列进行查询，需要使用Filter功能，在数据量很大的时候效率不高，甚至变成全表扫描。在实际业务中，主键查询也常常不能满足需求，而使用Filter在大数据量时效率很低。Tablestore推出了全局二级索引和多元索引，这两个功能弥补了原生Tablestore查询方式单一的缺点，本文主要为您分析全局二级索引以及多元索引的区别及选择。

全局二级索引

主表建立全局二级索引后，相当于多了一张Tablestore表，所以索引表的模型与Tablestore表一致。索引表相当于给主表提供了另外一种排序方式，即对查询条件预先设计了一种数据分布，加快数据查询的效率，索引表的查询方式仍然是基于主键点查、主键范围查、主键前缀范围查询。为了确保主键的唯一性，全局二级索引会将主表的主键列也放到索引表中。

多元索引

多元索引相比以上两种，底层增加了倒排索引，多维空间索引等，支持多字段自由组合查询、模糊查询、地理位置查询、全文检索等，相比功能较为单纯的二级索引更加丰富，而且一个索引可以满足多种维度的查询，支持多种查询条件，因此命名为多元索引。

索引选择

- 不一定需要索引
  - 如果基于主键和主键范围查询的功能已经可以满足业务需求，那么不需要建立索引。
  - 如果对某个范围内进行筛选，范围内数据量不大或者查询频率不高，可以使用Filter，不需要建立索引。
  - 如果是某种复杂查询，执行频率较低，对延迟不敏感，可以考虑通过DLA（数据湖分析）服务访问Tablestore，使用SQL进行查询。
- 全局索引or多元索引

  一个全局二级索引是一个索引表，类似于主表，其提供了另一种数据分布方式，或者认为是另一种主键排序方式。一个索引对应一种查询条件，预先将符合查询条件的数据排列在一起，查询效率很高。索引表可支撑的数据规模与主表相同，此外，全局二级索引的主键设计也同样需要考虑散列问题。

  一个多元索引是一系列数据结构的组合，其中的每一列都支持建立倒排索引等结构，查询时可以按照其中任意一列进行排序。一个多元索引可以支持多种查询条件，不需要对不同查询条件建立多个多元索引。相比全局二级索引，也支持多条件组合查询、模糊查询、全文索引、地理位置查询等。多元索引本质上是通过各种数据结构加快了数据的筛选过程，功能丰富，但在数据按照某种固定顺序读取这种场景上，效率不如全局二级索引。多元索引的查询效率与倒排链长度等因素相关，即查询性能与整个表的全量数据规模有关，在数据规模达到百亿行以上时，建议使用RoutingKey对数据进行分片，查询时也通过指定RoutingKey查询来减少查询涉及到的数据量。简而言之，查询灵活度和数据规模不可兼得。

18.3. 使用通配符查询时出现length of field value is longer than 10 for the [WILDCARD_QUERY] query异常

本文介绍使用通配符查询时出现length of field value is longer than 10 for the [WILDCARD_QUERY] query异常的现象、原因和解决方案。

现象

使用通配符查询时出现如下异常：

```
length of field value is longer than 10 for the [WILDCARD_QUERY] query
```

原因

带有通配符的字符串长度超过10字节。

解决方案

使用通配符查询时，要匹配的值可以是一个带有通配符的字符串，带有通配符的字符串长度不能超过10字节。

18.4. 使用多元索引Search接口查不到数据

本文介绍使用多元索引Search接口查不到数据的现象、原因和解决方案。

现象

使用多元索引Search接口查询数据时，出现查不到数据问题。

原因

- 数据表中的数据未正确同步到多元索引。
  - 数据表中的数据以异步方式同步到多元索引，所以多元索引中的数据存在一定延迟，增量数据同步延迟时间大部分在10秒以内，全量数据同步延迟时间与数据表的数据量成正比。
  - 多元索引中的列名区分大小写，可能造成与数据表中的列名不匹配。例如数据表中列名为ColumnName，多元索引中的对应列名为columnname。
  - 多元索引中列的数据类型与数据表中的数据类型不匹配。例如数据表中某列为Integer，多元索引中对应列的数据类型为Keyword。

    > **说明** 表格存储的Wide column模型是schema-free的存储结构，同一个属性列的值可以有多种数据类型，所以可能出现部分行未同步成功的情况。

  - 多元索引中列的数据类型与数据表中的数据格式不匹配。例如数据表中某列数据类型为String，值为-91,100，多元索引中对应列的数据类型为Geopoint。

    > **说明** 多元索引中地理位置类型格式为"纬度,经度"，且纬度范围为-90～+90，经度范围-180～+180，且顺序不能写反。

- 使用的查询方式错误。
  - 多元索引中数据类型为Keyword的字段使用MatchQuery查询部分值。例如数据表中的数据为"abc"，多元索引中对应列的数据类型为Keyword，查询条件为MatchQuery("ab")，由于Keyword是不可分词类型，无法使用MatchQuery匹配部分值。
  - 查询条件中参数设置错误。例如数据表中的数据为"abc"，多元索引中对应列的数据类型为Keyword，查询条件TermQuery("ab")。

- 多元索引中数据类型为Keyword的字段使用RangeQuery查询时条件设置问题。例如数据表中的数据为20，多元索引中对应列的数据类型为Keyword，查询条件为RangeQuery(>10)。

  > **说明** RangeQuery中不同数据类型的比较规则为Keyword数据类型的值按照字典序比较，Long和Double数据类型的值按照大小比较。

- 分词器使用错误，例如数据表中的数据为"abcdefg"，多元索引中对应列的数据类型为Text且设置分词器为单字分词，查询条件为MatchQuery("abcd")。

解决方案

- 确保创建多元索引时设置的列名和数据类型与数据表的对应关系正确，写入数据表中的数据格式与多元索引中的数据类型匹配。更多信息，请参见数据类型映射或数组和嵌套类型。
- 确保数据表中数据已同步到多元索引。
- 确保使用多元索引时选择了合适的查询方式和设置了正确的查询条件。更多信息，请参见使用SDK。
- 确保使用了正确的分词器。更多信息，请参见分词。

18.5. 如何使用预定义列

预定义列是指在数据表上预先定义一些非主键列以及其类型。本文介绍如何使用预定义列。

> **注意** 单个数据表默认最大支持添加32个预定义列。如果不满足使用需求，请提交工单申请调整。

如果要使用二级索引，您需要通过如下方式为数据表添加预定义列。使用多元索引时，无需为数据表添加预定义列。

- 创建数据表时添加预定义列。具体操作，请参见创建数据表。
- 创建数据表后添加预定义列。具体操作，请参见Java SDK预定义列操作或者Go SDK预定义列操作。

为数据表添加预定义列后，创建二级索引时，您可以将预定义列作为二级索引的主键列或者属性列。二级索引的属性列必须为预定义列的一列或者多列。

假设数据表的主键包括pk1、pk2和pk3三列，预定义列包括col1和col2两列，属性列包括col3和col4两列。创建二级索引时，pk1、pk2和pk3只能作为二级索引的主键列，col1和col2可作为二级索引的主键列或者属性列，col3和col4不能用于二级索引。

18.6. 创建二级索引时报错Don't support allow update operation on table with index and ttl

当创建二级索引时出现该异常，请确保数据表存在固定数据生命周期（TTL）时数据表已禁止更新，或者数据表中数据永不过期。

现象

创建二级索引时出现如下异常：

```
ErrorCode: OTSParameterInvalid, ErrorMessage: Don't support allow update operation on table with index and ttl
```

原因

数据表存在固定数据生命周期时，未设置数据表为禁止更新状态（即TableOptions.AllowUpdate = false）。

> 说明
>
> 创建二级索引时，数据表必须满足如下条件中的任意一个。
>
> - 数据表的数据生命周期为-1（数据永不过期）。
> - 数据表的数据生命周期不为-1时，数据表为禁止更新状态。

解决方案

创建二级索引时，请根据实际业务需求选择合适的处理方式。

- 如果对数据表的数据生命周期无限制，您可以通过控制台或者SDK修改数据表的数据生命周期为-1。
  - 通过控制台修改

    在数据表的**基本详情**页签，单击**修改表属性**，修改数据表的数据生命周期为-1，单击**确定**。
  - 通过SDK修改

    调用UpdateTable接口修改数据表的数据生命周期（TableOptions.TimeToLive）参数为-1。
- 如果需要为数据表设置固定的数据生命周期，您可以通过SDK修改数据表为禁止更新状态。

  调用UpdateTable接口修改数据表的允许更新（TableOptions.AllowUpdate）参数为false。

  > 注意 设置数据表为禁止更新状态后，您不能通过UpdateRow对数据表中的数据进行更新。

18.7. 使用SQL查询数据时如何选择查询方式

使用SQL查询数据时，您可以通过表的映射关系或者多元索引的映射关系进行数据查询，请根据实际场景选择合适的查询方式。

背景信息

表格存储作为结构化大数据存储，支持不同的索引结构，便于在不同场景的查询分析加速使用。索引结构包括通过数据表主键进行的单行读（GetRow）和范围读（GetRange）、自定义配置主键的二级索引表以及支持倒排索引和多维空间索引的多元索引。

使用SQL查询功能时，您可以通过显式访问二级索引表查询数据。对于多元索引，表格存储提供了自动多元索引选择策略和显式访问多元索引两种方式查询数据。更多信息，请参见索引选择策略。

样例场景

假设数据表exampletable包括主键列id（整型）以及属性列name（String类型）和context（String类型），且已为该数据表创建了多元索引exampletable_searchindex，该多元索引包括id（Integer类型）和context（Text类型）。

本文以该样例场景为例介绍通过SQL使用不同索引结构进行数据查询的实践操作。

- 使用表的映射关系

- 使用多元索引的映射关系

使用表的映射关系

请根据实际场景选择合适的使用方式。使用表的映射关系时支持设置数据是否要满足强一致性以及是否允许牺牲统计聚合精准度来提升查询性能。

> **说明** 关于创建表的映射关系的更多信息，请参见创建表的映射关系。

方式一：创建表的映射关系后，查询数据时只要求数据满足最终一致即可，且允许牺牲统计聚合精准度

1. 为数据表exampletable创建映射关系，映射表名称为exampletable_test，其他参数保持默认即可。

```
CREATE TABLE `exampletable_test` (`id` BIGINT, `name` MEDIUMTEXT, `context` MEDIUMTEXT,
PRIMARY KEY(id)) ENGINE='Tablestore';
```

2. 查询数据。

   当执行以下SQL语句查询数据时，由于id、name和context未全部包括在多元索引exampletable_searchindex中，因此表格存储会自动选择数据表进行数据查询。

```
SELECT * FROM exampletable_test LIMIT 10;
```

   当执行以下SQL语句查询数据时，由于id和context均包括在多元索引exampletable_searchindex中，因此表格存储会自动选择该多元索引进行数据查询。

```
SELECT id,context FROM exampletable_test LIMIT 10;
```

方式二：创建表的映射关系后，查询数据时要求保证数据强一致性

1. 为数据表exampletable创建映射关系，映射表名称为exampletable_test以及设置data_consistency为strong。

```
CREATE TABLE `exampletable_test` (`id` BIGINT, `name` MEDIUMTEXT, `context` MEDIUMTEXT,
PRIMARY KEY(id)) ENGINE='Tablestore', ENGINE_ATTRIBUTE='{"data_consistency": "strong"}'
;
```

2. 查询数据。更多信息，请参见查询数据。

   由于多元索引是满足数据最终一致，并不保证强一致性，因此表格存储不会通过任何多元索引进行数据查询。

方式三：创建表的映射关系后，查询数据时只要求数据满足最终一致即可，但是不允许牺牲统计聚合精准度

1. 为数据表exampletable创建映射关系，映射表名称为exampletable_test，设置data_consistency为eventual以及设置allow_inaccurate_aggregation为false。

```
CREATE TABLE `exampletable_test` (`id` BIGINT, `name` MEDIUMTEXT, `context` MEDIUMTEXT,
PRIMARY KEY(id)) ENGINE='Tablestore', ENGINE_ATTRIBUTE='{"data_consistency": "eventual"
, "allow_inaccurate_aggregation": false}';
```

2. 查询数据。更多信息，请参见查询数据。

   由于多元索引的统计聚合操作非绝对精确的结果，因此表格存储不会选择任何多元索引进行数据查询。

使用多元索引的映射关系

当要使用指定多元索引进行数据查询时，您可以通过为该多元索引创建映射关系来实现。

> **说明** 关于创建多元索引的映射关系的更多信息，请参见创建多元索引的映射关系。

1. 创建多元索引的映射关系，映射表名称为exampletable_searchindex_test。

```
CREATE TABLE `exampletable_searchindex_test` (`id` BIGINT, `contex` MEDIUMTEXT DEFAULT NULL) ENGINE='searchindex' ENGINE_ATTRIBUTE={"index_name": "exampletable_searchindex", "table_name": "exampletable"}
```

2. 查询数据、

```
SELECT id,context FROM exampletable_searchindex_test WHERE text_match(context, "tablestore cool") LIMIT 10;
```

18.8. SQL查询常见错误排查

本文介绍使用SQL查询时的常见错误处理方法，主要包括OTSUnsupportOperation、OTSQuotaExhausted、OTSParameterInvalid和OtsRequestTimeout四种类型的错误。

错误码	错误信息	描述	解决办法
OTSUnsupportOperation	Operation not supported	表格存储SQL目前覆盖部分SQL功能集合。当使用未支持的SQL语法时，系统会报错。	使用表格存储SQL支持的功能。 SQL功能会快速迭代，最新功能集合请以官网文档为准。更多信息，请参见SQL支持功能说明。 如果您有特定SQL语法的使用需求，请联系表格存储技术支持。
	text_match is only supported for TEXT field in filter conditions in search index	全文检索函数使用错误。全文检索函数（text_match和text_match_phrase）必须使用在多元索引中的TEXT类型字段，并且只能出现在过滤条件中。	确保正确的使用全文检索功能。更多信息，请参见全文检索。
	text_match_phrase is only supported for TEXT field in filter conditions in search index		

错误码	错误信息	描述	解决办法
OTSQuotaExhausted	The sql scanned rows of kv exceeds the quota! Search quota is 100000 rows, table quota is 100000 rows	选择KV引擎作为查询引擎时扫描到的行数或者数据量超过限制。	• 通过多元索引加速查询。 如果查询中包括聚合函数（例如count、sum、avg）、group by或者基于非主键的过滤查询，建议配置多元索引进行查询加速。 如果已配置多元索引仍出现该错误，建议检查多元索引是否包含所有查询时用到的字段。 • 如果未配置多元索引，请确保查询条件包含主键列且符合最左匹配原则，并为SQL语句添加limit参数来控制返回的行数。 **说明** 目前扫描最大行数为100000行，扫描最大数据量为128 MB，扫描最大时间为30秒。
	The sql scanned rows of search exceeds the quota! Search quota is 100000 rows, table quota is 100000 rows	选择多元索引引擎作为查询引擎时扫描到的行数或者数据量超限。	• 为SQL语句添加limit参数来控制返回的行数。 • 优化SQL语句中的查询条件，将部分SQL计算任务下推到多元索引执行。关于计算下推的更多信息，请参见计算下推。
	The sql duration time exceed the quota! Quota is 30 seconds	SQL运行超时。	• 通过多元索引加速查询。 如果查询中包括聚合函数（例如count、sum、avg）、group by或者基于非主键的过滤查询，建议配置多元索引进行查询加速。 如果已配置多元索引仍出现该错误，建议检查多元索引是否包含所有查询时用到的字段。 • 如果未配置多元索引，请确保查询条件包含主键列且符合最左匹配原则，并为SQL语句添加limit参数来控制返回的行数。

错误码	错误信息	描述	解决办法
OTSParameterInvalid	Field type mismatch, actual: STRING, expect: INTEGER, col: xxx, primary key: {"PrimaryKeys":[{"ColumnName":"xxx","Value":"xxx","PrimaryKeyOption":0}]}	数据表中属性列的数据类型和SQL中属性列的数据类型不匹配。	表格存储数据表是Free-Schema的，允许在同一个属性列中写入多种类型的数据，但是SQL是Strong-Schema的，每个字段必须具有特定的类型定义。 使用SQL时，请确保数据表中属性列的数据类型与SQL中属性列的数据类型相匹配。您可以根据报错信息修改数据表中指定行对应属性列的数据类型。
	Table 'instancename.tablename' doesn't exist	创建映射关系时，指定的表不存在。	SQL中的Create Table语句只能为已存在的表或者多元索引创建映射关系。请确保要创建映射关系的表或者多元索引存在。更多信息，请参见创建表的映射关系和创建多元索引的映射关系。
	Field type 'DECIMAL(11,0)' is not supported	SQL不支持数据类型DECIMAL。	SQL查询只支持部分数据类型。更多信息，请参见数据类型映射。
	Search engine: length of field value is longer than 32 for the [WILDCARD_QUERY] query	like条件的字符长度超过32个。	减少like条件的字符个数。
	Expression #2 of SELECT list is not in GROUP BY clause and contains nonaggregated column	使用GROUP BY分组查询时，GROUP BY中未包含聚合函数。	在SQL语句中添加聚合条件。更多信息，请参见查询数据。
	Offset + limit exceeds the quota! quota is 150000 rows	offset+limit超过最大限制。使用limit a,b表达式时，a+b超过最大限制。最大限制为150000。	确保offset+limit之和小于等于150000。 如果有更大offset+limit的使用需求，请提交工单或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。
OtsRequestTimeout	Search engine: search timeout, please retry	此次查询访问了多元索引且单次查询运行超时。	查看使用的查询特性是否需要在多元索引中配置预定义规则。例如要使用order by功能，您可以在创建多元索引时配置预排序。更多信息，请参见创建多元索引。

19.附录

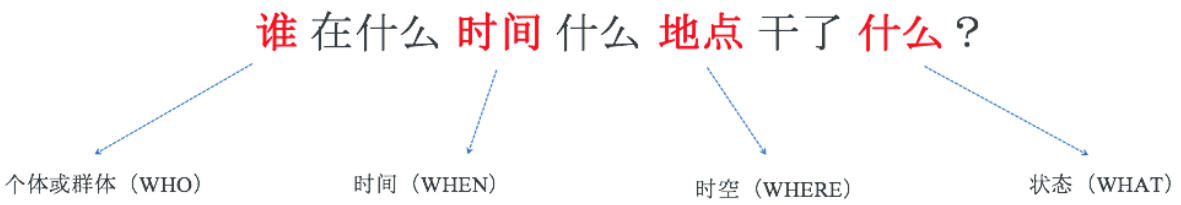
19.1. Timestream模型

19.1.1. 模型介绍

Timestream是针对时序场景设计的模型，本文主要为您介绍Timestream模型及其应用。

抽象模型

您可以通过Timestream的抽象模型对Timestream进行初步了解。



组成部分	描述
个体或群体（WHO）	描述产生数据的物体，可以是人、监控指标或者物体。个体或群体会有多维的属性，某一类唯一ID可以定位到个体，例如身份ID定位到人、设备ID定位到设备。维属性可以定位到个体，例如通过集群、机器ID、进程名来定位到某个进程。
时间（WHEN）	时间是时序数据最重要的特征，是区别于其他数据的关键属性。
时空（WHERE）	时空通常是通过纬度二维坐标定位到地点。科学计算领域（例如气象）通过经纬度和高度三维坐标来定位。
状态（WHAT）	用于描述特定个体在某一刻的状态，监控类时序数据通常是数值类型描述状态，轨迹数据是通过事件表述状态，不同场景有不同的表述方式。

完整模型

时序数据包含元数据和数据点两个部分。

- 元数据：由Name、Tags以及Attributes组成。Name+Tags可以唯一确定某个元数据。
- 数据点：由Timestamp、Location以及Fields组成。

元素	描述
Name	定义数据的类别
Tags	唯一标识个体的元数据
Attributes	个体的可变属性
Timestamp	数据产生的时间戳
Location	数据产生的空间信息
Fields	数据对应的值或状态，可提供多个值或状态，非一定是数值类型

应用案例

本案例通过事件类的物流轨迹场景展示Timestream数据模型是如何使用的。

元数据

Identifier

Name	Tags	Attributes	Timestamp	Fields	
				位置	状态
韵达	ID: 383449314xxxx	寄件地址：嘉兴 收件地址：杭州 当前位置： 30.069,120.322	1553682759	30.742,120.735 (嘉兴)	已发货
			1553743312	30.423,120.303 (杭州余杭临平公司)	运输中
			1553781886	30.069，120.322 (西湖区)	已签收

某快递物流轨迹

上图是一个快递的物流轨迹数据，记录的是快递在不同时间点的状态变化。该轨迹数据的元数据是快递本身，包含了单号、物流平台、快递当前位置信息以及快递寄件/收件等元信息，其中单号以及物流平台的组合（Identifier）唯一确定这个快递。

数据存储方式分析

- 将物流平台作为Name进行存储，一个快递平台的数据属于同一类数据，对数据检索性能有一定的提升。
- 将快递单号作为Tags存储，唯一确定一个快递。
- 快递的其他元信息存储在Attributes中，避免Tags过长导致的性能问题，同时能够支持这些信息的修改。
- 将快递的当前位置也存储在Attributes中，可以实现根据某个位置检索当前时间附近的快递。
- 快递的轨迹时序数据（位置/状态）存储在Fields中，可以查询某个快递在某个时间范围内的轨迹。

19.1.2. 快速入门

本文主要为您介绍如何通过示例代码快速使用Timestream模型。

操作步骤

1. 登录表格存储控制台，并创建表格存储实例。详情参见创建实例。
2. 下载并安装表格存储Java SDK包，详情参见安装。
3. 使用实例服务地址及账户密钥初始化对接实例，详情参见初始化。
4. 通过示例代码快速使用Timestream模型。

> **说明** 示例代码中，ApiService调用Timestream的API，更多关于ApiService，参见ApiService。

如果您希望了解更多关于Timestream模型的操作，参见Timestream SDK。

19.1.3. 基础操作

19.1.3.1. 概述

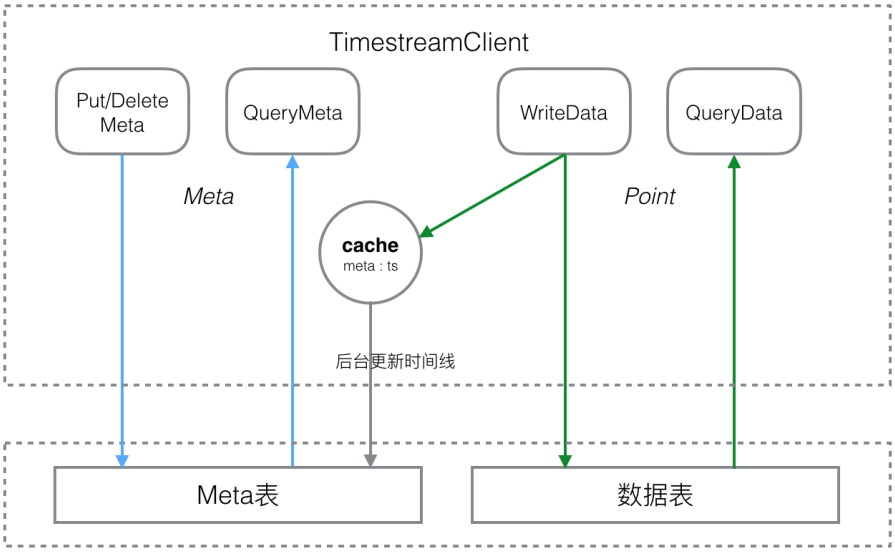
Timestream模型是针对时序场景设计的模型，Timestream模型的Java SDK包含以下操作。

- 初始化
- 表操作
- 写入

- 查询

19.1.3.2. 初始化

TimestreamDBClient是Timestream的客户端，您可以使用TimestreamDBClient进行建表、删表以及读写数据/时间线等操作。



其中时序数据是存储至数据表中，时间线是存储至元数据表（Meta表）中。数据表可以根据业务需求创建多个，但元数据表只能有一个。所有数据表的时间线元数据写入到同一个元数据表中，您通过TimestreamDBConfiguration配置元数据表的表名。 伴随数据的写入，后台默认不断更新时间线的lastUpdateTime，您可以通过设置TimestreamDBConfiguration中的dumpMeta来控制是否需要打开后台更新时间线，以及设置intervalDumpMeta来控制时间线更新的频率。

示例代码

1. 使用默认Writer配置创建TimestreamDBClient。

```
AsyncClient asyncClient = new AsyncClient(endpoint, accessKeyId, accessKeySecret, instance);
// 设置元数据表的表名
TimestreamDBConfiguration conf = new TimestreamDBConfiguration("metaTableName");
// 设置后台更新时间线的最大间隔
conf.setIntervalDumpMeta(10, TimeUnit.MINUTES);
TimestreamDBClient db = new TimestreamDBClient(asyncClient, conf);
```

2. 使用自定义Writer配置创建TimestreamDBClient。

```
// 自定义TableStoreWriter callback实现
class DefaultCallback implements TableStoreCallback<RowChange, ConsumedCapacity> {
  private AtomicLong succeedCount = new AtomicLong();
  private AtomicLong failedCount = new AtomicLong();
  public void onCompleted(final RowChange req, final ConsumedCapacity res) {
    succeedCount.incrementAndGet();
  }
  public void onFailed(final RowChange req, final Exception ex) {
    System.out.println("Got error:" + ex.getMessage());
    dirtyLog.info(gson.toJson(req));
    failedCount.incrementAndGet();
  }
}
AsyncClient asyncClient = new AsyncClient(endpoint, accessKeyId, accessKeySecret, insta
nce, clientConfiguration);
// 设置元数据表的表名
TimestreamDBConfiguration conf = new TimestreamDBConfiguration("metaTableName");
DefaultCallback callback = new DefaultCallback();
//TableStoreWriter配置
WriterConfig writerConfig = new WriterConfig();
// 设置writer的buffer为4096行
writerConfig.setBufferSize(4096);
TimestreamDBClient db = new TimestreamDBClient(asyncClient, config, new WriterConfig(),
callback);
```

19.1.3.3. 表操作

Timestream数据存储包含元数据表和数据表。数据表可以有多个，您可以根据自身的场景需求将数据写入到不同的表中，例如按数据精度分表。元数据表只能有一个，所有数据表的元数据信息全部记录到同一张表中。Timestream提供了元数据表和数据表的创建和删除功能。

元数据表

创建元数据表时可以预定义需要建索引的Attributes以及对应的索引属性和类型。索引类型支持LONG、DOUBLE、BOOLEAN、KEYWORD、GEO_POINT等类型，属性包含Index、Store和Array，其含义与多元索引相同。

> **说明** 没有指定索引类型的Attributes不会创建索引，后续时间线检索时也就不能指定这类Attributes作为查询条件。

操作步骤及示例代码如下。

1. 创建不包含任何Attributes索引的元数据表。

```
        db.createMetaTable();
```

2. 创建指定Attributes索引的元数据表。

```
        List<AttributeIndexSchema> indexSchemas = new ArrayList<AttributeIndexSchema>()
;
        indexSchemas.add(new AttributeIndexSchema("owner", AttributeIndexSchema.Type.KE
YWORD).setIsArray(true));
        indexSchemas.add(new AttributeIndexSchema("number", AttributeIndexSchema.Type.L
ONG));
        indexSchemas.add(new AttributeIndexSchema("score", AttributeIndexSchema.Type.DO
UBLE));
        indexSchemas.add(new AttributeIndexSchema("succ", AttributeIndexSchema.Type.BOO
LEAN));
        indexSchemas.add(new AttributeIndexSchema("loc", AttributeIndexSchema.Type.GEO_
POINT));
        db.createMetaTable(indexSchemas);
```

3. 删除元数据表。

```
        db.deleteMetaTable();
```

数据表

操作步骤及示例代码如下。

1. 创建数据表。

```
        db.createDataTable("datatable");
```

2. 删除数据表。

```
        db.deleteDataTable("datatable");
```

19.1.3.4. 写入

Timestream提供了数据点和元数据的写入接口。

数据点写入

Timestream提供了同步和异步两种数据点写入方式。其中异步接口的底层是通过TableStoreWriter来写入，其写入吞吐能力更高，对延时不是特别敏感的业务建议使用异步接口。

元数据写入

Timestream支持元数据同步写入。Timestream支持写入和删除元数据。

> **说明** 写入时间线元数据会覆盖该时间线元数据的原有数据。

示例代码

1. 写入数据点。

```
        // 构造时间线标示
        TimestreamIdentifier identifier = new TimestreamIdentifier.Builder("cpu")
                .addTag("Role", "OTSServer")
                .build();
        // 构造数据点，支持多个值
        Point point = new Point.Builder(System.currentTimeMillis(), TimeUnit.MILLISECON
DS)
                .addField("a1", 1)
                .addField("a2", true)
                .addField("a3", 12.0)
                .addField("a4", "value")
                .build();
        // 数据表操作对象
        TimestreamDataTable dataTable = db.dataTable(dataTableName);
        // 同步写入数据
        try {
          dataTable.write(
                  meta.getIdentifier(),
                  point);
        } catch (TableStoreException e) {
            // 异常处理
        }
        // 异步写入数据
        dataTable.asyncWrite(
                meta.getIdentifier(),
                point);
```

2. 写入时间线元数据。

```
        // 构造时间线标示
        TimestreamIdentifier identifier = new TimestreamIdentifier.Builder("cpu")
                .addTag("Role", "OTSServer")
                .build();
        // 构造时间线元数据
        TimestreamMeta meta = new TimestreamMeta(identifier).addAttribute("a1", "");
        TimestreamMetaTable metaTable = db.metaTable();
        // 同步写入时间线元数据
        metaTable.put(meta);
```

3. 删除时间线元数据。

```
        // 构造时间线标示
        TimestreamIdentifier identifier = new TimestreamIdentifier.Builder("cpu")
                .addTag("Role", "OTSServer")
                .build();
        TimestreamMetaTable metaTable = db.metaTable();
        // 同步删除时间线元数据
        metaTable.delete(identifier);
```

19.1.3.5. 查询

Timestream提供了数据点和元数据的查询接口。

元数据查询

元数据查询有两种方式：

- 时间线元数据检索，可以从Name、Tags、Attributes以及lastUpdateTime（数据最近更新时间）等维度进行过滤查询，并且支持翻页查询。
- 对指定时间线元数据进行精确查询。

示例代码

1. 多维度组合查询，只返回identifier。

```
        // 多条件组合查询，这里只是部分查询方式的示例，更多使用方式请参考SDK类方法
        Filter filter = and(
                Name.equal("cpu"),                                          // name查询条件，name等于cpu
                Tag.equal("cluster", "AY45"),                               // tag中cluster字段为AY45X
                Tag.notEqual("role", "Server"),                             // tag中role字段为OTSServer
                or(                                                         // attribute中status字段为Online或者Broken
                        Attribute.equal("status", "Online"),
                        Attribute.equal("status", "Broken")
                ),
                Attribute.notIn("machine", new String[]{"m1", "m2"}),       // attribute中machine是在[m1, m2]中
                Attribute.prefix("machine", "m"),                           // attribute中machine的前缀是m
                LastUpdateTime.in(TimeRange.range(1000, 2000, TimeUnit.MILLISECONDS)),  //时间线的最近更新时间为[1000, 2000)
                Attribute.inGeoBoundingBox("loc", "123,456", "234,567")             // loc在矩阵范围内
        );
        TimestreamMetaTable metaTable = db.metaTable();
        TimestreamMetaIterator iter1 = db.metaTable()
                .filter(filter)
                .identifierOnly()       // 只返回identifer
                .fetchAll();
        System.out.print(iterator.getTotalCount());   //获取命中的时间线条数
        while (iter1.hasNext()) {
            System.out.print(iterator.next().toString());
        }
```

2. 指定Name。翻页查询会返回完整的时间线元数据。

```
            // 查询name为cpu的所有时间线
            Filter filter = Name.equal("cpu");
            TimestreamMetaTable metaTable = db.metaTable();
            Iterator<TimestreamMeta> iterator = metaTable.filter(filter)
                    .offset(2)  // 设置offset为2
                    .fetchAll();
```

3. 查询指定的Attributes。

```
            // 查询name为cpu的所有时间线
            Filter filter = Name.equal("cpu");
            TimestreamMetaTable metaTable = db.metaTable();
            Iterator<TimestreamMeta> iterator = metaTable.filter(filter)
                    .selectAttributes("machine", "status")
                    .fetchAll();
```

4. 精确查询单条时间线元数据。

```
            TimestreamIdentifier identifier = new TimestreamIdentifier.Builder("cpu")
                    .addTag("cluster", "AY45")
                    .build();
            TimestreamMetaTable metaTable = db.metaTable();
            TimestreamMeta meta = metaTable.get(identifier).fetch();
```

数据查询

数据查询目前支持单条时间线的某个时间范围以及准确时间点的数据查询。

示例代码

1. 查询某个时间范围内的所有数据。

```
            TimestreamIdentifier identifier = new TimestreamIdentifier.Builder("cpu")
                    .addTag("cluster", "AY45")
                    .build();
            TimestreamDataTable dataTable = db.dataTable("data_table");
            Iterator<Point> iter = dataTable.get(identifier)
                    .timeRange(TimeRange.range(0, 10000, TimeUnit.MILLISECONDS))    // 查询[0, 10000)范围内的数据
                    .fetchAll();
```

2. 指定列名查询某个时间范围内的所有数据。

```
            TimestreamIdentifier identifier = new TimestreamIdentifier.Builder("cpu")
                    .addTag("cluster", "AY45")
                    .build();
            TimestreamDataTable dataTable = db.dataTable("data_table");
            Iterator<Point> iter = dataTable.get(identifier)
                    .select("load1", "load5", "load15", "error")    // 查询load1, load5, load15, error这四列
                    .timeRange(TimeRange.range(0, 10000, TimeUnit.MILLISECONDS))    // 查询[0, 10000)范围内的数据
                    .fetchAll();
```

3. 查询某个时间点的数据。

```
            TimestreamIdentifier identifier = new TimestreamIdentifier.Builder("cpu")
                    .addTag("cluster", "AY45")
                    .build();
            TimestreamDataTable dataTable = db.dataTable("data_table");
            Iterator<Point> iter = dataTable.get(identifier)
                    .select("load1", "load5", "load15", "error")    // 查询load1，load5，
load15，error这四列
                    .timestamp(10000, TimeUnit.SECONDS)         // 查询时间为10000的数据
点
                    .fetchAll();
```

19.1.3.6. Timestream模型限制

本文主要为您介绍Timestream模型的使用限制。

限制项	说明
Name长度	最大值：100Byte
Tag个数	最大值：12
Tag名字和值的总长度	最大值：500Byte
Tag名	字符中不能包含等号（=）
Attribute名	“h”、“n”、“s”、“t”为保留字段