阿里云
表格存储

开发指南

文档版本：20190716

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
	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。	 **禁止:** 重置操作将丢失用户配置数据。
	该类警示信息可能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，恢复业务所需时间约10分钟。
	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。	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。	单击 确定。
`courier` 字体	命令。	执行 `cd /d C:/windows` 命令，进入Windows系统文件夹。
*## *	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。	`swich` *{stand | slave}*

目录

1 概述

表格存储（Table Store）是阿里云自研的NoSQL多模型数据库，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提供PB级存储、千万TPS以及毫秒级延迟的服务能力。本手册主要为您介绍表格存储的基础概念、模型以及功能。

基础概念

使用表格存储前，请先了解以下基础概念：

基础概念	描述
实例	实例是您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。
读写吞吐量	读/写吞吐量的单位为读服务能力单元和写服务能力单元，简称CU，是数据读写操作的最小计费单位。
地域	地域是指阿里云物理数据中心所在的位置。
服务地址	每个表格存储实例对应一个服务地址，应用程序在进行表和数据操作时需要指定服务地址。

模型

表格存储提供多种模型，您可以针对业务需求选择相应的模型进行应用。表格存储提供以下几种模型：

模型	描述
Wide column模型	Wide column模型可应用在元数据、大数据等多种场景。 支持多种功能，包括数据版本、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。
Timeline模型	Timeline模型是针对消息数据场景所设计的数据模型，它能满足消息数据场景对消息保序、海量消息存储、实时同步的特殊需求，同时支持全文检索与多维度组合查询。可以同时应用在IM、Feed流等消息场景的实现上。
Timestream模型	Timestream模型是针对时序场景设计的模型。
Grid模型	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，可以帮助您方便地实现多维网格数据的存储、查询和管理。

功能

表格存储提供以下功能：

功能	描述
主键列自增	若设置一列主键为自增列，在写入一行数据时，这一列主键无需填值，表格存储会自动生成这一主键列的值。该值在分区键上保证唯一，且严格递增。
使用条件更新	条件更新功能只有在满足条件时才对表中的数据进行更改，当不满足条件时更新失败。
局部事务	使用局部事务，您可以创建一个范围不超过一个分区键值的事务，并在该事务内进行读写操作。
原子计数器	原子计数器是将列当成一个原子计数器来使用，便于为某些在线应用提供实时统计功能，比如统计帖子的PV（实时浏览量）等。
使用过滤器	过滤器可以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。由于只返回了符合条件的数据行，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，解决大数据的复杂查询难题。
全局二级索引	全局二级索引支持在属性列创建索引。
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。通道服务可以简单地实现对表中历史存量和新增数据的消费处理。
HBase支持	开源HBase API的Java应用可以通过Table Store HBase Client来直接访问表格存储服务。

2 使用限制

本文主要为您介绍表格存储的使用限制。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	如有需求提高上限，请提交工单。
单实例中表的个数	64	如有需求提高上限，请提交工单。
实例名称长度	3-16 Bytes	实例名称需由 [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（-）。 实例名称不能包含[‘ali’，‘ay’，‘ots’，‘taobao’，‘admin’] 这几个单词。

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	0-5000	如有需求提高上限，请提交工单。

列限制

资源	限制值	说明
列名长度限制	1-255 Bytes	字符集为 [a-z, A-Z, 0-9]和下划线（_），首字符必须是字母或下划线（_）。
主键包含的列数	1-4	至少 1 列，至多 4 列。

资源	限制值	说明
String 类型主键列列值大小	1 KB	单一主键列 String 类型的列列值大小上限 1 KB。
String 类型属性列列值大小	2 MB	单一属性列 String 类型的列列值大小上限 2 MB。
Binary 类型主键列列值大小	1 KB	单一主键列 Binary 类型的列列值大小上限 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	不限制	无
单行数据大小	不限制	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	PutRow、UpdateRow 或 BatchWriteRow 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	读请求一行数据中获取的列的最大个数。
表级别操作 QPS	10	一个实例的表级别操作每秒不超过 10 次，表级别操作见表操作。
单表 UpdateTable 的次数	上调：无限制，下调：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/写能力值。
BatchGetRow 一次操作请求读取的行数	100	无
BatchWriteRow 一次操作请求写入行数	200	无

操作	限制值	说明
BatchWriteRow 一次操作的数据大小	4 MB	无
GetRange 一次返回的数据	5000 行或者 4 MB	一次返回数据的行数超过 5000 行，或者返回数据的数据大小大于 4 MB。满足以上任一条件时，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

3 基础概念

3.1 实例

实例（Instance）是您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。

开通表格存储服务后，您需要通过管理控制台来创建实例，然后在实例内进行表的创建和管理。

每个云账号最多创建 10 个实例，每个实例内最多创建 64 张表。如果您需要增加限额，请提交工单。

规格

表格存储支持两种实例规格：高性能实例，容量型实例。两种规格都能够支持单表 PB 级别的数据量，主要区别在于使用成本及适用场景，具体说明如下：

> **注意:**
> 创建实例时请谨慎选择规格类型，创建后无法修改。

类型/对比项	使用场景	计费类型	读性能	写性能	并发力
高性能实例	适用于对读写性能和并发都要求非常高的场景，例如游戏、金融风控、社交应用、推荐系统等。	· 预留读/写吞吐量 · 按量读/写吞吐量	高	高	高

类型/对比项	使用场景	计费类型	读性能	写性能	并发力
容量型实例	适用于对读性能不敏感，但对成本较为敏感的业务，例如日志监控数据、车联网数据、设备数据、时序数据、物流数据、舆情监控等。	按量读/写吞吐量	中	高	中

各区域实例规格支持情况

地域名称	高性能实例	容量型实例
华东 1	支持	支持
华东 1 金融云	支持	暂不支持
华东 2	支持	支持
华东 2 金融云	暂不支持	支持
华北 2	支持	支持
华北 3	暂不支持	支持
华北 5	暂不支持	支持
华南 1	支持	支持
香港	暂不支持	支持
亚太东南 1（新加坡）	支持	暂不支持
美国东部 1（弗吉尼亚）	支持	暂不支持
美国西部 1（硅谷）	支持	暂不支持
亚太东北 1（东京）	暂不支持	支持
欧洲中部 1（法兰克福）	暂不支持	支持
中东东部 1（迪拜）	暂不支持	支持
亚太东南 2（悉尼）	暂不支持	支持
亚太东南 3（吉隆坡）	暂不支持	支持
亚太东南 5（雅加达）	暂不支持	支持
亚太南部 1（孟买）	暂不支持	支持

命名规范

实例的名称在一个地域内必须唯一，不同的地域内实例名称可以相同。具体命名规范如下：

- 必须由英文字母、数字或连字符（-）组成。
- 首字符必须为英文字母。
- 末尾字符不能为连字符（-）。
- 大小写不敏感。
- 长度在 3 Byte – 16 Byte 之间。
- 实例名称不能包含 [‘ali’，‘ay’，‘ots’，‘taobao’，‘admin’] 这几个单词。

3.2 服务地址

每个表格存储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。

不同访问场景下表格存储的服务地址格式如下：

说明：
各个地域（region）对应的英文表示参考地域。

- 从公网访问表格存储

  服务地址格式：

  ```
  https://instanceName.region.ots.aliyuncs.com
  ```

  例如，华东 1 节点，实例名称为 myInstance 的服务地址为：

  ```
  https://myInstance.cn-hangzhou.ots.aliyuncs.com
  ```

- 从同区域经典网络的 ECS 服务器访问表格存储

  应用程序从同区域的经典网络 ECS 服务器上通过内网访问表格存储，可以获得更低的响应延迟，且不产生外网流量。

  服务地址格式：

  ```
  https://instanceName.region.ots-internal.aliyuncs.com
  ```

  例如，华东 1 节点，实例名称为 myInstance 的服务地址为：

  ```
  https://myInstance.cn-hangzhou.ots-internal.aliyuncs.com
  ```

- 从 VPC 网络的 ECS 服务器访问表格存储

  服务地址格式：

  ```
  https://vpcName-instanceName.region.vpc.ots.aliyuncs.com
  ```

  例如，华东 1 节点，应用程序从名称为 testVPC 的 VPC 网络访问实例名称为 myInstance 的服务地址为：

  ```
  https://testVPC-myInstance.cn-hangzhou.vpc.ots.aliyuncs.com
  ```

  该 VPC 访问地址只支持来自于 testVPC 网络内的服务器的访问。

3.3 读写吞吐量

读/写吞吐量的单位为读服务能力单元和写服务能力单元，简称CU （Capacity Unit），是数据读写操作的最小计费单位。应用程序通过API进行表格存储读写操作时，会消耗对应的写服务能力单元和读服务能力单元。

- 1单位读能力表示从数据表中读一条4KB数据。
- 1单位写能力表示向数据表写一条4KB数据。
- 操作数据大小不足4KB的部分向上取整，如写入7.6KB数据消耗2单位写能力，读出0.1KB数据消耗1单位读能力。

预留读/写吞吐量

预留读/写吞吐量是表的一个属性。应用程序在创建表的时候，可以为该表指定预留读/写吞吐量。当设置的预留吞吐量大于0时，表格存储会为表分配和预留相应的资源，应用程序每秒不超过预留吞吐量的访问将会按照预留吞吐量的单价进行计费。由于预留读/写吞吐量在单价上低于按量读/写吞吐量，配置合适的预留读/写吞吐量可以进一步降低成本。

例如，刚建表之后如果需要导入大量数据，可以设置较大的预留写吞吐量，能够以较低的写成本将数据导入进来，当数据导入完毕后，再将预留读/写吞吐量下调。

说明：

- 预留读/写吞吐量可以设置为0。
- 不存在的表将被视作预留读/写吞吐量均为0，访问不存在的表将根据操作类型消耗1个按量读CU或者1个按量写CU。

限制

- 容量型实例下的表不支持预留读/写吞吐量。

- 当预留读/写吞吐量大于0时，即使没有读写请求也会进行计费，所以表格存储限制用户能够自行设置的单表预留读写吞吐量最大为5000 （读和写分别不超过5000）。如果用户有单表预留读写吞吐量需要超出5000的需求，可以提交工单提高预留读写吞吐量。

修改预留读/写吞吐量

应用程序可以通过UpdateTable接口动态修改表的预留读/写吞吐量配置。预留读/写吞吐量的更新有如下规则：

- 每个自然日内（UTC时间00:00:00到第二天的00:00:00，北京时间早上8点到第二天早上8点），上调或者下调预留读/写吞吐量的总次数不做限制，一张表上的两次更新的间隔必须大于2分钟。
- 预留读/写吞吐量调整完毕后1分钟内生效。

按量读/写吞吐量

按量读/写吞吐量是数据表在每一秒钟实际消耗的读/写吞吐量中超出预留读/写吞吐量的部分，统计周期为1秒。每个小时内，表格存储对预留吞吐量取平均值，对按量吞吐量取累加值来作为用户实际消耗的吞吐量。

由于按量读/写吞吐量的模式无法预估需要为数据表预留的计算资源，表格存储需要提供足够的服务能力以应对突发的访问高峰，所以按量吞吐量的单价高于预留吞吐量的单价。合理设置数据表的预留吞吐量能够有效地降低使用成本。

假如数据表设置的预留读吞吐量为100CU，连续3秒的访问情况如下：

- T0：读操作实际消耗120CU读吞吐量，则这1秒内预留吞吐量为100，消耗的按量读吞吐量为20CU。
- T1：读操作实际消耗95CU读吞吐量，则这1秒内预留吞吐量为100，消耗的按量读吞吐量为0CU。
- T2：读操作实际消耗110CU读吞吐量，则这1秒内预留吞吐量为100，消耗的按量读吞吐量为10CU。

T0至T2时刻的消耗的读吞吐量为：100CU预留读吞吐量以及30CU按量读吞吐量。

说明：

由于按量读/写吞吐量无法准确估计需要预留的资源，在某些极端访问情况下，若单个分片键每秒钟的访问需要消耗10000 CU，表格存储可能会返回 OTSCapacityUnitExhausted 错误给应用程序。此时，应用程序需要使用退避重试等策略来减少访问该表的频率。

3.4 地域

地域（Region）是指阿里云物理数据中心所在的位置。表格存储部署在多个地域中，您可以根据自身的业务需求创建不同地域中的表格存储实例。

表格存储当前支持的地域与RegionID的对应关系如下表所示：

地域名称	RegionID
华东 1	cn-hangzhou
华东 1 金融云	cn-hangzhou-finance
华东 2	cn-shanghai
华东 2 金融云	cn-shanghai-finance-1
华北 2	cn-beijing
华北 3	cn-zhangjiakou
华北 5	cn-huhehaote
华南 1	cn-shenzhen
香港	cn-hongkong
亚太东南 1（新加坡）	ap-southeast-1
美国东部 1（弗吉尼亚）	us-east-1
美国西部 1（硅谷）	us-west-1
亚太东北 1（东京）	ap-northeast-1
欧洲中部 1（法兰克福）	eu-central-1
中东东部 1（迪拜）	me-east-1
亚太东南 2（悉尼）	ap-southeast-2
亚太东南 3（吉隆坡）	ap-southeast-3
亚太东南 5（雅加达）	ap-southeast-5
亚太南部 1（孟买）	ap-south-1

4 Wide Column

4.1 模型介绍

表格存储（Table Store）是阿里云自研的NoSQL多模型数据库，Wide Column 模型是表格存储的采用的模型之一，本文主要为您介绍Wide Column 模型的构成以及与关系模型的区别。

Wide Column 模型

Wide Column 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（Primary Key）
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。在同一个分区键内，我们提供跨行事务。更多信息，请参见主键和属性文档。
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，一行可存储不限个数个属性列。
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）	Table Store 支持多种数据类型，包含 String、Binary、Double、Integer 和 Boolean。
生命周期（Time To Live）	每个表可定义数据生命周期。例如生命周期配置为一个月，则该表数据中在一个月之前写入的数据就会被自动清理。数据的写入时间由版本来决定，版本一般由服务端在数据写入时根据服务器时间来定，也可由应用指定。更多详情，请参见数据版本和生命周期文档。
最大版本数（Max Versions）	每个表可定义每一列最多保存的版本数，用于控制一列的版本的个数。 当一个属性列的版本个数超过 Max Versions 时，最早的版本将被异步删除。

Wide Column 模型对比关系模型区别

Wide Column 模型对比关系模型区别如下：

模型	描述
Wide Column 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。

模型	描述
关系模型	二维（行、列）以及固定的Schema。

4.2 主键和属性

表、行、主键和属性是表格存储的核心组件。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

主键是表中每一行的唯一标识，主键由 1 到 4 个主键列组成。创建表的时候，必须明确指定主键的组成、每一个主键列的名字、数据类型以及主键的顺序。表格存储的主键数据类型可以是String、Binary或Integer类型。

表格存储根据表的主键索引数据，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。表格存储会根据表中每一行分区键的值所属的范围自动将这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。表格存储服务会根据特定的规则对分区进行分裂和合并，这个过程是系统自动的。

说明:

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

属性

属性由多个属性列组成。每行的属性列个数没有限制，即每行的属性列可不同。一个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，属性列中的值会根据您的需求保留多个版本，供查询和使用。另外，属性列中的数据是有生命周期（TTL）的。具体参见数据版本和生命周期。

4.3 数据版本和生命周期

本文为您介绍表格存储的数据版本以及生命周期功能，数据版本以及生命周期可以帮助您有效的管理数据。

版本号

每次更新属性列的值都会为该值生成一个新版本，版本的值即为版本号（时间戳）。在写入数据时，您可以自定义属性列的版本号。如果不指定版本号，表格存储会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）作为属性列生成版本号。

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。版本号主要可以实现以下功能：

- 数据的生命周期（TTL）

  版本号可以定义数据表的生命周期。例如属性列版本号为 1468944000000 （即 2016-07-20 00:00:00 UTC），当数据表的 TTL设置为 86400 （一天）时，该版本的数据将会在 2016-07-21 00:00:00 UTC 过期，随后会被后台系统自动删除。

  当数据的版本号完全由服务端决定时，写入的数据在写入后经过设置的 TTL 后会被系统清理。
- 每行数据的版本读取

  读取一行数据时，可以指定每列最多读取多少版本或者读取的版本号范围。

最大版本数

最大版本数（Max Versions）表示该数据表中的属性列能够保留多少个版本的数据。当一个属性列的版本个数超过 Max Versions 时，最早的版本将被异步删除。

在创建数据表时，您可以自定义属性列的最大版本数。建表后，您也可以通过 UpdateTable 接口动态更改数据表的 Max Versions。

说明：

- 超过 Max Versions 的数据版本为无效数据，即使数据还没有被真正删除，该数据对用户已经不可见，无法读出。
- 当调小 Max Versions 时，如果数据版本个数超过新设的 Max Versions，最早的版本会被系统异步删除。
- 当调大 Max Versions 时，如果以前版本个数超过旧的 Max Versions 还没有被系统删除，数据会被重新读出来。

有效版本偏差

有效版本偏差（Max Version Offset）是您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。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 Max Version Offset 可以避免这种情况。

为了保证数据写入成功，表格存储在处理写请求时会对属性列的版本号进行检查。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)。属性列版本号为毫秒，其除以 1000 换算成秒之后必须属于这个范围。如果当版本不属于这个范围，该行数据写入失败。

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

指定有效版本偏差时，注意以下几点：

- 建数据表时，用户若不设置有效版本偏差，将使用默认值 86400。
- 建表后，可以通过 UpdateTable 接口动态更改有效版本偏差。
- 有效版本偏差为非 0 值，可以大于 1970-01-01 00:00:00 UTC 时间到当前时间的秒数。

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，以减少用户的数据存储空间，降低存储成本。

例如，一个数据表的 TTL 设置为 86400 （一天），在 2016-07-21 00:00:00 UTC 时，该数据表上所有版本号小于 1468944000000 （除以 1000 换算成秒之后即 2016-07-20 00:00:00 UTC）的属性列都将过期，系统会自动清理这些过期的数据。

说明：

- 超过 TTL 的过期数据为无效数据，即使数据还没有被真正删除，该数据对用户已经不可见，无法读出。
- 当调小 TTL 时，可能会有数据因为 TTL 变小而过期，这部分数据会被系统异步删除。
- 当调大 TTL 时，如果有版本号在上个 TTL 之外的数据还没有被系统删除，数据会被重新读出。
- 如果希望数据永不过期，将TTL设置为 -1。

- 建表后，可以通过 UpdateTable 接口动态更改 TTL。

4.4 命名规则和数据类型

本文主要介绍表格存储的命名规则和数据类型。

命名规则

表格存储的表名以及列名须符合以下规则：

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 字符之间。
表明是否可重复	- 同一个实例下不能有同名的表。 - 不同实例内的表名称可以相同。

主键列数据类型

主键列的数据类型只能是 String、Integer 和 Binary：

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	8 Bytes
Binary	二进制数据，可为空	长度不超过 1 KB

属性列数据类型

表格存储的属性列支持以下 5 种数据类型：

数据类型	定义	大小限制
String	UTF-8，可为空	参考限制说明
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	参考限制说明

4.5 基础数据操作

表格存储的表由行组成，每一行包含主键和属性。本节将介绍表格存储数据的操作方法。

表格存储行简介

组成表格存储表的基本单位为行，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，并且每一行的属性可以不同。更多信息请参见表格存储的数据模型概念。

表格存储的数据操作有以下三种类型：

- 单行操作
  - GetRow：读取单行数据。
  - PutRow：新插入一行。如果该行内容已经存在，则先删除旧行，再写入新行。
  - UpdateRow：更新一行。应用可以增加、删除一行中的属性列，或者更新已经存在的属性列的值。如果该行不存在，则新增一行。
  - DeleteRow：删除一行。
- 批量操作
  - BatchGetRow：批量读取多行数据。
  - BatchWriteRow：批量插入、更新或者删除多行数据。
- 范围读取
  - GetRange：读取表中一个范围内的数据。

表格存储单行操作

- 单行写入操作

  表格存储的单行写操作有三种：PutRow、UpdateRow 和 DeleteRow。下面分别介绍每种操作的行为语义和注意事项：

  - PutRow：新写入一行。如果这一行已经存在，则该行旧的数据会被删除，再新写入一行。
  - UpdateRow：更新一行。表格存储会根据请求的内容在这一行中新增列，修改或者删除指定列的值。如果这一行不存在，则会插入新的一行。但是有一种特殊的场景，若 UpdateRow 请求只包含删除指定的列，且该行不存在，则该请求不会插入新行。
  - DeleteRow：删除一行。如果删除的行不存在，则不会发生任何变化。

  应用程序通过设置请求中的 condition 字段来指定写入操作执行时，是否需要对行的存在性进行检查。condition 有三种类型：

  - 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。

  condition 为 EXPECT_NOT_EXIST 的 DeleteRow、UpdateRow 操作是没有意义的，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，如参数检查失败、单行数据量过多、行存在性检查失败等等，会返回错误码给应用程序。如果操作成功，表格存储会将操作消耗的服务能力单元返回给应用程序。

  各操作消耗的写服务能力单元的计算规则如下：

  - PutRow：本次消耗的写 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。若指定条件检查不为 IGNORE，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。如果操作不满足应用程序指定的行存在性检查条件，则操作失败并消耗 1 个写 CU 和 1 个 读 CU。更多详情请参见 PutRow 详解。
  - UpdateRow：本次消耗的写 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。UpdateRow 中包含的需要删除的属性列，只有其列名计入该属性列数据大小。若指定条件检查不为 IGNORE，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。如果操作不满足应用程序指定的行存在性检查条件，则操作失败并消耗 1 个写 CU 和 1 个读 CU。更多详情请参见 UpdateRow 详解。
  - DeleteRow：被删除的行主键数据大小除以 4 KB 向上取整。若指定条件检查不为 IGNORE，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。如果操作不满足应用程

序指定的行存在性检查条件，则操作失败并消耗 1 个写 CU。更多详情请参见 DeleteRow 详解。

写操作会根据指定的 condition 情况消耗一定的读 CU。

示例：

下面将举例说明单行写操作的写 CU 和读 CU 的计算。

示例 1，使用 PutRow 进行行写入操作：

```
// PutRow 操作
// row_size=len('pk')+len('value1')+len('value2')+8Byte+1300Byte+
3000Byte=4322Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(1300Byte), 'value2':String(3000Byte
)}
}

// 原来的行
// row_size=len('pk')+len('value2')+8Byte+900Byte=916Byte
// row_primarykey_size=len('pk')+8Byte=10Byte
{
    primary_keys:{'pk':1},
    attributes:{'value2':String(900Byte)}
}
```

读/写服务能力单元（CU）的消耗情况如下：

- 将 condition 设置为 EXPECT_EXIST 时：消耗的写 CU 为 4322 Byte 除以 4 KB 向上取整，消耗的读 CU 为该行主键数据大小 10 Byte 除以 4 KB 向上取整。该 PutRow 操作消耗 2 个写 CU 和 1 个读 CU。
- 将 condition 设置为 IGNORE 时：消耗的写 CU为 4322 Byte 除以 4 KB 向上取整，消耗 0 个读 CU。该 PutRow 操作消耗 2 个写 CU 和 0 个读 CU。
- 将 condition 设置为 EXPECT_NOT_EXIST 时：指定的行存在性检查条件检查失败，该 PutRow 操作消耗 1 个写 CU 和 1 个读 CU。

示例 2，使用 UpdateRow 新写入一行：

```
// UpdateRow 操作
// 删除的属性列列名长度计入 row_size
// row_size=len('pk')+len('value1')+len('value2')+8Byte+900Byte=
922Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(900Byte), 'value2':Delete}
}

// 原来的行不存在
// row_size=0
```

CU 的消耗情况如下：

- 将 condition 设置为 IGNORE 时：消耗的写 CU 为 922 Byte 除以 4 KB 向上取整，消耗 0 个读 CU。该 UpdateRow 操作消耗 1个写 CU 和 0 个读 CU。
- 将 condition 设置为 EXPECT_EXIST 时：指定的行存在性检查条件检查失败，该 PutRow 操作消耗1个写 CU 和1个读 CU。

示例 3，使用 UpdateRow 对存在的行进行更新操作：

```
// UpdateRow 操作
// row_size=len('pk')+len('value1')+len('value2')+8Byte+1300Byte+
3000Byte=4322Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(1300Byte), 'value2':String(3000Byte
)}
}
// 原来的行
// row_size=len('pk')+len('value1')+8Byte+900Byte=916Byte
// row_primarykey_size=len('pk')+8Byte=10Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(900Byte)}
}
```

CU的消耗情况如下：

- 将 condition 设置为 EXPECT_EXIST 时：消耗的写 CU 为 4322 Byte 除以 4 KB 向上取整，消耗的读 CU 为该行主键数据大小 10 Byte 除以 4 KB 向上取整。该 UpdateRow 操作消耗 2 个写 CU 和 1 个读 CU。
- 将 condition 设置为 IGNORE 时：消耗的写 CU 为 4322 Byte 除以 4 KB 向上取整，消耗 0 个读 CU，该 UpdateRow 操作消耗 2 个写 CU 和 0 个读 CU。

示例 4，使用 DeleteRow 删除不存在的行：

```
//原来的行不存在
//row_size=0

//DeleteRow 操作
//row_size=0
//row_primarykey_size=len('pk')+8Byte=10Byte
{
    primary_keys:{'pk':1},
}
```

修改前后的数据大小均为 0，无论读写操作成功还是失败至少消耗 1CU。因此，该 DeleteRow 操作消耗 1 个写 CU。

CU 的消耗情况如下：

- 将 condition 设置为 EXPECT_EXIST 时：消耗的写 CU 为该行主键数据大小 10 Byte 除以 4 KB 向上取整，消耗的读 CU 为该主键数据大小 10 Byte 除以 4 KB 向上取整。该 DeleteRow 操作消耗 1 个写 CU 和 1 个读 CU。
- 将 condition 设置为 IGNORE 时：消耗的写 CU 为该行主键数据大小 10 Byte 除以 4 KB 向上取整，消耗 0 个读 CU。该 DeleteRow 操作消耗 1 个写 CU 和 0 个读 CU。

更多信息请参见 API Reference 中的 PutRow、UpdateRow 和 DeleteRow 章节。

- 单行读取操作

表格存储的单行读操作只有一种：GetRow。

应用程序提供完整的主键和需要返回的列名。列名可以是主键列或属性列，也可以不指定要返回的列名，此时请求返回整行数据。

表格存储根据被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 向上取整作为本次读取操作消耗的读 CU。如果操作指定的行不存在，则消耗 1 个读CU，单行读取操作不会消耗写 CU。

使用 GetRow 读取一行消耗的写 CU 的计算，示例如下：

```
//被读取的行

//row_size=len('pk')+len('value1')+len('value2')+8Byte+1200Byte+
3100Byte=4322Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(1200Byte), 'value2':String(3100Byte
)}
}

//GetRow 操作
//获取的数据 size=len('pk')+len('value1')+8Byte+1200Byte=1216Byte
{
    primary_keys:{'pk':1},
    columns_to_get:{'value1'}
}
```

消耗的读 CU 为 1216 Byte 除以 4 KB 向上取整，该 GetRow 操作消耗 1 个读 CU。

更多信息请参见 API Reference 的 GetRow 章节。

多行操作

表格存储提供了 BatchWriteRow 和 BatchGetRow 两种多行操作。

- BatchWriteRow 用于插入、修改、删除一个表或者多个表中的多行记录。BatchWriteRow 操作由多个 PutRow、UpdateRow、DeleteRow 子操作组成。

  BatchWriteRow 的各个子操作独立执行，表格存储会将各个子操作的执行结果分别返回给应用程序。返回结果可能存在部分请求成功、部分请求失败的现象。即使整个请求没有返回错误，应用程序也会检查每个子操作返回的结果，从而拿到正确的状态。BatchWriteRow 的各个子操作单独计算写服务能力单元。
- BatchGetRow 用于读取一个表或者多个表中的多行记录。

  BatchGetRow 各个子操作独立执行，表格存储会将各个子操作的执行结果分别返回给应用程序。返回结果可能存在部分请求成功、部分请求失败的现象。即使整个请求没有返回错误，应用程序也会检查每个子操作返回的结果，从而拿到正确的状态。

  BatchGetRow 的各个子操作单独计算读服务能力单元。

更多信息请参见 API Reference 中的 BatchWriteRow 与 BatchGetRow 章节。

范围读取操作

表格存储提供了范围读取操作 GetRange，该操作将指定主键范围内的数据返回给应用程序。

表格存储表中的行按主键进行从小到大排序，GetRange 的读取范围是一个左闭右开的区间。操作会返回主键属于该区间的行数据，区间的起始点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。

GetRange 操作需要指定请求列名，请求列名中可以包含多个列名。如果某一行的主键属于读取的范围，但是不包含指定返回的列，那么请求返回结果中不包含该行数据。不指定请求列名，则返回完整的行。

GetRange 操作需要指定读取方向，读取方向可以为正序或逆序。假设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。

GetRange 操作可以指定最大返回行数。表格存储按照正序或者逆序最多返回指定的行数之后即结束该操作的执行，即使该区间内仍有未返回的数据。

GetRange 操作可能在以下几种情况下停止执行并返回数据给应用程序：

- 返回的行数据大小之和达到 4 MB。
- 返回的行数等于 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被全部使用，余量不足以读取下一条数据。同时 GetRange 请求的返回结果中还包含下一条未读数据的主键，应用程序可以使用该返回值作为下一次 GetRange 操作的起始点继续读取。如果下一条未读数据的主键为空，表示读取区间内的数据全部返回。

表格存储的读取计算为，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读取的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，若读取范围中包含 10 行，每行主键数据大小与实际读取到的属性列数据之和为 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）。

示例

下面举例说明 GetRange 操作的行为。假设表的内容如下，PK1、PK2 是表的主键列，类型分别为 String 和 Integer；Attr1、Attr2 是表的属性列。

PK1	PK2	Attr1	Attr2
'A'	2	'Hell'	'Bell'
'A'	5	'Hello'	不存在
'A'	6	不存在	'Blood'
'B'	10	'Apple'	不存在
'C'	1	不存在	不存在
'C'	9	'Alpha'	不存在

示例 1，读取某一范围内的数据：

```
// 请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 2)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INTEGER, 1)

// 响应
cosumed_read_capacity_unit: 1
rows: {
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
        attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING,
 "Bell")
      },
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
        attribute_columns:("Attr1", STRING, "Hello")
      },
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
        attribute_columns:("Attr2", STRING, "Blood")
      },
      {
```

```
        primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
        attribute_columns:("Attr1", STRING, "Apple")
      }
    }
```

示例 2，利用 INF_MIN 和 INF_MAX 读取全表数据：

```
// 请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", INF_MIN)
exclusive_end_primary_key: ("PK1", INF_MAX)

// 响应
cosumed_read_capacity_unit: 1
rows: {
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
        attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING,
 "Bell")
      },
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
        attribute_columns:("Attr1", STRING, "Hello")
      },
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
        attribute_columns:("Attr2", STRING, "Blood")
      },
      {
        primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
        attribute_columns:("Attr1", STRING, "Apple")
      }
      {
        primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 1)
      }
      {
        primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 9)
        attribute_columns:("Attr1", STRING, "Alpha")
      }
    }
```

示例 3，在某些主键列上使用 INF_MIN 和 INF_MAX：

```
// 请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)

// 响应
cosumed_read_capacity_unit: 1
rows: {
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
        attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING,
 "Bell")
      },
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
```

```
        attribute_columns:("Attr1", STRING, "Hello")
      },
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
        attribute_columns:("Attr2", STRING, "Blood")
      }
    }
```

示例 4，逆序读取：

```
// 请求
table_name: "table_name"
direction: BACKWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INTEGER, 1)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 5)

// 响应
cosumed_read_capacity_unit: 1
rows: {
      {
        primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 1)
      },
      {
        primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
        attribute_columns:("Attr1", STRING, "Apple")
      },
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
        attribute_columns:("Attr2", STRING, "Blood")
      }
    }
```

示例 5，指定列名不包含 PK：

```
// 请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MAX)
columns_to_get: "Attr1"

// 响应
cosumed_read_capacity_unit: 1
rows: {
      {
        attribute_columns: {"Attr1", STRING, "Alpha"}
      }
    }
```

示例 6，指定列名中包含 PK：

```
// 请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MAX)
columns_to_get: "Attr1", "PK1"
```

```
// 响应
cosumed_read_capacity_unit: 1
rows: {
      {
        primary_key_columns:("PK1", STRING, "C")
      }
      {
        primary_key_columns:("PK1", STRING, "C")
        attribute_columns:("Attr1", STRING, "Alpha")
      }
    }
```

示例 7，使用 limit 和断点：

```
// 请求 1
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)
limit: 2

// 响应 1
cosumed_read_capacity_unit: 1
rows: {
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
        attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING,
 "Bell")
      },
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
        attribute_columns:("Attr1", STRING, "Hello")
      }
    }
next_start_primary_key:("PK1", STRING, "A"), ("PK2", INTEGER, 6)

// 请求 2
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 6)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)
limit: 2

// 响应 2
cosumed_read_capacity_unit: 1
rows: {
      {
        primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
        attribute_columns:("Attr2", STRING, "Blood")
      }
    }
```

示例 8，使用 GetRange 操作消耗的读 CU 计算：

在以下表中执行 GetRange 操作，其中 PK 1 是表的主键列，Attr1、Attr2 是表的属性列。

PK1	Attr1	Attr2
1	不存在	String（1000Byte）
2	8	String（1000Byte）
3	String（1000Byte）	不存在
4	String（1000Byte）	String（1000Byte）

```
// 请求
table_name: "table2_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", INTEGER, 1)
exclusive_end_primary_key: ("PK1", INTEGER, 4)
columns_to_get: "PK1", "Attr1"

// 响应
cosumed_read_capacity_unit: 1
rows: {
      {
        primary_key_columns:("PK1", INTEGER, 1)
      },
      {
        primary_key_columns:("PK1", INTEGER, 2),
        attribute_columns:("Attr1", INTEGER, 8)
      },
      {
        primary_key_columns:("PK1", INTEGER, 3),
        attribute_columns:("Attr1", STRING, String (1000Byte) )
      },
    }
```

此次 GetRange 请求中：

- 获取的第一行数据大小为：len ('PK1') + 8 Byte = 11 Byte
- 第二行数据大小为：len ('PK1') + 8 Byte + len ('Attr1') + 8 Byte = 24 Byte
- 第三行数据大小为：len ('PK1')+ 8 Byte + len ('Attr1') + 1000 Byte = 1016 Byte

消耗的读服务能力单元为获取的三行数据之和 11 Byte + 24 Byte + 1016 Byte = 1051 Byte 除以 4 KB 向上取整，该 GetRange 操作消耗 1 个读服务能力单元。

更多详细信息请参见 API Reference 中的 GetRange 章节。

最佳实践

表格存储数据操作的最佳实践

使用表格存储 SDK 进行数据操作

使用 TableStore Java SDK 进行数据操作

使用 TableStore Python SDK 进行数据操作

4.6 主键列自增

若设置某一列主键为自增列，在写入一行数据时，这一列主键无需填值，表格存储会自动生成这一主键列的值。该值在分区键上保证唯一，且严格递增。

特点

表格存储的主键列自增主要有以下特点：

- 生成的自增列的值唯一，且严格递增。
- 自动生成的自增列为 64 位的有符号长整型。
- 分区键级别严格递增。
- 自增列功能是表级别的，同一个实例下面可以有自增列的表，也可以有非自增列的表。

使用主键列自增功能后，条件更新的逻辑和之前一样，具体如下表所示：

API	IGNORE	EXPECT_EXIST	EXPECT_NOT_EXIST
PutRow：已存在行	失败	成功	失败
PutRow：不存在行	成功	失败	失败
UpdateRow：已存在行	失败	成功	失败
UpdateRow：不存在行	成功	失败	失败
DeleteRow：已存在行	成功	成功	失败
DeleteRow：不存在行	失败	失败	失败

限制

主键列自增有以下限制：

- 表格存储支持多个主键，第一个主键为分区键，分区键不允许设置为自增列，其它任一主键都可以设置为自增列。
- 每张表最多只允许设置一个主键为自增列。
- 属性列不能设置为自增列。
- 仅支持在创建表的时候指定自增列，对于已存在的表不支持创建自增列。

接口

以下为自增列的相关接口说明：

接口	说明
CreateTable	· 建表的时候需要设置某一列为自增列。 · 表创建好后，无法再次更改表为自增表。
UpdateTable	无法通过UpdateTable更改表的自增属性。
PutRow/UpdateRow/BatchWriteRow	· 写入的时候，自增的列不需要设置具体值，只需要设置一个占位符，例如AUTO_INCREMENT。 · 可以设置ReturnContent中的ReturnType为RT_PK，即返回完整主键值。返回的完整主键值可以用于GetRow查询。
GetRow/BatchGetRow	GetRow的时候需要完整主键列，可以通过设置PutRow、UpdateRow或BatchWriteRow中的ReturnType为RT_PK获取到。

场景

Table Store主键列自增功能在IM系统中的应用

使用

JAVA SDK：主键列自增

计费

主键列自增功能不影响现有计费逻辑，返回的主键列数据不会额外消耗读CU。

4.7 条件更新

条件更新功能只有在满足条件时才对表中的数据进行更改，当不满足条件时更新失败。

该功能支持算术运算（=、!=、>、>=、<、<=）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
- BatchWriteRow

列判断条件包括行存在性条件和列条件，其中：

- 行存在性条件分为：
  - `IGNORE`：忽略
  - `EXPECT_EXIS`：期望存在
  - `EXPECT_NOT_EXIST`：期望不存在

  对表进行更改操作时，会首先检查行存在性条件，若不满足，则更改失败，对用户抛错。
- 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。

条件更新可以实现乐观锁的功能，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`，更新该行同时使列 `A=2`。若更新失败，代表有其他客户端已经成功更新了该行。

操作步骤

1. 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 这一列，条件检查不通过。
 singleColumnValueCondition.setPassIfMissing(false);
 // 只判断最新版本。
 singleColumnValueCondition.setLatestVersionsOnly(true);
```

2. 构造 CompositeColumnValueCondition。

```
 // composite1 条件为 (Col0 == 0) AND (Col1 > 100)
 CompositeColumnValueCondition composite1 = new CompositeColumnValue
Condition(CompositeColumnValueCondition.LogicOperator.AND);
 SingleColumnValueCondition single1 = new SingleColumnValueCondition
("Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL,
ColumnValue.fromLong(0));
 SingleColumnValueCondition single2 = new SingleColumnValueCondition
("Col1",
         SingleColumnValueCondition.CompareOperator.GREATER_THAN,
ColumnValue.fromLong(100));
 composite1.addCondition(single1);
 composite1.addCondition(single2);

 // composite2 条件为 ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10
)
 CompositeColumnValueCondition composite2 = new CompositeColumnValue
Condition(CompositeColumnValueCondition.LogicOperator.OR);
 SingleColumnValueCondition single3 = new SingleColumnValueCondition
("Col2",
```

```
          SingleColumnValueCondition.CompareOperator.LESS_EQUAL,
ColumnValue.fromLong(10));
 composite2.addCondition(composite1);
 composite2.addCondition(single3);
```

3. 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(pkValue));
     PrimaryKey primaryKey = primaryKeyBuilder.build();

     // 读一行
     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 这一列, 使列值 +1
     RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);
     Condition condition = new Condition(RowExistenceExpectation.
EXPECT_EXIST);
     ColumnCondition columnCondition = new SingleColumnValueCon
dition("Col0", SingleColumnValueCondition.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());
     }
 }
```

使用场景示例

对高并发的应用进行更新:

```
  // 取回当前值
  old_value = Read();
  // 对当前值进行计算, 比如加 1 操作
  new_value = func(old_value);
  // 使用最新值进行更新
  Update(new_value);
```

在高并发的环境下，old_value 可能被其他客户端更新，若使用 Conditional Update，就可以做到 Update (new_value) if value 等于 old_value。

说明:

在网页计数和游戏等高并发场景下，使用条件更新后会有一定几率的失败，需要一定次数的重试。

费用计算

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。

4.8 局部事务

表格存储为您提供了局部事务功能，即您可以创建一个范围不超过一个分区键值的事务，并在该事务内进行读写操作。最终提交事务从而使这些改动永久生效，或者丢弃事务从而放弃这些改动。

在使用局部事务时，您可以先在指定的分区键值上创建一个局部事务，此时表格存储服务端会返回一个事务ID。您可以使用此事务ID在对应的分区键值范围内进行读写操作，然后使用事务ID选择提交该事务，使事务中的所有数据修改生效。或者使用事务ID放弃该事务，则该事务中的所有修改都不会应用到原有数据上。

使用局部事务功能可以实现单行或多行读写的原子操作，从而扩展了使用场景。

说明:

- 分区键的概念请参阅主键和属性。
- 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，请通过工单进行申请，或加入钉钉群“表格存储公开交流群”进行咨询。

相关接口

- `StartLocalTransaction`：创建一个局部事务。
- `CommitTransaction`：提交一个事务。
- `AbortTransaction`：丢弃一个事务。
- `PutRow`、`UpdateRow`、`DeleteRow`、`BatchWriteRow`等写接口均支持局部事务。
- `GetRow`、`GetRange`等读接口均支持局部事务。

典型使用场景

- 读-写场景（简单场景）

  当您要进行读取-修改-写回（Read-Modify-Write）操作时，您可以选择：

  - 使用条件更新。
  - 使用原子计数器。

  但这两种方法都存在某些限制：

  - 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，或需要多次写入的情况。
  - 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。

  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：

  - 首先使用StartLocalTransaction针对这个分区键值创建一个事务。
  - 使用GetRow或GetRange读取数据。请求中需要带上事务ID。
  - 客户端本地修改数据。
  - 使用PutRow、UpdateRow、DeleteRow、或BatchWriteRow将修改后的数据写回，且请求中需要带上事务ID。
  - 使用CommitTransaction提交该事务。
- 邮箱场景（复杂场景）

  我们可以使用局部事务来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，我们在一张物理表上同时使用了一张主表和两张索引表，其主键列为：

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

表	UserID	Type	IndexField	MailID
SendTime表	用户ID	"SendTime"	$SendTime	邮件ID

其中，我们用Type列来区分主表和不同的索引表，不同的索引行用IndexField列保存不同含义的字段，而主表本身不用这一列。

我们可以使用局部事务来完成以下操作：

- 列出某个用户发送的最近100封邮件：
  - 使用UserID创建一个局部事务，获取事务ID。
  - 使用事务ID对SendTime表调用GetRange，获取100封邮件。
  - 使用事务ID对主表调用BatchGetRow，获取100封邮件的详细信息。
  - 提交或丢弃事务（因为该事务没有任何写操作，两个操作是等同的）。
- 将某个目录下的所有邮件移到另一个目录下：
  - 使用UserID创建一个局部事务，获取事务ID。
  - 使用事务ID对Folder表调用GetRange，获取若干封邮件。
  - 使用事务ID对Folder表调用BatchWriteRow。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。
  - 提交事务。
- 统计某个目录下已读邮件与未读邮件的数量（非最优方案，见下文解释）：
  - 使用UserID创建一个局部事务，获取事务ID。
  - 使用事务ID对Folder表调用GetRange，获取若干封邮件。
  - 使用事务ID对主表调用BatchGetRow，获取每封邮件的已读状态。
  - 提交或丢弃事务。

在这个场景中，我们还可以再增加新的索引表以加速一些常用操作。有了局部事务后，我们不用再担心主表与索引表的状态不一致，极大地降低了开发的难度。比如“统计邮件数量”这个功能在上面的方案中需要读取很多封邮件，开销较大，我们可以用一个新的索引表来保存已读和未读邮件的数量，从而降低开销，加速查询。

注意事项

- 在事务期间，对应分区键值相当于被锁上，其它不持有事务ID在这个范围内的写请求都会失败。在事务提交或放弃或事务超时后，对应的锁也会被释放。
- 同一个事务中所有写请求的分区键值必须与创建事务时的分区键值相同，读请求则无此限制。

- 一个局部事务只能同时被一个请求使用，在事务被使用期间，其它使用此事务ID的操作都会失败。
- 若用户在事务中写入了未填版本的Cell，则在事务内进行读取时同样会读到未填版本的对应Cell，且该Cell的版本被认为高于任何已填版本。
- 若用户在事务中写入了未填版本的Cell，该Cell的版本会在提交时由表格存储的服务端确定，规则与正常的写入一个未填版本的Cell相同。
- 不可使用事务ID访问事务范围（即创建时使用的分区键值）以外的数据。
- 在创建事务后，若长时间未使用该事务，则表格存储服务端会认为此事务超时，并将事务丢弃。
- 未提交的事务可能失效，若出现此情况，用户需要重试该事务内的操作。
- 若创建事务时超时，此请求可能在表格存储的服务端已执行成功，此时用户需要等待该事务超时后重新创建。
- 若`BatchWriteRow`请求中带有事务ID，则此请求中所有行只能操作该事务ID对应的表。

限制

- 每个事务中写入的数据量最大为4MB，按正常的写请求数据量计算规则累加。
- 每个事务中两次读写操作最大间隔为60秒，超过60秒未操作的事务被视为超时。
- 每个事务从创建开始生命期最长为60秒，超过60秒未提交或丢弃的事务被视为超时。

写入的数据量计算方式请参阅计量项和计费说明。

错误码

- 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。

计量计费

- `StartLocalTransaction`、`CommitTransaction`、`AbortTransaction`请求分别消耗1个单位的写能力单元。
- 其它读写请求的计量计费与正常的读写请求相同。

关于计量计费详情，请参阅计量项和计费说明。

示例代码

- 创建局部事务

  我们可以调用AsyncClient或SyncClient的startLocalTransaction方法来创建一个局部事务，参数为一个分区键的值，并获得对应的事务ID：

```
PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrim
aryKeyBuilder();
primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.
fromString("txnKey"));
PrimaryKey primaryKey = primaryKeyBuilder.build();
StartLocalTransactionRequest request = new StartLocalTransactio
nRequest(tableName, primaryKey);
String txnId = client.startLocalTransaction(request).getTransac
tionID();
```

- 在事务范围内进行读写

  在事务范围内进行读写的方式与正常读写几乎相同，只要填入事务ID即可。

  写入一行数据：

```
PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrim
aryKeyBuilder();
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);
```

  读取这行数据：

```
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

GetRowRequest  request = new GetRowRequest(criteria);
request.setTransactionId(txnId);
```

```
GetRowResponse getRowResponse = client.getRow(request);
```

- 提交事务

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

- 丢弃事务

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

4.9 原子计数器

原子计数器是将列当成一个原子计数器来使用，便于为某些在线应用提供实时统计功能，比如统计帖子的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后端服务器，后端服务器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后端服务器，在保证强一致性的情况下，提升原子计数器的写入性能。

接口说明

RowUpdateChange类新增原子计数器接口：

- RowUpdateChange increment(Column column)：对列执行增量变更（如：+X，-X等）。
- void addReturnColumn(String columnName)：对于涉及原子加的列，设置需要返回列值的列名。
- void setReturnType(ReturnType.RT_AFTER_MODIFY)：设置返回类型，返回指定的原子加列的新值。

写入数据时，使用RowUpdateChange接口对整型列做列值的增量变更，如下所示：

```
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();

        RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);

        // 将price列值+10，不允许设置时间戳
        rowUpdateChange.increment(new Column("price", ColumnValue.
fromLong(10)));
```

```
        // 设置ReturnType将原子计数器的结果返回
        rowUpdateChange.addReturnColumn("price");
        rowUpdateChange.setReturnType(ReturnType.RT_AFTER_MODIFY);

        // 对price列发起原子计数器操作
        UpdateRowResponse response = client.updateRow(new UpdateRowR
equest(rowUpdateChange));

        // 打印出更新后的新值
        Row row = response.getRow();
        System.out.println(row);
    }
```

说明:

- RowUpdateChange.addReturnColumn(列名)：设置需要返回的列。
- RowUpdateChange.setReturnType(RT_AFTER_MODIFY：指定本次修改需要返回上述设定的列值。

使用场景

您可以使用原子计数器对某一行中的数据做实时统计。假设某一用户需要使用表格存储图片元信息并统计图片数信息，表格内每一行对应某用户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。

- UpdateRow：增加一张新图片，原子计数器+1。
- UpdateRow：删除一张旧图片，原子计数器-1。
- GetRow：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，即当您增加一张新图片时，原子计数器会相应+1，而不会出现-1的情况。

限制

原子计数器主要存在以下几方面的限制：

- 仅支持Integer类型。
- 作为原子计数器的列，若写入前该列不存在，则默认值为0。若写入前该列已存在且列值为非Integer类型，则抛出OTSParameterInvalid错误。
- 增量值可以是正数或负数，但不允许计算溢出。若出现计算溢出，则抛出OTSParameterInvalid错误。
- 默认不返回原子加列的结果。可以通过addReturnColumn() + setReturnType()接口，指定返回哪些原子加列的结果。
- 在单次更新请求时，不能对某一列同时进行更新和原子计数的操作。假设列A已经执行原子计数器操作，则列A不能再执行其他操作（如列的覆盖写，列删除等）。

- 在一次 BatchWriteRow 请求中，可以支持对同一行的多次更新操作。但若某一行已执行原子计数器操作，则该行在此批量请求中只能出现一次。
- 列上的原子计数器只作用在最新版本上，不支持对列的特定版本做原子计数器。更新完成后，原子计数器会插入一个新的版本。
- 原子计数器操作可能会由于网络超时、系统错误等导致失败。此时，该应用程序只需重试该操作即可。这会产生更新两次计数器的风险，导致计数有可能偏多或偏少。针对此类异常场景，建议使用条件更新精确变更列值。

4.10 过滤器

过滤器（Filter）可以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。由于只返回了符合条件的数据行，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。

表格存储过滤器的过滤条件支持算术运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），支持最多 10 个条件的组合，可以用于 GetRow、BatchGetRow 和 GetRange 接口中。

使用说明

目前表格存储仅支持以下两个过滤器：

- SingleColumnValueFilter：只判断某个参考列的列值。
- CompositeColumnValueFilter：会对多个参考列的列值判断结果进行逻辑组合，决定最终是否过滤。

这两个过滤器都是基于参考列的列值决定是否过滤某行。API 文档参考：CompositeColumnValueFilter 和 SingleColumnValueFilter。

构造过滤器

- 构造 SingleColumnValueFilter。

```
  // 设置过滤器，当 Col0 的值为 0 时返回该行。
  SingleColumnValueFilter singleColumnValueFilter = new SingleColu
mnValueFilter("Col0",
          SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue
.fromLong(0));
  // 如果不存在 Col0 这一列，也不返回。
  singleColumnValueFilter.setPassIfMissing(false);
```

- 构造 CompositeColumnValueFilter。

```
  // composite1 条件为 (Col0 == 0) AND (Col1 > 100)
  CompositeColumnValueFilter composite1 = new CompositeColumnValue
Filter(CompositeColumnValueFilter.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 条件为 ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <=
10)
  CompositeColumnValueFilter composite2 = new CompositeColumnValue
Filter(CompositeColumnValueFilter.LogicOperator.OR);
  SingleColumnValueFilter single3 = new SingleColumnValueFilter("
Col2",
          SingleColumnValueFilter.CompareOperator.LESS_EQUAL,
ColumnValue.fromLong(10));
  composite2.addFilter(composite1);
  composite2.addFilter(single3);
```

过滤器是对读取后的结果再进行一次过滤，所以 SingleColumnValueFilter 或 CompositeColumnValueFilter 中的参考列必须在读取的结果内。如果您指定了要读取的列，但其中不包含参考列，则过滤器无法获得这些参考列的值。

当某个参考列不存在时，SingleColumnValueFilter 的 passIfMissing 参数决定此时是否满足条件，即您可以选择当参考列不存在时的行为。

使用场景示例

以物联网中的智能电表为例，智能电表按一定的频率（比如每隔 15 秒）将当前的电压、电流、度数等信息写入表格存储。在按天做分析的时候，我们需要拿到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，来决定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 条之多，然后再对这 5760 条信息进行过滤，拿到了最终的 10 个电压出现不稳定时的监控信息。

使用过滤器 Filter 只返回了实际需要的 10 条数据，大大降低了返回的数据量。而且，不需要再对结果进行初步的过滤处理，节省了开发代价。

费用计算

使用过滤器之后，虽然有效降低了返回的数据量，但由于服务器端进行过滤计算是在数据返回给用户之前进行的，并没有降低磁盘 IO 次数，所以消耗的读 CU 与不使用过滤器的情况下相同，即使用过滤器与否不影响 CU 计算。

例如，使用 GetRange 读取到 100 条记录，共 200KB 数据，共消耗了 50 个读 CU，在使用了过滤器之后，实际返回了 10 条数据，共 20KB，但仍然会消耗 50 个读 CU。

特别说明

在 GetRow、BatchGetRow 和 GetRange 接口中使用过滤器除了不影响费用计算外，也不会改变使用接口的原生语义及限制项设定。

使用 GetRange 接口时，会受到一次返回数据的行数超过 5000 行或者返回数据的数据大小大于 4MB 的限制，当在该次返回的 5000 行或者 4MB 数据中没有满足 Filter 过滤条件的记录时，得到的 Response 中的 Rows 为空，但是 next_start_primary_key 可能不为空，此时需要使用 next_start_primary_key 继续读取数据，直到 next_start_primary_key 为空。详情请参见 GetRange 接口说明。

5 Timeline模型

5.1 模型介绍

Timeline 模型是针对消息数据场景所设计的数据模型，它能满足消息数据场景对消息保序、海量消息存储、实时同步的特殊需求，同时支持全文检索与多维度组合查询。可以同时应用在IM、Feed流等消息场景的实现上。

模型结构

Timeline模型以简单为设计目标，核心模块构成比较清晰明了。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。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

Timeline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。

Timeline

```
<dependency>
    <groupId>com.aliyun.openservices.tablestore</groupId>
```

```
    <artifactId>Timeline</artifactId>
    <version>2.0.0</version>
</dependency>
```

TableStore Java SDK(模型已合入SDK)

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

5.2 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。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秘钥初始化对接实例，详情参见初始化。
4. 通过示例代码快速使用Timeline模型。

如果您希望了解更多关于Timeline模型的操作，参见Timeline SDK。

5.3 基础操作

5.3.1 概述

Timeline模型主要适用于对消息数据场景，可以满足消息数据场景对消息保序、海量消息存储、实时同步的特殊需求，同时支持全文检索与多维度组合查询。可以同时应用在IM、Feed流等消息场景的实现上。Timeline模型Java SDK包含以下操作。

- 初始化
- Meta管理
- Timeline管理
- Queue管理

5.3.2 初始化

初始化Factory

用户将SyncClient作为参数，初始化StoreFactory，通过工厂创建Meta数据、Timeline数据的管理Store。错误重试的实现依赖SyncClient的重试策略，用户通过设置SyncClient实现重试。如有特殊需求，可自定义策略（只需实现RetryStrategy接口）。

```
/**
 * 重试策略设置
 * 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 factory = 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，表名、主键字段、索引名、索引类型等参数。

```
TimelineIdentifierSchema idSchema = new TimelineIdentifierSchema.
Builder()
        .addStringField("timeline_id").build();

IndexSchema metaIndex = new IndexSchema();
metaIndex.addFieldSchema( //配置索引字段、类型
        new FieldSchema("group_name", FieldType.TEXT).setIndex(true).
setAnalyzer(FieldSchema.Analyzer.MaxWord)
        new FieldSchema("create_time", FieldType.Long).setIndex(true)
);

TimelineMetaSchema metaSchema = new TimelineMetaSchema("groupMeta",
idSchema)
        .withIndex("metaIndex", metaIndex); //设置索引

TimelineMetaStore timelineMetaStore = serviceFactory.createMetaStore(
metaSchema);
```

建表

根据metaSchema的参数创建表，如果设置索引配置，建表成功后创建索引。

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

删表

如果表存在索引，在删表前会先删除索引，然后删除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，表名、主键字段、索引名、索引类型等参数。

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、类型
        new FieldSchema("text", FieldType.TEXT).setIndex(true).
setAnalyzer(FieldSchema.Analyzer.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，如果设置索引配置，建表成功后创建索引。

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

删表

如果表存在索引，在删表前会先删除索引，然后再删除Store相应的表。

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

5.3.3 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，具体请参考概述。

Insert

TimelineIdentifer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);
```

5.3.4 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，具体请参考概述。

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();
```

5.3.5 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、手动设置，相关配置在TimelineSchema中。

```
timelineQueue.store(message);//自增列
timelineQueue.store(sequenceId, message);//手动
```

StoreAsync

异步存储消息，用户可自定义回调，对成功、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、有回调两种。用户可自定义回调，对成功、失败做自定义处理。

```
timelineQueue.batchStore(message);//自增列
timelineQueue.batchStore(sequenceId, message);//手动

timelineQueue.batchStore(message, callback);//自增列
timelineQueue.batchStore(sequenceId, message, callback);//手动
```

Get

通过SequenceId读取单行消息，消息不存在时不抛错，返回null。

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

GetLatestTimelineEntry

读取最新一条消息，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，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，用户可自定义回调，对成功、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);
```

6 Timestream模型

6.1 模型介绍

Timestream模型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，其中，

- 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

元素	描述
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

下面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。

以上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则是快递本身，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。下面分析一下数据存储方式：

- 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在在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）放在Fields中，可以查询某个快递在某个时间范围内的轨迹。

6.2 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秘钥初始化对接实例，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。

6.3 基础操作

6.3.1 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

6.3.2 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 ConsumedCa
pacity 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, instance, 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);
```

6.3.3 表操作

Timestream数据存储包含两类表：元数据表和数据表。其中，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。

示例代码

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.KEYWORD).setIsArray(true));
```

```
        indexSchemas.add(new AttributeIndexSchema("number",
AttributeIndexSchema.Type.LONG));
        indexSchemas.add(new AttributeIndexSchema("score",
AttributeIndexSchema.Type.DOUBLE));
        indexSchemas.add(new AttributeIndexSchema("succ", AttributeI
ndexSchema.Type.BOOLEAN));
        indexSchemas.add(new AttributeIndexSchema("loc", AttributeI
ndexSchema.Type.GEO_POINT));
        db.createMetaTable(indexSchemas);
```

3. 删除元数据表。

```
        db.deleteMetaTable();
```

数据表

示例代码

1. 创建数据表。

```
        db.createDataTable("datatable");
```

2. 删除数据表。

```
        db.deleteDataTable("datatable");
```

6.3.4 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TableStore Writer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.MILLISECONDS)
                .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();
        // 同步写入时间线元数据
        metaTable.delete(identifier);
```

6.3.5 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 Timestream
Identifier.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 Timestream
Identifier.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 Timestream
Identifier.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 Timestream
Identifier.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();
```

6.3.6 限制项

限制项	最大值	说明
Name长度	100Byte	无
Tag个数	12	无
Tag名字和值的总长度	500Byte	所有Tag的名字和值的大小总和不超过500Byte

限制项	允许值	说明
Tag名	字符中不能包含等号（=）	无
Attribute名	"h"、"n"、"s"、"t"为保留字段	无

7 Grid模型

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

挑战

- 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。

- 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。

Grid模型介绍

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。

Grid模型实现

具体参见基于TableStore的海量气象格点数据解决方案。

8 多元索引

8.1 简介

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

表格存储的表（Table）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，比如监控数据、日志数据等场景。但之前表格存储只支持主键查询，比如单行读、范围读等，其他类型的查询就很难支持，比如按照非主键列查询，多个列的自由组合查询等。

为了解决上述问题，表格存储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，可以解决上述查询问题，包括但不限于：

- 非主键列的条件查询
- 多列的自由组合查询
- 全文检索
- 地理位置查询
- 前缀查询
- 模糊查询
- 嵌套结构查询

索引区别

除了主表的主键查询，表格存储目前提供了两种加速查询的索引结构：全局二级索引（Global Secondary Index）、多元索引。下表展示了三种索引的区别：

索引类型	原理	场景
表	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。	· 可以确定完整主键（Key）。 · 可以确定主键的前缀（Key prefix）。
全局二级索引	全局二级索引是通过创建一张或多张新表，使用新表的主键列查询，相当于把表的主键查询能力扩展到了不同的列。	· 提前能确定待查询的列，且待查询列的数量比较少。 · 每种查询都能确定完整主键列或主键列的前缀。
多元索引	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。	除表和二级索引适合场景之外的其他所有查询、分析场景。

> **说明:**
> 更详细的索引对比参见TableStore索引功能详解。

注意事项

索引同步

用户为某个表创建了多元索引（Search Index），则后续写入这个表中的数据会先写入表中，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表中读取新写入的数据然后写入多元索引。这个过程是一个异步过程。

采用异步方式创建多元索引不会降低表格存储原有的写入能力。目前索引延迟在秒级别，大部分在10秒以内。在表格存储控制台可以实时查看索引创建的延迟情况。

TTL

多元索引不支持TTL功能，如果原表开通了TTL，则不能创建多元索引。

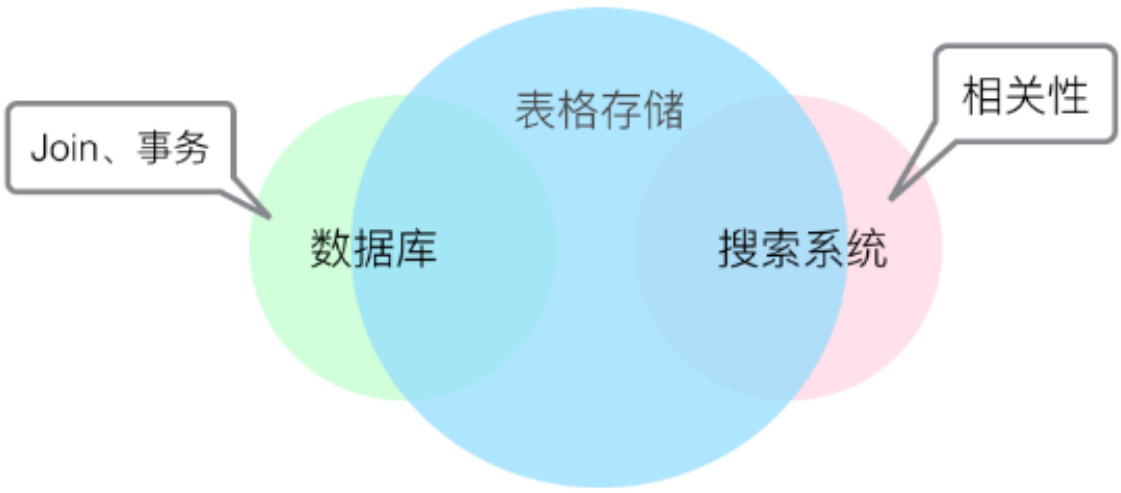
版本

多元索引不支持多版本，如果原表开通了多版本，则不能创建多元索引。

如果在单版本中自定义了每次写入的timestamp，且先写入大版本号，后写入小版本号，这时候先写入的大版本号的索引可能会被后写入的小版本号的索引覆盖。

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。以下是表格存储与数据库及搜索系统的主要的区别：



除了Join、事务和相关性外，数据库和搜索系统中的其他功能表格存储能覆盖，同时具备数据库的数据高可靠性，以及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。

8.2 功能

本文主要为您介绍多元索引的的核心动能。

核心功能

非主键列的查询

原有的表仅支持按完整主键列或主键列前缀查询，无法支持非主键列的查询，在某些场景下就无法满足需求。使用了多元索引后，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的字段的组合方式。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，比如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，比如朋友圈、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地理位置查询功能，包括：

- Near：以一个点为原点，查询指定附近距离圈内的点，比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，则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

表格存储多元索引同样提供分词能力，有了分词能力后就可以支持全文检索。但是表格存储和搜索引擎不一样的地方是，不提供相关性，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，分别是单字分词、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

表格存储多元索引也提供前缀查询，除了中英文外，其他语言也适用，比如前缀查询“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，比如图片标签：某个系统中存储了大量图片，每个图片都有多个实体，比如有房子，有轿车，有人。在每个图片中，这些实体占的位置，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。

数据格式如果用JSON表示：

```
{
 "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。比如电商搜索中，搜索一个“笔记本电脑”可能第一页全是某一个品牌的电脑，这样对用户并不友好，表格存储的去重功能可以避免这种情况。

排序

表提供了主键的字母序排序功能，如果需要按照其他字段排序，那就需要使用多元索引的排序能力了，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。默认是按照表中的主键顺序返回结果。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

目前表格存储还不支持SQL，但是SQL中的大部分功能已经可以在多元索引中找到相匹配的功能：

SQL	多元索引	是否支持
Show	API：DescribeSearchIndex	支持
Select	参数：ColumnsToGet	支持
From	参数：index name	已经支持单索引，多索引还未支持
Where	Query：TermQuery等各种Query	支持
Order by	参数：sort	支持
Limit	参数：limit	支持
Delete	API：先Query再DeleteRow	支持
Like	Query：wildcard query	支持
And	参数：operator = and	支持
Or	参数：operator = or	支持
Not	Query：bool query	支持
Between	Query：range query	支持
Null	ExistQuery	支持

8.3 使用多元索引

8.3.1 概述

本文主要为您介绍多元索引的接口、字段、查询以及计费。

SDK

您可以使用以下语言的SDK实现多元索引功能。

- Java SDK
- Python SDK
- Go SDK
- Node.js SDK
- .NET SDK

接口

名称	API	说明
创建	CreateSearchIndex	创建一个多元索引
描述	DescribeSearchIndex	获取多元索引详细描述信息
列表	ListSearchIndex	列出多元索引的列表
删除	DeleteSearchIndex	删除某个多元索引
查询	Search	查询接口

字段

表格存储多元索引的字段值来源于表中同名字段的值，两者的字段类型必须相匹配，匹配规则如下:

多元索引中字段类型	对应表中字段类型	描述
Long	Integer	64位长整型
Double	Double	64位长浮点数
Boolean	Boolean	bool值
Keyword	String	不可分词字符串
Text	String	可分词字符串或文本，详见分词
Geopoint	String	位置点坐标信息，格式：维度,经度，维度在前，经度在后，比如"35.8,-45.91"
Nested	String	嵌套类型，比如"{["a": 1], ["a": 3]}"

注意:

上述表中的类型，必须一一对应，否则数据会被当做脏数据丢弃，尤其是GeoPoint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，会出现数据在表中查到，但是在多元索引中查询不到的情况。

多元索引字段除了类型外，还有一些附加属性：

属性	类型	名称	说明
Index	Boolean	是否索引	· 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。 · 如果没索引，则不能按照该列进行查询。
EnableSortAndAgg	Boolean	是否排序	· 如果为true，则可以使用该列进行排序。 · 如果为false，则不能使用该列排序。
Store	Boolean	是否附加存储	如果为true，则会在索引中附加存储该列的原始值，查询时如果要读取该列的值，会优先从索引中直接读取，无须反查主表，使用后查询性能更优和更稳定。

属性	类型	名称	说明
IsArray	Boolean	是否数组	· 如果为true，则表示该列是一个数组，在写入时，也必须按照json数组格式写入，比如["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。

说明:

Array和Nested的区别详见Nested和Array对比。

字段类型和字段属性相交情况如下：

类型	Index	EnableSortAndAgg	Store	Array
Long	支持	支持	支持	支持
Double	支持	支持	支持	支持
Boolean	支持	支持	支持	支持
keyword	支持	支持	支持	支持
Text	支持	不支持	支持	支持
Geopoint	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

查询

查询时需要指定一个SearchRequest，参数如下：

参数	类型	描述
offset	Integer	本次查询的开始位置。

参数	类型	描述
limit	Integer	本次查询需要返回的最大数量。
getTotalCount	Boolean	是否返回匹配的总行数，默认关闭，加上后会影响查询性能。
Sort	Sort	指定排序的字段和方式。
collapse	Collapse，唯一的参数是设置字段名。	返回结果中按照哪个字段折叠。
query	Query	具体的Query类型，完整的query列表如下表：

名称	Query	说明
匹配所有行	MatchAllQuery	常用于查询表中数据总行数。
匹配查询	MatchQuery	会先对query内容做分词，然后查询分词后的结构，不同分词直接的关系是OR。
短语匹配查询	MatchPhraseQuery	类似于MatchQuery，但是分词后的多个词必须在文档中相邻才算匹配。
精确查询	TermQuery	不分词，精确查找query的内容，一般用于字符串完全匹配。
多词精确查询	TermsQuery	类似于TermQuery，但是可以一次指定多个值，类似于SQL中的`in`。
前缀查询	PrefixQuery	查询query中的值为前缀的数据。
范围查询	RangeQuery	范围查询。
通配符查询	WildcardQuery	通配符查询，类似于SQL中的like。
复合条件组合查询	BoolQuery	多个条件组合查询，组合关系支持And、Or、Not等。
地理边界框查询	GeoBoundingBoxQuery	根据一个矩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矩形范围内时，满足查询条件。

名称	Query	说明
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。
地理多边形查询	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内时，满足查询条件。

计费

具体参见计量项和计费说明。

8.3.2 创建多元索引

如果要在某张表上使用多元索引功能，首先需要在这张表上创建一个多元索引，一张表可以创建多个多元索引。

只有在多元索引中包括的字段（包括主键列和属性列）才可以通过Search接口查询到。

定义

创建多元索引的API是CreateSearchIndex，具体参数说明如下：

- TableName：指定为哪一张表创建多元索引。
- IndexName：多元索引的名称。

- IndexSchema：多元索引的Schema定义：
  - IndexSetting
    - RoutingFields：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。
  - FieldSchemas
    - FieldName：必选，String类型，字段名称，必须是表中有的列名。
    - FieldType：必选，字段类型，详见字段类型。
    - Index：可选，Boolean类型，默认为true，表示是否对该字段建立索引。
    - IndexOptions：可选，表示是否存储position，offset等term信息到倒排链中，一般用默认值即可。
    - Analyzer：可选，分词器类型，目前支持多种分词类型，详见分词。
    - Analyzer：可选，分词器类型，目前支持多种分词类型。
    - EnableSortAndAgg：可选，Boolean类型，默认为true，表示是否允许该字段支持排序和统计。
    - Store：可选，Boolean类型，默认为true，表示是否在索引中存储原始值，可以加速查询速度。

释疑

一张表中的数据应该创建几个索引?

比如有一张表有5个字段：id，name、age、city、sex，需要按照name或age或city查询，那么有两种创建多元索引的方式：

- 第一种：一个字段建立一个多元索引

  这时候需要创建3个多元索引，分别是name_index、age_index、city_index，如果要按照城市查询就查询city_index，如果要按照年龄查询就查询age_index。

  如果要查询年龄小于12岁，且城市是成都的学生，这种方式就查询不了了。

  这种实现类似于二级索引，但是在多元索引中并没有任何好处，而且会带来更高的费用，所以不建议按这种方式创建多元索引。

- 第二种：一张表中多个字段创建在一个多元索引中

  这时候只需要创建一个多元索引，名字是student_index，字段包括name、age、city，如果要按照城市查询，就查询student_index中city字段即可，如果要查询年龄，就查询student_index中age字段。

  如果要查询年龄小于12岁，且城市是成都的学生，就查询student_index中age和city字段即可。

  这种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。我们极力推荐大家使用这种方式。

限制

1. 创建索引的时效性

当创建完多元索引后需要等几秒钟后才能使用，但是这个过程中不妨碍数据写入，只会影响索引元信息的查询和索引查询。

2. 数量限制

详见使用限制。

示例

```
/**
 *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(
KEYWORD)和整型(LONG)。
 */
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(TABLE_NAME); // 设置表名
    request.setIndexName(INDEX_NAME); // 设置索引名
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD) // 设置字
段名、类型
                    .setIndex(true) // 设置开启索引
                    .setEnableSortAndAgg(true), // 设置开启排序和统计功能
            new FieldSchema("Col_Long", FieldType.LONG)
                    .setIndex(true)
                    .setEnableSortAndAgg(true)));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); // 调用client创建SearchIndex
```

```
}
```

8.3.3 查询多元索引描述信息

如果要在某张表上使用多元索引功能，首先需要在这张表上创建一个多元索引，一张表可以创建多个多元索引。

定义

名称：DescribeSearchIndex

参数：

- TableName：表名，指定查询哪张表对应的Index。
- IndexName：索引名，指定查询哪个索引。

示例

```
private static DescribeSearchIndexResponse describeSearchIndex(
SyncClient client) {
    DescribeSearchIndexRequest request = new DescribeSearchIndexR
equest();
    request.setTableName(TABLE_NAME); // 设置表名
    request.setIndexName(INDEX_NAME); // 设置索引名
    DescribeSearchIndexResponse response = client.describeSearchIndex(
request);
    System.out.println(response.jsonize()); // 输出response的详细信息
    return response;
}
```

8.3.4 列出多元索引列表

查询某个实例，或者某张表关联的所有多元索引的列表信息。

定义

名称：ListSearchIndex

参数：

- TableName：可选，表名，如果为空，则返回该实例下已经开通的所有index列表。如果指定了表，则返回该表关联的多元索引列表。

示例

```
private static List<SearchIndexInfo> listSearchIndex(SyncClient client
) {
    ListSearchIndexRequest request = new ListSearchIndexRequest();
    request.setTableName(TABLE_NAME); // 设置表名
    return client.listSearchIndex(request).getIndexInfos(); // 返回表下
所有SearchIndex
```

```
}
```

8.3.5 删除多元索引

删除一个多元索引。

定义

名称：DeleteSearchIndex

参数：

- TableName：表名，需要删除那种表中的索引。
- IndexName：索引名，需要删除的索引名。

示例

```
private static void deleteSearchIndex(SyncClient client) {
    DeleteSearchIndexRequest request = new DeleteSearchIndexRequest();
    request.setTableName(TABLE_NAME); // 设置表名
    request.setIndexName(INDEX_NAME); // 设置索引名
    client.deleteSearchIndex(request); // 调用client删除对应的多元索引
}
```

8.3.6 数组和嵌套类型

多元索引提供了Long、Double、Boolean、Keyword、Text和GeoPoint等基本类型。除了基本类型外，还提供了两种特殊的类型。

一种是数组，数组属于附加类型，可以附加在Long、Double、Boolean、Keyword、Text和GeoPoint等类型之上。比如Long类型 + 数组后，那就是数组长整型，该字段中可以包括多个长整型数字，查询的时候其中任何一个匹配都可以返回该行数据。

另一种是嵌套类型，有点像数组，但是功能更丰富。

数组类型

基本类型的数组形式，比如：

- Long Array: 长整型的数组形式，格式：“[1000, 4, 5555]”。
- Boolean Array:布尔值的数组形式，格式：“[true, false]”。
- Double Array：浮点数的数组形式，格式：“[3.1415926, 0.99]”。
- Keyword Array：字符串的数组形式，格式：“[杭州, 西安]”。
- Text Array：文本的数组形式，格式：“[杭州, 西安]”，对于Text类型是否数组区别不大，一般不这么用。
- GeoPoint Array：地理位置点的数组形式，格式：“[[34.2, 43.0], [21.4，45.2]]”。

数组类型仅是多元索引中的概念，表中还未支持数组，索引对于数组类型的字段，在表中都必须是String类型，对应的多元索引中的类型必须是相应的类型，比如Long、Double等。比如有一个字段price是Double Array类型，那么在表中price必须是string类型，在多元索引中必须是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）可以包含多个子文档，多个子文档保存在一个嵌套类型的列中。对于一个嵌套类型的列，需要指定其子文档的结构，即子文档中包含哪些字段，以及每个字段的属性。以Java代码为例：

```
// 构造子文档的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));

// 将子文档的FieldSchema设置到NESTED列的subfieldSchemas中。
FieldSchema nestedFieldSchema = new FieldSchema("tags", FieldType.
NESTED)
    .setSubFieldSchemas(subFieldSchemas);
```

上面的代码定义了一个嵌套类型列的格式，这列列名为tags，子文档中包含两个字段，其中一个字段名为tagName，类型为字符串类型(KEYWORD)，另一个Field名为score，类型为浮点数(DOUBLE)。

嵌套类型列在写入时是作为字符串类型写到主表的，字符串的格式是json对象的数组。按照上面例子里定义的Nested格式，可以再举一个数据格式的例子，比如：

```
[{"tagName":"tag1", "score":0.8}, {"tagName":"tag2", "score":0.2}]
```

即这一列中包含了两个子文档。需要注意，即使只有一个子文档，也应该按照json数组的格式构造字符串。

嵌套类型的局限性：

1. 使用了嵌套类型的索引不支持IndexSort功能，而IndexSort功能在很多场景下可以带来很大性能提升。
2. 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一样，支持所有查询Query，支持排序以及未来的统计聚合。

如果想了解更多Array和Nested的对比，可以阅读这篇文章：《Array和Nested对比》。

8.3.7 排序

通过多元索引的Search接口查询索引数据时，支持设置排序(Sort)。

多元存储支持多种排序方式(表主键排序、字段值排序、地理距离排序和相关性分数排序)。

如果查询时没有设置排序方式，则默认会根据索引上的IndexSort配置来决定排序方式，默认IndexSort为表主键排序。

目前支持的排序方式：

- ScoreSort

  分数排序。按照查询结果的相关性分数进行排序，适用于有相关性的场景，比如全文索引等。
- PrimaryKeySort

  表主键排序。按照表的主键值大小进行排序。
- FieldSort

  按照某一列的值进行排序。
- GeoDistanceSort

  地理距离排序。按照距离某个地理点的距离进行排序。

8.3.8 分词

多元索引提供了分词能力，如果用户的字段类型设置为Text类型，那么这个字段可以额外设置一个分词参数，指定这个字段按照哪一种方式分词，非Text类型不能设置分词类型。

对于Text类型数据，一般常用于MatchQuery和MatchPhraseQuery查询，少部分场景也会用到TermQuery、TermsQuery、PrefixQuery和WildcardQuery等。

目前支持的分词类型如下：

类型

单字分词

- 名称：single_word
- 适用于：汉语、英语、日语等所有语言文字
- 参数：
  - caseSensitive：是否大小写敏感，默认是false，这时所有英文字母会转换为小写。
  - 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。

当设置字段分词类型为单字分词后，汉语会按照“字”切分，比如“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，比如"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。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，那么可以设置参数caseSensitive为true。

对于数字和英文字母连接在一起的词，比如商品型号等，也会按照空格或标点符号切分，结果就是数字英文不会拆分开，比如“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。如果需要将英文和数字拆分开，可以设置参数delimitWord为true，这样“iphone6”会被拆分成“iphone”和“6”。

分隔符分词

- 名称：split
- 适用于：汉语、英语、日语等所有语言文字
- 参数：
  - delimiter：分隔符，默认是空白字符，可以自定义分隔符。

我们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，这时候产品提供的分词类型就无法满足用户需求了。

为了解决这个问题，我们提供了分隔符分词，也叫自定义法分词，用户自己先按照自己的方式分词后，然后按照特定分隔符分割后写入表格存储。

 说明:
创建多元索引时，该字段分词配置中的分隔符必须和写入数据中的分隔符保持一致，否则可能会查询不到数据。

最小数量语义分词

- 名称：min_word
- 适用于：汉语
- 参数：
  - 无

除了提供字级别的分词外，多元索引还提供语义级别分词，其中最基础的语义分词是最小语义分词，按照语义对Text字段内容分词时，会切分成最小数量的语义词。

比如“梨花茶”会切分成“梨”和“花茶”，切分后的结果没有重合。再比如“中华人民共和国”会被切分成“中华人民共和国”。

一般情况下，全文检索场景中这种分词就可以满足基本需求。

最大数量语义分词

- 名称：max_word
- 适用于：汉语
- 参数：
  - 无

除了最小数量语义分词外，还提供一种更复杂的最大数量语义分词，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。

比如“梨花茶”会切分成“梨花”和“花茶”，切分后的结果没有重合。再比如“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”。

这种分词方式优点是分词后结果更多，查询时命中的概率更大，但是缺点是索引大小会有比较大的膨胀，同时如果使用MatchPhraseQuery查询，因为查询词中也会按照同样方式分词，那么位置信息会重叠，就有可能导致搜索不到，最适合用MatchQuery而非MatchPhraseQuery查询。

模糊分词

- 名称：fuzzy
- 适用于：汉语、英语、日语等各种语言
- 参数：
  - minChars：最小字符切分单元，也就是切分的字符组合中字符数量必须大于等于这个数，建议取值2。
  - maxChars：最大字符切分单元，也就是切分的字符组合中字符数量必须小于等于这个数，建议取值不超过7。
- 限制：
  - Text字段长度不超过32字符，如果超过会将限制之外的字符截断丢弃，只保留前32个字符。

有一种场景是文本内容较短，比如标题、电影名称、书名等，但是需要非常快速的查询到结果，比如用在下拉提示等功能中，针对这种需求，多元索引提供了模糊分词，对文本内容进行Ngram分词，结果介于minChars和maxChars之间。

优势是可以以很低的延迟返回结果，缺点是索引膨胀会很大，所以适用于短文本，不适用于长文本。

对比

下面从几个关键维度对不同分词做一个比较。

	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	中	小	小	大	巨大
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	32
召回率	高	低	低	中	中

8.3.9 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。

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(数据量很大时为估计值),
     * 如果只为了取行数，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);

    /**
    * 设置返回命中的总行数。
    */
    searchRequest.setGetTotalCount(true);
    SearchResponse resp = client.search(searchRequest);
    /**
     * 判断返回的结果是否是完整的，当isAllSuccess为false时，代表可能有部分节点查询失败，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()); // 总行数
```

```
    System.out.println(resp.getRequestId());
}
```

8.3.10 匹配查询

MatchQuery一般应用于全文检索场景，作用于Text类型字段。不管是Text还是MatchQuery的内容，都会按照配置好的分词器做切分。如果是Query，则会按照切分好后的Term去查询。

比如某一行数据的title字段值是：“杭州西湖风景区”，使用单字分词，那么如果MatchQuery中的查询词是：“湖风”，就可以命中这一行数据。

参数

- fieldName：字段名，在哪个字段上查询。
- text：查询词，该词会被分词成多个Term。
- minimumShouldMatch：只有某一行数据的fieldName字段的值中至少包括了minimumShouldMatch个term才会返回这一行数据，否则认为不命中。
- operator：关系符，默认是Or，也就是说分词后的多个term只要有部分命中，就认为命中。如果是And，则需要分词后的所有Term都在字段值中有才算命中。

示例

```
/**
 * 查询表中Col_Keyword这一列的值能够匹配"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"); // 设置要匹配的字段
    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
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME, searchQuery);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount());
    System.out.println("Row: " + resp.getRows()); // 不设置columnsToGet，默认只返回主键

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    searchRequest.setColumnsToGet(columnsToGet);

    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());
}
```

8.3.11 短语匹配查询

与MatchQuery类似，但是分词后的多个Term的位置关系会被考虑，分词后的多个Term必须在行数据中以同样的顺序和位置存在才算命中。

比如字段值是：“杭州西湖风景区”，Query中是：“杭州风景区”，如果是MatchQuery，则可以命中该行数据，但是如果是MatchPhraseQuery，则不能命中该行数据，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）。

参数

- fieldName：字段名。
- text：查询词，会被分词器分词多个Term，然后再去查询。

示例

```
/**
 * 查询表中Col_Text这一列的值能够匹配"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"); // 设置要匹配的字段
    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
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME, searchQuery);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount());
    System.out.println("Row: " + resp.getRows()); // 默认只返回主键

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    searchRequest.setColumnsToGet(columnsToGet);

    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());
}
```

8.3.12 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，但是对于分词字符串类型，只要分词后有词条可以精确匹配即可。比如某个分词字符串类型的字段，值

为“tablestore is cool”，假设分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。

参数

- fieldName：字段名。
- term：查询词，该词不会被分词，会被当做完整次去匹配。

示例

```
/**
 * 查询表中Col_Keyword这一列精确匹配"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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
}
```

8.3.13 多值精确查询

类似于TermQuery，但是可以指定多个值，等价于SQL中的In。

参数

fieldName：字段名。

terms：多个查询词，只要有一个词相等，该行数据就会被返回。

示例

```
/**
 * 查询表中Col_Keyword这一列精确匹配"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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
}
```

8.3.14 前缀查询

根据前缀条件查询表中的数据，对于分词字符串类型(TEXT)，只要分词后的词条中有词条满足前缀条件即可。

参数

- fieldName：字段名
- prefix：前缀值

示例

```
/**
 * 查询表中Col_Keyword这一列前缀为"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
    prefixQuery.setFieldName("Col_Keyword");
    prefixQuery.setPrefix("hangzhou");
    searchQuery.setQuery(prefixQuery);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
```

8.3.15 范围查询

根据范围条件查询表中的数据，对于分词字符串类型(TEXT)，只要分词后的词条中有词条满足范围条件即可。

参数

- fieldName：字段名
- from：其实位置的值
- to：结束位置的值
- includeLow：Boolean值，结果中是否需要包括from值
- includeUpper：Boolean值，结果中是否需要包括to值

示例

```
/**
 * 查询表中Col_Long这一列大于3的数据，结果按照Col_Long这一列的值逆序排序。
 * @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");  // 设置针对哪个字段
    rangeQuery.greaterThan(ColumnValue.fromLong(3));  // 设置该字段的范围
条件，大于3
    searchQuery.setQuery(rangeQuery);

    // 设置按照Col_Long这一列逆序排序
    FieldSort fieldSort = new FieldSort("Col_Long");
    fieldSort.setOrder(SortOrder.DESC);
    searchQuery.setSort(new Sort(Arrays.asList((Sort.Sorter)fieldSort
)));

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    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());

    /**
     * 设置SearchAfter后重新查询，设置SearchAfter=5，则按照Col_Long逆序顺序
后，从Col_Long=5之后的行开始返回（相当于设置条件为小于5）。
     */
    searchQuery.setSearchAfter(new SearchAfter(Arrays.asList(
ColumnValue.fromLong(5))));
    searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME,
searchQuery);
    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());
```

```
}
```

8.3.16 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。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。比如查询“table*e”，可以匹配到“tablestore”。

参数

- fieldName：字段名。
- value：含有通配符的值。目前支持两种通配符：“*”和“？”。不能以“*”开头，且字符长度不能超过10字节。

示例

```
/**
 * 使用通配符查询，查询表中Col_Keyword这一列的值匹配"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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
}
```

8.3.17 多字段自由组合查询

复合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。

可以设置多种组合方式，比如设置多个子查询条件为mustQueries，则要求这些子查询条件必须都满足；设置多个子查询条件为mustNotQueries时，要求这些子查询条件都不能满足。

参数

- mustQuerys：多个Query列表，必须要满足的条件，等价于 And 操作符。

- mustNotQuerys：多个Query列表，必须不能满足的条件，等价于 Not 操作符。
- shouldQuerys：可以满足，也可以不满足，如果满足则整体的相关性分数更高，等价于 Or 操作符。
- minimumShouldMatch：至少要有多少个should query满足才算命中这行数据。

示例

```
/**
 * 通过BoolQuery进行复合条件查询。
 * @param client
 */
public static void boolQuery(SyncClient client) {
    /**
     * 查询条件一：RangeQuery，Col_Long这一列的值要大于3
     */
    RangeQuery rangeQuery = new RangeQuery();
    rangeQuery.setFieldName("Col_Long");
    rangeQuery.greaterThan(ColumnValue.fromLong(3));

    /**
     * 查询条件二：MatchQuery，Col_Keyword这一列的值要匹配"hangzhou"
     */
    MatchQuery matchQuery = new MatchQuery(); // 设置查询类型为
MatchQuery
    matchQuery.setFieldName("Col_Keyword"); // 设置要匹配的字段
    matchQuery.setText("hangzhou"); // 设置要匹配的值

    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * 构造一个BoolQuery，设置查询条件是必须同时满足"条件一"和"条件二"
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustQueries(Arrays.asList(rangeQuery, matchQuery
));
        searchQuery.setQuery(boolQuery);
        SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
        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());
    }

    {
        /**
         * 构造一个BoolQuery，设置查询条件是至少满足"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); // 设置最少满足一个条件
        searchQuery.setQuery(boolQuery);
        SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
        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());
```

```
    }
}
```

8.3.18 嵌套查询

嵌套类型查询。用于查询嵌套类型中子文档的数据。

嵌套类型不能直接查询，需要通过NestedQuery来包装一下，NestedQuery中需要指定嵌套类型的字段路径(path)以及一个子query(可以是任意query)。

嵌套类型查询只能查询嵌套类型的字段。

可以在一个请求中同时查询普通字段和嵌套类型字段。

参数

- path：路径名，是嵌套类型字段内容的树状路径。比如news.title，表示news嵌套类型中的title嵌套类型。
- query：nested字段中子字段上的query，可以是任意类型query。

示例

```
/**
 * 有一类型为NESTED的列，子文档包含nested_1和nested_2两列，现在查询col_nested
.nested_1为"tablestore"的数据。
 * @param client
 */
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NESTED字段路径
    TermQuery termQuery = new TermQuery(); // 构造NestedQuery的子查询
    termQuery.setFieldName("col_nested.nested_1"); // 设置字段名，注意带
有Nested列的前缀
    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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
```

```
}
```

8.3.19 地理距离查询

地理距离查询。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

- fieldName：字段名。
- centerPoint：中心地理坐标点，是一个经纬度值。
- distanceInMeter：Double类型，距离中心点的距离，单位是米。

示例

```
/**
 * 查询表中Col_GeoPoint这一列的值距离中心点不超过一定距离的数据。
 * @param client
 */
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); // 设置到中心点的距离条
件，不超过10000米
    searchQuery.setQuery(geoDistanceQuery);

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setColumns(Arrays.asList("Col_GeoPoint"));  //设置返回
Col_GeoPoint这一列
    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());
}
```

8.3.20 地理长方形范围查询

地理边界框查询。根据一个矩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矩形范围内时，满足查询条件。

参数

- fieldName：字段名。
- topLeft：矩形框的左上角的坐标。
- bottomRight：矩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矩形。

示例

```
/**
 * Col_GeoPoint是GeoPoint类型，查询表中Col_GeoPoint这一列的值在左上角为"10,0
", 右下角为"0,10"的矩形范围内的数据。
 * @param client
 */
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"); // 设置比较哪个字段
的值
    geoBoundingBoxQuery.setTopLeft("10,0"); // 设置矩形左上角
    geoBoundingBoxQuery.setBottomRight("0,10"); // 设置矩形右下角
    searchQuery.setQuery(geoBoundingBoxQuery);

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setColumns(Arrays.asList("Col_GeoPoint"));  //设置返回
Col_GeoPoint这一列
    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());
}
```

8.3.21 地理多边形范围查询

地理多边形查询。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内时，满足查询条件。

参数

- fieldName：字段名。
- points：组成多边形的距离坐标点。

示例

```
/**
 * 查询表中Col_GeoPoint这一列的值在一个给定多边形范围内的数据。
 * @param client
 */
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);

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
```

```
    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setColumns(Arrays.asList("Col_GeoPoint"));  //设置返回
Col_GeoPoint这一列
    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());
}
```

8.3.22 列存在查询

列存在查询（ExistQuery），也叫NULL查询或者空值查询。一般用于稀疏数据中，判断某一行的某一列是否存在，比如查询所有数据中，address列不为空的行有哪些。

说明:

如果需要查询某一列为空，则需要和bool query中的must_not语句结合。

参数

fieldName：列名。

示例

```
/**
 * 使用列存在查询，查询表中address这一列的不为空的数据
 * @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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
```

```
}
```

8.4 使用限制

文本主要为您介绍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	200	可被索引的字段数
EnableSortAndAgg字段数量	100	可被排序和统计聚合的字段数
Nested嵌套层数	1	最多嵌套一层nested
Nested字段数量	25	嵌套中子字段的个数
表主键列长度之和	1000	所有PK列的长度累加后不超过1000
表主键列中String长度	1000	所有PK列的长度累加后不超过1000
表属性列中String长度（索引成Keyword）	4KB	无
表属性列中String长度（索引成Text）	2MB	同Table中属性列长度限制
通配符查询的Query长度	10	不超过10字符

Search

名称	最大值	说明
offset + limit	2000	如果超过请使用next_token
timeout	10s	-
CU	10万	· 除扫描、分析类外。 · 超过请工单联系我们。

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。 · 针对于分析型查询(AP)或者Text类型查询，由于每个请求耗时会很长，每个索引允许最多占用8核CPU。 · 上述是默认限制，如果业务需求超过上述限制，可以提工单联系我们。

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。 · 超过请工单联系我们。
同步延迟	10s	· 写入速率稳定情况下10秒内。 · 99%情况下在1分钟内。 · 新建索引最多会有1分钟的初始化时间。
行数	100亿	超过请工单联系我们。
总大小	10TB	超过请工单联系我们。

其他限制

名称	值
功能开放区域	北京、上海、杭州、深圳、新加坡、印度、香港、张家口
即将开放区域	美国（硅谷）

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。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，我们的工程师会在后续功能开发中优先考虑您的需求。

8.5 实践

本文主要为您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
- 爬虫数据
- 物联网元数据
- 订单系统

9 全局二级索引

9.1 使用前须知

本文为您介绍全局二级索引的基本概念、使用限制以及注意事项。

基本概念

名词	描述
索引表	对主表某些列数据的索引，只能读不能写。
预定义列	表格存储为Schema-free模型，原则上一行数据可以写入任意列，无需在schema中指定。但是也可以在建表时预先定义一些列以及其类型。
单列索引	只为某一个列建立索引。
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	被映射到索引表非PK列中的主表预定义列。
索引列补齐	自动将没有出现在用户指定索引列中的主表PK列补充到索引表PK中。

限制

- 同一张主表下，最多建立5张索引表。（超出索引表数目限制后，新建索引表将失败）
- 同一张主表下，最多建立15个预定义列。（超出预定义列数目限制后，建主表将失败）
- 对于一张索引表，其索引列最多有4列。（为主表PK以及主表预定义列的任意组合，超出限制后，建表将会失败）
- 对于一张索引表，其属性列最多有8列。（超出限制后，建表将会失败）
- 索引列的类型为整型、字符串、二进制，与主表PK列的约束相同。
- 多个列组合成索引，大小限制与主表PK列相同。
- 类型为字符串或二进制的列，作为索引表的属性列时，限制与主表相同。
- 暂不支持TTL表建立索引，有需求请钉钉联系表格存储技术支持。
- 不支持在使用多版本功能的表上建立索引。（如果表打开了多版本，建索引会失败；如果有索引，打开主表多版本功能会失败）
- 有索引的主表上，写入数据时，不允许自定义版本。（否则主表写将会失败）
- 索引表上不允许使用Stream功能。

- 有索引表的主表上，同一个Batch写中，同一行（主键相同）不允许重复存在。（否则主表写将会失败）

注意事项

- 对于每张索引表，系统会自动进行索引列补齐。在对索引表进行扫描时，您需要注意填充对应PK列的范围（一般为负无穷到正无穷）。例如：主表有PK0、PK1两列PK，Defined0一列预定义列。

  如果您指定在Defined0列上建立索引，则系统会将索引表的PK生成为Defined0、 PK0 、 PK1。您可以指定在Defined0列及PK1列上建立索引，则生成索引表的PK为Defined0 、PK1、PK0。您还可以在PK列上建立索引，则生成索引表的PK为PK1、PK0。您在建立索引表时，只需要指定您需要的索引列，其它列会由系统自动添加。例如主表有PK0、PK1两列PK，Defined0作为预定义列：

  - 如果在Defined0上建立索引，那么生成的索引表PK将会是Defined0、PK0、PK1三列。
  - 如果在PK1上建立索引，那么生成的索引表PK将会是PK1, PK0两列。
- 选择主表的哪些预定义列作为主表的属性列。将主表的一列预定义列作为索引表的属性列后，查询时不用反查主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主表。您可以根据您的查询模式以及成本的考虑，作出相应的选择。
- 不建议把时间相关列作为索引表PK的第一列，这样可能导致索引表更新速度变慢。建议将时间列进行哈希，然后在哈希后的列上建立索引，如果有类似需求可以钉钉联系表格存储技术支持一同讨论表结构。
- 不建议取值范围非常小，甚至可枚举的列作为索引表PK的第一列。例如性别，这样将导致索引表水平扩展能力受限，从而影响索引表写入性能。

9.2 功能介绍

本文主要为您介绍全局二级索引的功能。

- 主表与索引表之间异步同步，正常情况下同步延迟达到毫秒级别。
- 支持单列索引及组合索引，支持索引表带有属性列（Covered Indexes）。主表可以预先定义若干列（称为预定义列），可以对任意预定义列和主表 PK 列进行索引，可以指定主表的若干个预定义列作为索引表的属性列（索引表也可以不包含任何属性列）。当指定了主表的某些预定义

列作为索引表的属性列时，读索引表可以直接得到主表中对应预定义列的值，无需反查主表。例如：主表有 PK0、 PK1、 PK2 三列主键，Defined0、Defined1、Defined2 三列预定义列。

- 索引列可以是 PK2，没有属性列。
- 索引列可以是 PK2，属性列可以是 Defined0。
- 索引列可以是 PK1、PK2，没有属性列。
- 索引列可以是 PK1、PK2，把 Defined0 作为属性列。
- 索引列可以是 PK2、PK1、PK0，把 Defined0、Defined1、Defined2 作为属性列。
- 索引列可以是 Defined0，没有属性列。
- 索引列可以是 Defined0、PK1，把 Defined1 作为属性列。
- 索引列可以是 Defined1、Defined0，没有属性列。
- 索引列可以是 Defined1、 Defined0，把 Defined2 作为属性列。

· 支持稀疏索引（Sparse Indexes）：如果主表的某个预定义列作为索引表的属性列，当主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。但如果部分索引列缺失，则不会为此行建立索引。例如：主表有 PK0、PK1、PK2 三列主键，Defined0、Defined1、Defined2 三列预定义列，设置索引表 PK 为 Defined0、Defined1，索引表属性列为 Defined2。

- 当主表某行中，只包含 Defined0、Defined1 这两列，不包含 Defined2 列时，会为此行建立索引。
- 当主表某行中，只包含 Defined0、Defined2 这两列，不包含 Defined1 列时，不会为此行建立索引。

· 支持在已经存在的主表上进行创建、删除索引的操作。 后续版本将支持新建的索引表中包含主表中的存量数据。
· 查索引表不会自动反查主表，用户需要自行反查。后续版本将支持自动根据索引表反查主表的功能。

9.3 使用场景

全局二级索引支持在指定列上建立索引，生成的索引表中数据按您指定的索引列进行排序，主表的每一个写入都将自动异步同步到索引表。您只向主表中写入数据，根据索引表进行查询，在许多场景下，将极大的提高查询的效率。本文为您介绍电话话单查询场景下如何使用全局二级索引。

以我们常见的电话话单查询为例，建立主表如下：

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
345678	1532574861	123456	100	2
456789	1532584054	345678	200	3

- `CellNumber`、`StartTime`作为表的联合主键，分别代表主叫号码与通话发生时间。
- `CalleNumber`、`Duration`、`BaseStationNumber`三列为表的预定义列，分别代表被叫号码、通话时长、基站号码。

每次用户通话结束后，都会将此次通话的信息记录到该表中。可以分别在被叫号码，基站号码列上建立二级索引，来满足不同角度的查询需求（具体建立索引的示例代码见附录）。

假设有以下几种查询需求：

- 查询号码`234567`的所有主叫话单。

  由于表格存储为全局有序模型，所有行按主键进行排序，并且提供顺序扫描(`getRange`)接口，所以只需要在调用`getRange`接口时，将PK0列的最大及最小值均设置为`234567`，PK1列（通话发生时间）的最小值设置为`0`，最大值设置为`INT_MAX`，对主表进行扫描即可：

```
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.fromLong(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.fromLong(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(endPrimary
KeyBuilder.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(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }

        // 若nextStartPrimaryKey不为null，则继续读取.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
        } 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上进行查询。

  索引表IndexOnBeCalledNumber：

PK0	PK1	PK2
CalledNumber	CellNumber	StartTime
123456	234567	1532574734
123456	345678	1532574795
123456	345678	1532574861
654321	123456	1532574644
765432	234567	1532574714
345678	456789	1532584054

说明:

系统会自动进行索引列补齐。即把主表的PK添加到索引列后面，共同作为索引表的PK。所以索引表中有三列PK。

由于索引表IndexOnBeCalledNumber是按被叫号码作为主键，可以直接扫描索引表得到结果：

```
private static void getRangeFromIndexTable(SyncClient client, long
cellNumber) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
eryCriteria(INDEX0_NAME);

    // 构造主键
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1,
PrimaryKeyValue.fromLong(cellNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
PrimaryKeyValue.INF_MIN);
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.INF_MAX);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrima
ryKeyBuilder.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(endPrimary
KeyBuilder.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(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }

        // 若nextStartPrimaryKey不为null，则继续读取.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
        } 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，但是查询不仅把基站号码列作为条件，同时把通话发生时间列作为查询条件，因此我们可以在基站号码和通话发生时间列上建立组合索引。

  索引表IndexOnBaseStation1：

PK0	PK1	PK2
BaseStationNumber	StartTime	CellNumber
001	1532574644	123456
001	1532574714	234567
002	1532574795	345678
002	1532574861	345678
003	1532574734	234567
003	1532584054	456789

  然后在IndexOnBaseStation1索引表上进行查询：

```
private static void getRangeFromIndexTable(SyncClient client,
                                           long baseStationNumber,
                                           long startTime) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
eryCriteria(INDEX1_NAME);

    // 构造主键
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3,
PrimaryKeyValue.fromLong(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.fromLong(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
PrimaryKeyValue.INF_MIN);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrima
ryKeyBuilder.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(endPrimary
KeyBuilder.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(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }

        // 若nextStartPrimaryKey不为null，则继续读取.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
        } 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。

  在该查询中不仅把基站号码列与通话发生时间列作为查询条件，而且只把通话时长列作为返回结果。您可以上一个查询中的索引，查索引表成功后反查主表得到通话时长：

```
private static void getRowFromIndexAndMainTable(SyncClient client,
                                                long baseStatio
nNumber,
                                                long startTime,
                                                long endTime,
                                                String colName) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
eryCriteria(INDEX1_NAME);

    // 构造主键
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3,
PrimaryKeyValue.fromLong(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.fromLong(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
PrimaryKeyValue.INF_MIN);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrima
ryKeyBuilder.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(endPrimary
KeyBuilder.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" + strEndTime + "的所有话单通话时长:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new
GetRangeRequest(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn mainCalledNumber = curIndexPrimaryKey.
getPrimaryKeyColumn(PRIMARY_KEY_NAME_1);
            PrimaryKeyColumn callStartTime = curIndexPrimaryKey.
getPrimaryKeyColumn(PRIMARY_KEY_NAME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder
.createPrimaryKeyBuilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KE
Y_NAME_1, mainCalledNumber.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KE
Y_NAME_2, callStartTime.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build
(); // 构造主表PK

            // 反查主表
            SingleRowQueryCriteria criteria = new SingleRowQ
ueryCriteria(TABLE_NAME, mainTablePK);
            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);
        }

        // 若nextStartPrimaryKey不为null，则继续读取.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
        } else {
            break;
        }
    }
}
```

为了提高查询效率，可以在基站号码列与通话发生时间列上建立组合索引，并把通话时长列作为索引表的属性列：

数据库中的记录将会如下所示：

索引表`IndexOnBaseStation2`：

PK0	PK1	PK2	Defined0
BaseStationNumber	StartTime	CellNumber	Duration

PK0	PK1	PK2	Defined0
001	1532574644	123456	60
001	1532574714	234567	10
002	1532574795	345678	5
002	1532574861	345678	100
003	1532574734	234567	20
003	1532584054	456789	200

然后在`IndexOnBaseStation2`索引表上进行查询：

```
private static void getRangeFromIndexTable(SyncClient client,
                                           long baseStationNumber,
                                           long startTime,
                                           long endTime,
                                           String colName) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
eryCriteria(INDEX2_NAME);

    // 构造主键
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_3,
PrimaryKeyValue.fromLong(baseStationNumber));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.fromLong(startTime));
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
PrimaryKeyValue.INF_MIN);
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrima
ryKeyBuilder.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(endPrimary
KeyBuilder.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" + strEndTime + "的所有话单通话时长:");
    while (true) {
```

```
        GetRangeResponse getRangeResponse = client.getRange(new
GetRangeRequest(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }

        // 若nextStartPrimaryKey不为null，则继续读取.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
        } else {
            break;
        }
    }
}
```

由此可见，如果不把通话时长列作为索引表的属性列，在每次查询时都需先从索引表中解出主表的PK，然后对主表进行随机读。当然，把通话时长列作为索引表的属性列后，该列被同时存储在了主表及索引表中，增加了总的存储空间占用。

- 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，最小通话时长。

  相对于上一条查询，这里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。用户可以使用与上条查询相同的的查询方式，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SQL-on-OTS，省去客户端的计算，直接使用SQL语句返回最终统计结果，SQL-on-OTS的开通使用文档，请参见可见OLAP on Table Store：基于Data Lake Analytics的Serverless SQL大数据分析。其兼容绝大多数MySql语法，可以更方便的进行更复杂的、更贴近用户业务逻辑的计算。

9.4 接口说明

本文档以Java SDK为例，对createTable、scanFromIndex等接口进行详细说明。

- 创建主表的同时创建索引表。

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta(TABLE_NAME);
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KE
Y_NAME_1, PrimaryKeyType.STRING)); // 为主表设置PK列
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KE
Y_NAME_2, PrimaryKeyType.INTEGER)); // 为主表设置PK列
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_CO
L_NAME_1, DefinedColumnType.STRING)); // 为主表设置预定义列
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_CO
L_NAME_2, DefinedColumnType.INTEGER)); // 为主表设置预定义列
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_CO
L_NAME_3, DefinedColumnType.INTEGER)); // 为主表设置预定义列

    int timeToLive = -1; // 数据过期时间设置为永不过期
    int maxVersions = 1; // 最大版本数为1（带索引表的主表只支持版本数为1）
```

```
    TableOptions tableOptions = new TableOptions(timeToLive,
maxVersions);

    ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME); // 新建索引表Meta
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // 指定主表的
DEFINED_COL_NAME_1列作为索引表的PK
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); // 指定主表的
DEFINED_COL_NAME_2列作为索引表的属性列
    indexMetas.add(indexMeta); // 添加索引表到主表

    CreateTableRequest request = new CreateTableRequest(tableMeta,
tableOptions, indexMetas); // 创建主表

    client.createTable(request);
}
```

- 为已经存在的主表添加索引表。

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME); // 新建索引Meta
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_2); // 指定
DEFINED_COL_NAME_2列为索引表的第一列PK
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // 指定
DEFINED_COL_NAME_1列为索引表的第二列PK
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME,
indexMeta, false); // 将索引表添加到主表上
    client.createIndex(request); // 创建索引表
}
```

说明:

当前在添加索引表时，尚不支持索引表中包含主表中已经存在的数据。即新建索引表中将只包含主表从创建索引表开始时的增量数据。如果有对存量数据建索引的需求，请钉钉联系表格存储技术支持。

- 删除索引表。

```
private static void deleteIndex(SyncClient client) {
    DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME,
INDEX_NAME); // 指定主表名称及索引表名称
    client.deleteIndex(request); // 删除索引表
}
```

- 读取索引表中数据。

  需要返回的属性列在索引表中，您可以直接读取索引表：

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
eryCriteria(INDEX_NAME); // 设置索引表名

    // 设置起始主键
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    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); // 主表PK最小值
```

```
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.INF_MIN); // 主表PK最小值
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrima
ryKeyBuilder.build());

    // 设置结束主键
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1,
PrimaryKeyValue.INF_MAX); // 设置需要读取的索引列最大值
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
PrimaryKeyValue.INF_MAX); // 主表PK最大值
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.INF_MAX); // 主表PK最大值
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimary
KeyBuilder.build());

    rangeRowQueryCriteria.setMaxVersions(1);

    System.out.println("扫描索引表的结果为:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new
GetRangeRequest(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }

        // 若nextStartPrimaryKey不为null, 则继续读取.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
        } else {
            break;
        }
    }
}
```

需要返回的属性列不在索引表中，您需要反查主表：

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQu
eryCriteria(INDEX_NAME); // 设置索引表名

    // 设置起始主键
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    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); // 主表PK最小值
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.INF_MIN); // 主表PK最小值
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrima
ryKeyBuilder.build());

    // 设置结束主键
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1,
PrimaryKeyValue.INF_MAX); // 设置需要读取的索引列最大值
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1,
PrimaryKeyValue.INF_MAX); // 主表PK最大值
```

```
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2,
PrimaryKeyValue.INF_MAX); // 主表PK最大值
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimary
KeyBuilder.build());

    rangeRowQueryCriteria.setMaxVersions(1);

    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new
GetRangeRequest(rangeRowQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimary
KeyColumn(PRIMARY_KEY_NAME1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimary
KeyColumn(PRIMARY_KEY_NAME2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder
.createPrimaryKeyBuilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1
, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2
, ke2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build
(); // 根据索引表PK构造主表PK

            // 反查主表
            SingleRowQueryCriteria criteria = new SingleRowQ
ueryCriteria(TABLE_NAME, mainTablePK);
            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);
        }

        // 若nextStartPrimaryKey不为null，则继续读取.
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
        } else {
            break;
        }
    }
}
```

9.5 API/SDK

本文主要为您介绍全局二级索引的API及SDK。

创建主表时同时创建索引表

CreateTable用于创建主表时指定预定义列，同时创建索引表。

后续写入数据后，索引表中将包含主表中所有符合索引构建条件的数据。详情请参见CreateTable。

单独创建索引表

CreateIndex用于在已有的主表上创建索引表。

详情请参见CreateIndex。

说明:
当前版本中使用CreateIndex单独创建的索引表中，不包含主表在执行CreateIndex前已经写入的数据，后续版本会提供支持。

删除索引表

DeleteIndex用于删除指定主表下面的指定索引表，此主表上的其它索引表不受影响。

详情请参见DeleteIndex。

删除主表

DeleteTable用于删除主表，且主表下面的所有索引表都会被删除。详情请参见DeleteTable。

9.6 计量计费

使用二级索引功能后，由于索引表的存在，将产生额外的数据存储量。当向主表中写入数据时，在建立索引的过程中将产生一定的读写 CU。本文主要为您介绍二级索引功能的费用组成及计费方式。

说明:
CU（Capacity Unit）指读服务能力单元和写服务能力单元，是数据读写操作的最小计费单位。每秒1次4KB单行读操作为1个预留读 CU。

使用二级索引功能后，由于索引表的存在，将产生额外的数据存储量。当向主表中写入数据时，在建立索引的过程中将产生一定的读写 CU。

索引表相关的计量分三部分：索引表构建时的读写 CU 计量、索引表数据存储量计量、索引表读取数据计量。

计费项	说明
数据存储	主表以及索引表的数据存储费用。
构建索引表的读 CU	为了支持索引行的计算（旧行删除、新行写入、原始行更新）而进行的必要读操作所产生的 CU。
构建索引表的写 CU	向索引表中写入索引行所产生的 CU。
正常读 CU	通过读接口对主表或者索引进行读取产生的 CU。

计费项	说明
正常写 CU	通过写接口对主表进行写入所产生的 CU。

索引的存储，计算及读取的规则为：

- 存储及读取计算规则都与主表相同，详情请参见表格存储计量计费。
- 索引表构建时的计量：
  - 只有产生了有效的索引行才会产生写 CU。
  - 只要可能产生旧索引行的删除、旧索引行的更新、新索引行的写入，均会产生与索引列大小之和相当的读 CU。

索引表构建时的读 CU 计量

主表创建二级索引时，会产生一定量的读 CU，具体规则如下：

- 当通过 PUT 操作写入一行新的数据，并且此行数据以前不存在时（非覆盖写场景）：
  - 如果此表的非主键列上建有索引，但是此行数据不包含索引列（即此行不会生成索引行），则只产生一个读 CU。
  - 如果此行数据所在的列建有索引，并且可以根据此行数据创建有效的索引（即此行会生成索引行），则产生一个读 CU。
- 当通过 PUT 操作写入一行新的数据，并且此行数据以前存在时（覆盖写场景）：
  - 如果此张表的非主键列上建有索引，但此行数据旧值的非主键列不包含索引列，则只产生一个读 CU。
  - 如果此行数据旧值所在的列建有索引，产生的读 CU 为：

    该行旧值中，除主键外，所有与索引表主键相关列的大小之和，按4KB向上取整。如果为0，按1CU计算。
- 当通过 UPDATE 操作更新一行数据时，并且此行数据以前不存在（非更新场景）：
  - 如果此行数据不涉及任何与索引表相关的列，则不产生读 CU。
  - 如果此行数据涉及与索引表相关的列，则产生一个读 CU。
- 当通过 UPDATE 操作更新一行数据时，并且此行数据以前存在（更新场景）：
  - 如果此次更新涉及到的所有属性列与任何索引表都无关，则不产生读 CU。
  - 如果此次更新涉及到的所有属性列中，部分列作为索引表的主键或者属性列，产生的读 CU 为：

    该行旧值中，除主键外，所有与索引表主键相关列的大小之和，按 4KB 向上取整。如果为 0 则按 1CU 计算。

- 当通过 DELETE 操作删除主表中一行数据时，产生的读 CU 为：

  该行除主键外，所有与索引表主键相关列的大小之和按 4 KB 向上取整，如果为 0 则按 1CU 计算。
- 对于主键自增的表，新写入数据时不产生读 CU。修改一行通过主键自增写入的数据时会产生读 CU，计算规则请参见上述 UPDATE 更新操作 。

  说明：

  我们建议通过主键自增功能写入数据，这样可以大大减少由于索引表而产生的读 CU。

  对于非主键自增的表，只要发生了与索引相关列的读取，即使没有读到数据，仍会产生1个读 CU。但对于主键自增的表，新写入数据时不会发生索引相关列的读取，不产生读 CU。

索引表构建时的写 CU 计量

在为写入主表中的数据创建索引时，会产生一定量的写 CU。计量原则如下：

- 如果对主表写入一行数据后，索引表的数据没有发生变化，则不产生写 CU。
- 如果对主表写入一行数据后，索引表增加了一行，则产生与增加的索引行大小相当的写 CU。
- 如果对主表写入一行数据后，索引表删除了一行，则产生与删除的索引行大小相当的写 CU。
- 如果对主表写入一行数据后，索引表更新了一行，则产生与更新的索引行属性列大小相当的写 CU。
- 如果对主表写入一行数据后，索引表删除了一行并又增加了一行，则产生与删除的索引行和增加的索引行之和大小相当的写 CU。

具体细则如下：

- 当通过 PUT 操作写入一行新的数据，并且此行数据以前不存在（非覆盖写场景）：
  - 如果此张表的非主键列上建有索引，但是此行数据不涉及索引列（即此行不会生成索引行），则不产生写 CU。
  - 如果此行数据所在的列建有索引，并且可以根据此行数据创建有效的索引（即此行会生成索引行），则每张索引表产生的写 CU 为：

    如果生成了有效的索引行，索引表写 CU 的个数计量与主表相同，按 4KB 向上取整，否则不产生写 CU。

- 当通过 PUT 操作写入一行新的数据，并且此行数据以前存在时（覆盖写场景）：
  - 如果此张表的非主键列上建有索引，但是此行数据旧值的非主键列不涉及索引列，则不产生写 CU。
  - 如果此行数据旧值的主键或者属性列上建有索引，则每张索引表产生的写 CU 为：

    对于该次 PUT 操作影响到的所有索引（稀疏索引可能不受影响），均计算相应的写 CU。
- 当通过 UPDATE 操作更新一行数据，并且此行数据以前不存在（非更新场景）：
  - 如果此行数据不涉及任何索引表相关的列，则不产生写 CU。
  - 如果此行数据涉及索引表相关的列，则每张索引表产生的写 CU 为：
    - 如果该行能够生成有效的索引行，则以索引行的大小除以 4KB 向上取整进行写 CU 计量。
    - 如果该行不能够生成有效的索引行，则不会产生索引表的写 CU。
- 当通过 UPDATE 操作更新一行数据，并且此行数据以前存在（更新场景）：
  - 如果此次更新涉及到的所有属性列，都与任何索引表无关，则不产生写 CU。
  - 如果此次更新涉及到的所有属性列中，部分列作为索引表的主键或者属性列，则每张索引表产生的写 CU 为：
    - 如果该行的旧值生成了有效的索引行，则按旧的索引行的主键大小，产生索引行的删除 CU。
    - 如果该行的新值生成新的有效的索引行，则按新的索引行的主键大小，产生新的索引行的写入 CU。
    - 如果该行的新值没有生成新的有效的索引行，只是更新了旧的索引行的属性列，则只产生旧的索引行的更新 CU。

    计算规则均按索引行的大小除以 4KB 向上取整。
- 当通过 DELETE 操作删除主表中一行数据时，产生的写 CU 为：

  对每张索引表，如果该行有相应的索引行，则该行中所有与索引表主键相关的列的大小之和，按 4KB 向上取整，否则写 CU 为 0。
- 对于主键自增的表，新写入数据时会产生索引表的写 CU，写 CU 计算规则与上述的通过 PUT 操作写入一行新数据计算规则相同。修改一行通过主键自增写入的数据时，会产生写 CU，计算规则与规则4通过UPDATE 操作更新一行数据计算规则相同。

索引表数据存储量计量

对于索引表，其数据存储量与正常主表没有区别。索引表的数据量是索引表中所有行的数据量之和，所有行的数据量是所有单行数据的主键和属性列数据量之和，详情请参见数据存储量。

索引表读取计量

通过控制台、SDK或者其它途径（如 DLA）进行索引表的读取时，读 CU 计量规则与主表相同，没有区别。

计算举例

我们以包含两张索引表的主表为例，说明在不同写入模式下 CU 的计算。

假设有主表 Table，其中有两列主键 PK0 和 PK1 ，另外有三列预定义列 Col0、Col1、Col2。主表上建有两张索引表 Index0 和 Index1。其中 Index0 的主键为 Col0、PK0、PK1，有一属性列 Col2；Index1 的主键为 Col1、Col0、PK0、PK1，没有属性列。通过UPDATE 接口更新 PK0 、PK1 。

- 该行以前不存在
  - 更新 Col3 列：不产生读写 CU
  - 更新 Col1 列：
    - 产生1个读 CU
    - 不产生写 CU
  - 更新 Col0、 Col1 两列：
    - 产生1个读 CU
    - 对于 Index0，产生与 Col0、PK0、PK1 大小之和相当的写 CU。对于 Index1，产生 Col0、 Col1、PK0、PK1 大小之和相当的写 CU。
- 该行以前存在
  - 更新 Col3 列：不产生读写 CU
  - 更新 Col2 列：
    - 产生旧的 Col0 列大小相当的读 CU，如果 Col0 列以前不存在，则按 1CU 计算。
    - 对于 Index0，如果 Col0 列以前不存在，则不产生写 CU；如果存在，产生Col0、PK0 、PK1、Col2 大小之和相当的写 CU。对于 Index1，则不产生写 CU。
  - 更新 Col1 列：
    - 产生旧的 Col0 列以及 Col1 列大小之和相当的读 CU，如果为 0 则按 1CU 计算。
    - 对于 Index0，不产生写 CU。对于 Index1，会产生旧的 Col0 列、新的 Col1 列、PK0 、PK1大小之和相当的写 CU（写入新的索引行）。如果旧的 Col0 列不存在，则不产生写 CU（没有生成新的索引行）。另外，如果旧的 Col0 列以及旧的 Col1 列都存在，则会产生旧的 Col0、旧的Col1、PK0、PK1大小之和相当的写 CU（删除旧的索引行）

9.7 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 = "BaseStatio
nNumber";

    private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KE
Y_NAME_1, PrimaryKeyType.INTEGER));
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KE
Y_NAME_2, PrimaryKeyType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_CO
L_NAME_1, DefinedColumnType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_CO
L_NAME_2, DefinedColumnType.INTEGER));
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_CO
L_NAME_3, DefinedColumnType.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, indexMetas);

        client.createTable(request);
    }
```

10 Tunnel Service

10.1 概述

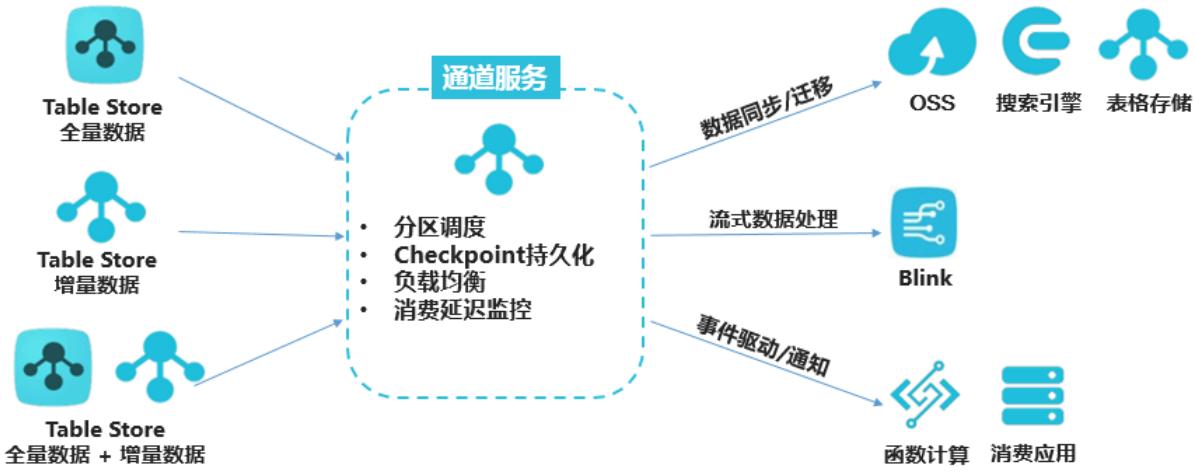
通道服务（Tunnel Service）是基于表格存储数据接口之上的全增量一体化服务。通道服务为您提供了增量、全量、增量加全量三种类型的分布式数据实时消费通道。通过为数据表建立数据通道，您可以简单地实现对表中历史存量和新增数据的消费处理。

背景

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：



功能

通道服务提供了以下功能：

功能	描述
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。

功能	描述
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RPO（恢复点目标，recovery point objective）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，提高水平扩展数据消费速度。

10.2 数据消费框架原理介绍

通道服务是基于表格存储数据接口之上的全增量一体化服务，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。

Tunnel Client 为通道服务的自动化数据消费框架，Tunnel Client 通过每一轮的定时心跳探测（Heartbeat）来进行活跃 Channel 的探测，Channel 和 ChannelConnect 状态的更新，数据处理任务的初始化、运行和结束等。

Tunnel Client 可以解决全量和增量数据处理时的常见问题，例如，如何做负载均衡、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即可。

下文将介绍 Tunnel Client 的自动化数据处理流程、自动化的负载均衡以及自动化的容错处理。更多的细节请参见 Tunnel Client 的源码，Tunnel Client 的源码已经开放在Github上。

自动化数据处理流程

Tunnel Client 通过每一轮的定时心跳探测（Heartbeat）来进行活跃 Channel 的探测，Channel 和 ChannelConnect 状态的更新，数据处理任务的初始化、运行和结束等，这里仅介绍大致的数据处理逻辑，更多的细节可以参阅源码。

1. Tunnel Client 资源的初始化

   a. 将 Tunnel Client 状态由 Ready 置为 Started。

   b. 根据 TunnelWorkerConfig 里的 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。

   c. 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。

   d. 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。）

   e. 初始化TunnelStateMachine（会进行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 里的 heartbeatIntervalInSec 参数决定。

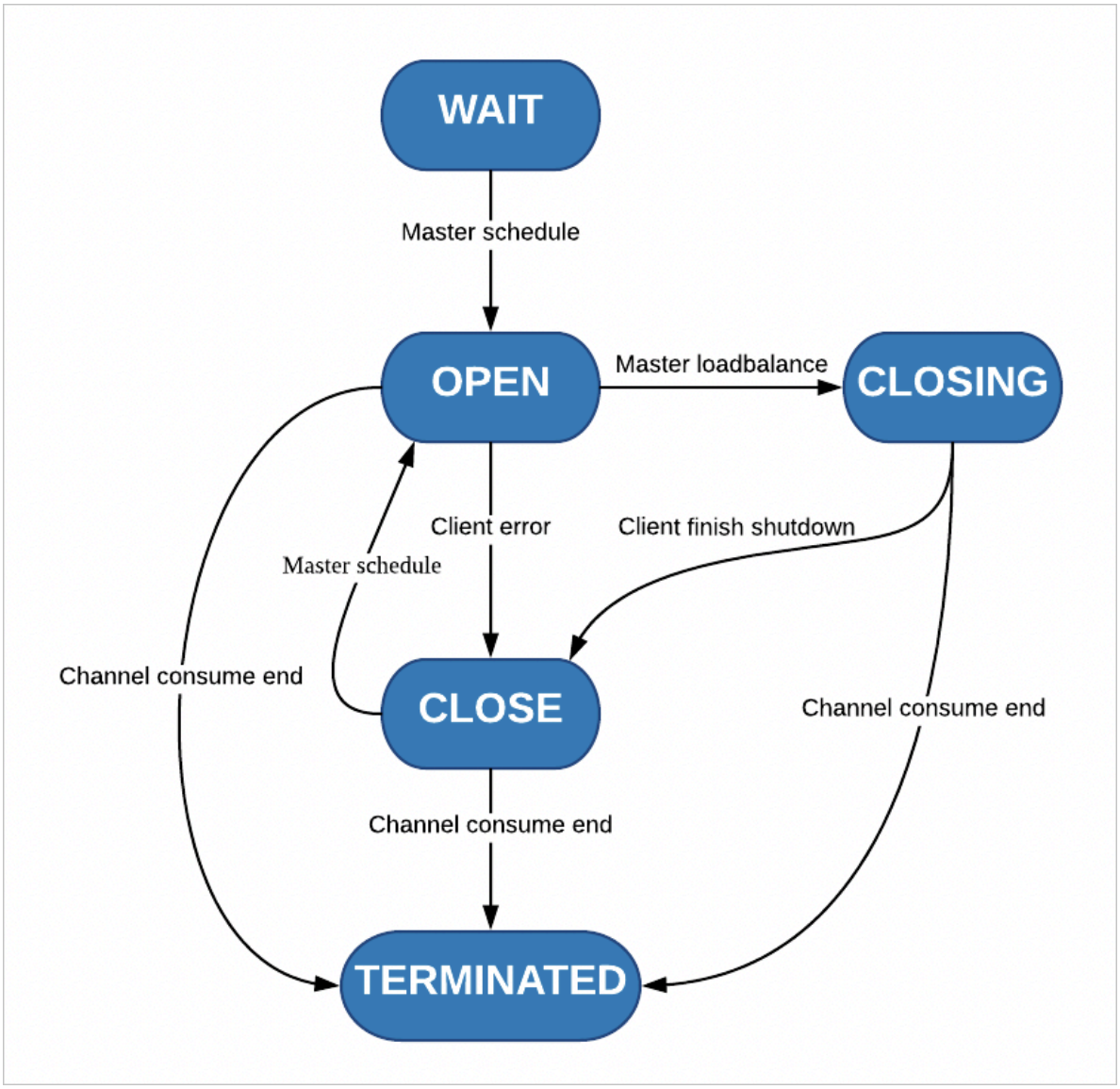
   a. 进行 heartbeat 请求，从 Tunnel 服务端获取最新可用的 Channel 列表，Channel 中会包含有 ChannelId，Channel 的版本和 Channel 的状态信息。

   b. 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，规则大致如下：

      - 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，若未出现的Channel，则直接插入。
      - 新建 ChannelConnect：若此 Channel 未新建有其对应的 ChannelConnect，则会新建一个 WAIT 状态的 ChannelConnect，若对应的 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），处理详细的细节可以参见源码里的ProcessDataPipeline 类。
      - Update 已有 ChannelConnect：Merge 完成后，若 Channel 对应的ChannelConnect 存在，则根据相同 ChannelId 的 Channel 状态来更新ChannelConnect 的状态，比如 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，详细的细节可以参见源码中的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 client 上，以达到负载均衡的目的。Tunnel 服务端通过以下 channel 状态机来驱动每个 channel 的消费以及进行负载均衡。



Tunnel 服务端和 client 通过一个心跳和 channel 版本号更新机制进行状态变换通信。

a. 每个 channel 最初均处于 Wait 状态。
b. 增量类型 channel 需要等待父分区上 channel 消费完毕转为 Terminated 之后才可以转为可消费状态 Open。
c. Open 状态的分区会调度到各个 client上。
d. 在需要负载均衡时，Tunnel 服务端和 client 有一个 channel 状态 Open->Closing->Closed 的调度协议，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 进行消费时（TunnelId 相同），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，用户可以很容易的通过增加 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： 当客户端（Tunnel Client）没有被正常 shutdown 时（比如异常退出或者手动结束），我们会自动帮用户进行资源的回收，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。
- 容错处理： 当客户端出现 Heartbeat 超时等非参数类错误时，表格存储会自动帮用户 Renew Connect，以保证数据消费可以稳定的进行持续同步。

10.3 快速入门

您可以在表格存储管理控制台快速体验通道服务（Tunnel Service）功能。

前提条件

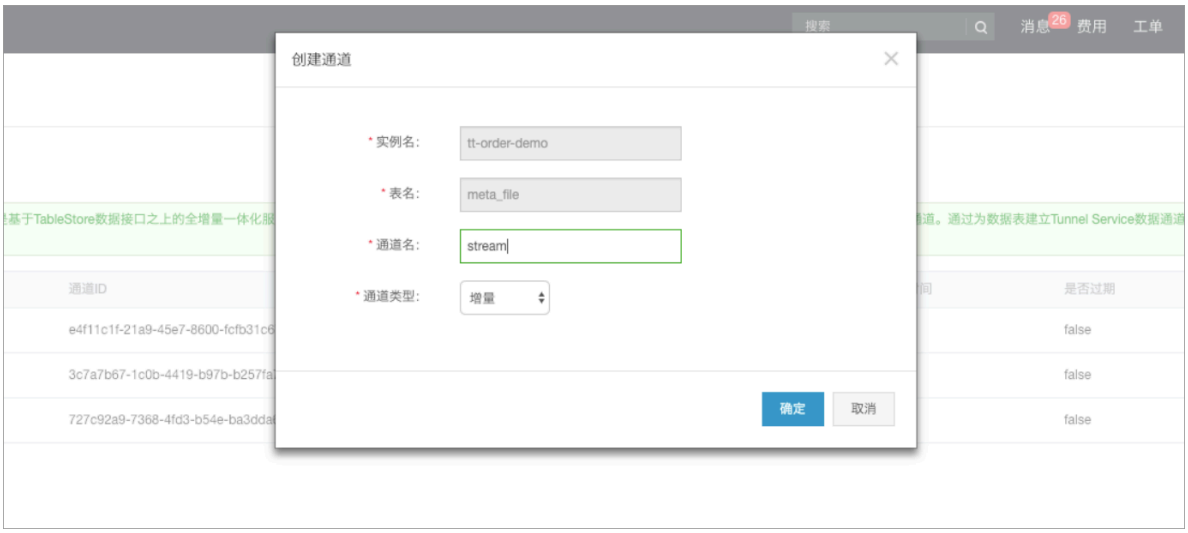
您已经开通表格存储服务。

创建数据通道

1. 登录表格存储控制台。
2. 进找到目标数据表，然后在该数据表的右侧单击通道管理。
3. 在管理通道页面的右上角，单击创建通道。

4. 在弹出的创建通道窗口，输入通道名称，并选择通道类型。

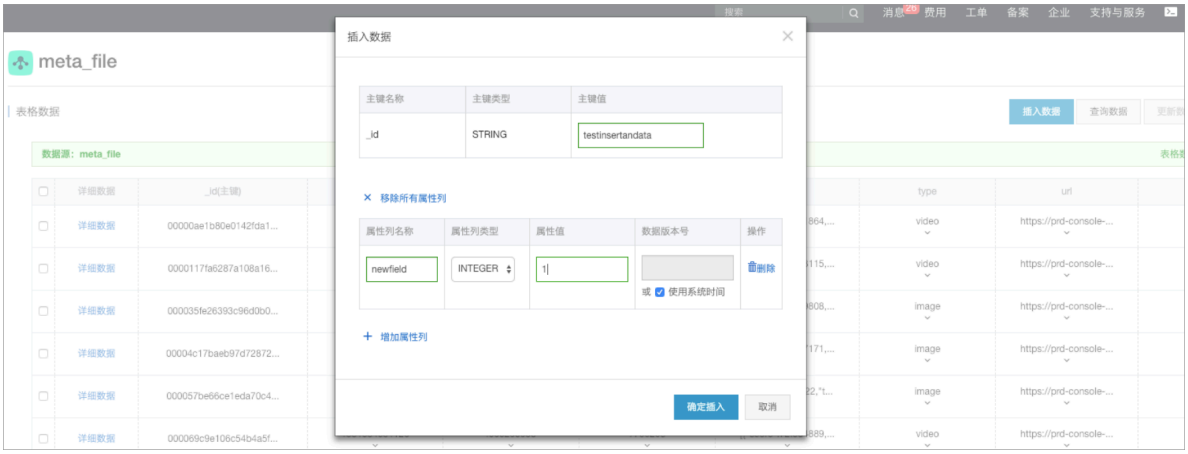
   通道服务为您提供了增量、全量、增量加全量三种类型的分布式数据实时消费通道。本文档中以增量类型为例。



   创建成功后，您可以在管理通道页面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。

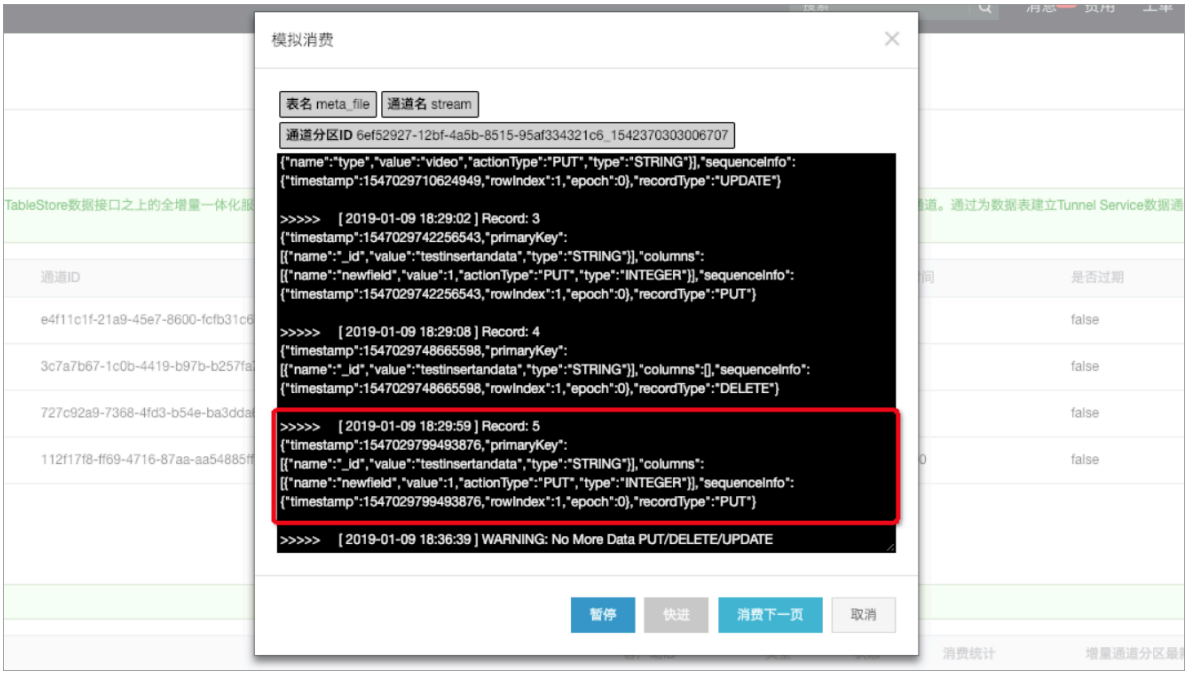
预览通道中的数据格式

1. 在控制台数据管理页面随机写入或删除数据。



2. 返回管理通道页面，找到新创建的通道，在其右侧单击展示通道分区列表，页面底部会显示该通道的通道分区列表。

3. 在通道分区的右侧单击模拟消费。在弹出的页面中可以预览通道中的数据格式。



开启通道的数据消费

1. 在通道列表中复制通道ID。

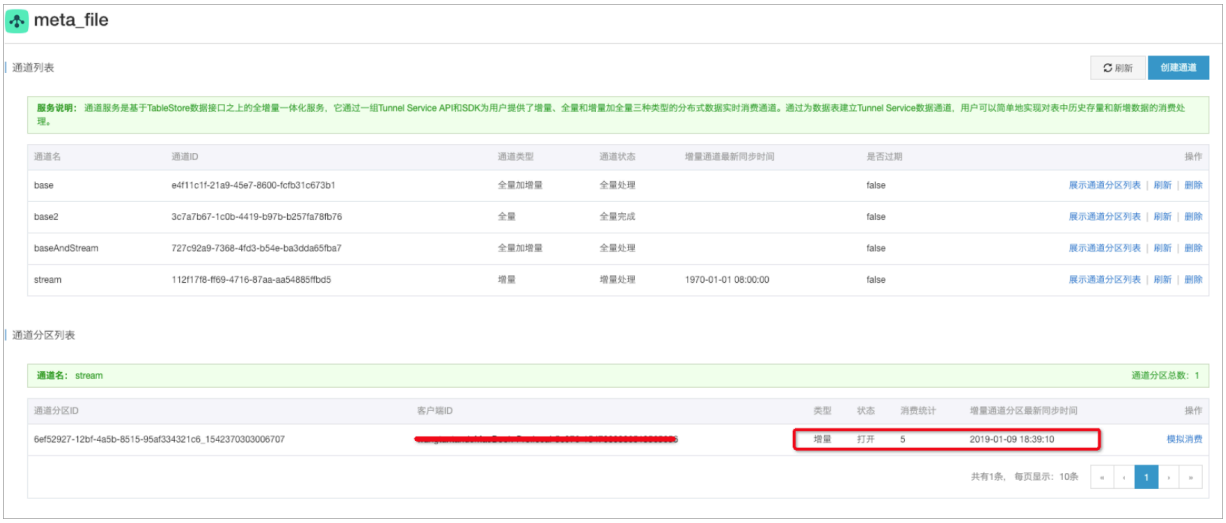
2. 使用任一语言的通道SDK，开启通道的数据消费。

```
// 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 {
            // Mock Record Process.
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void shutdown() {
        System.out.println("Mock shutdown");
    }
}

// TunnelWorkerConfig里面还有更多的高级参数，这里不做展开，会有专门的文档介
绍。
TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProc
essor());
// 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

在数据消费标准输出可以看到增量数据消费日志，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。



10.4 SDK

您可以使用以下SDK实现通道服务（Tunnel Service）。

- **Go SDK**
- **Java SDK**

10.5 增量同步性能白皮书

本文主要介绍 Tunnel 增量性能的测试，主要包括测试环境、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**
  - **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
  - 型号：共享通用型 (mn4) ecs.mn4.4xlarge
  - 配置：
    - CPU：16核
    - 内存：64GB
    - 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.currentTimeMillis() - 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。

- 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

  1. 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）。
  2. 创建增量通道。
  3. 使用压力器进行增量数据的写入。
  4. 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）。
  5. 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	lgFPVmWbN	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	vnLiJFDdQ	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能力，是从实际使用角度出发进行的性能测试。

测试结果

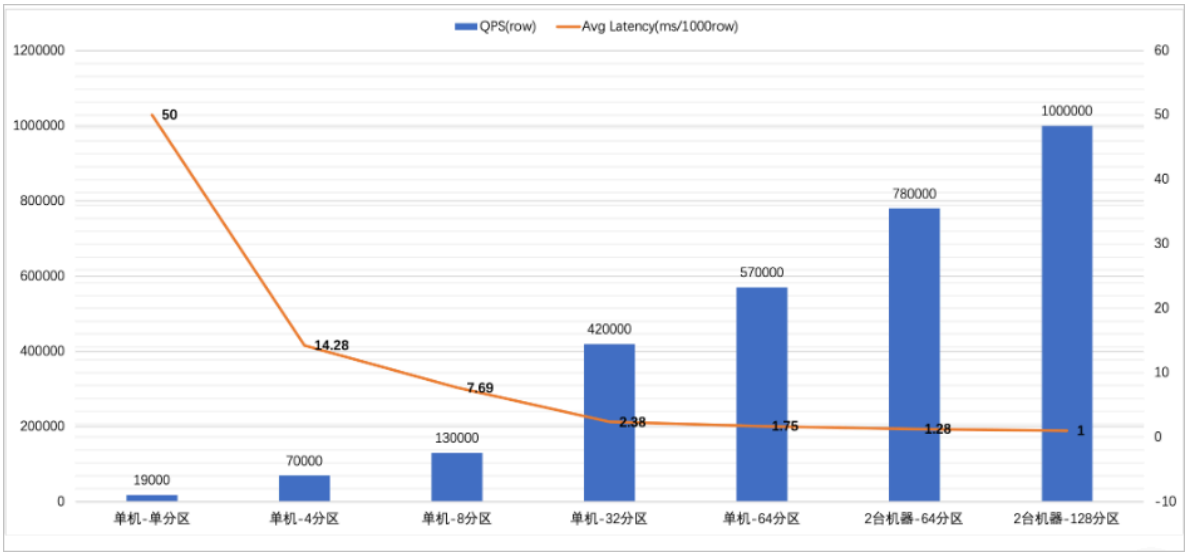
该部分主要概述各个场景下的指标测试结果，测试的细节可以参见测试细则部分。

· 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57W的QPS。两台机器对64分区进行同步后，平均QPS成功达到了78W行左右，约等于单

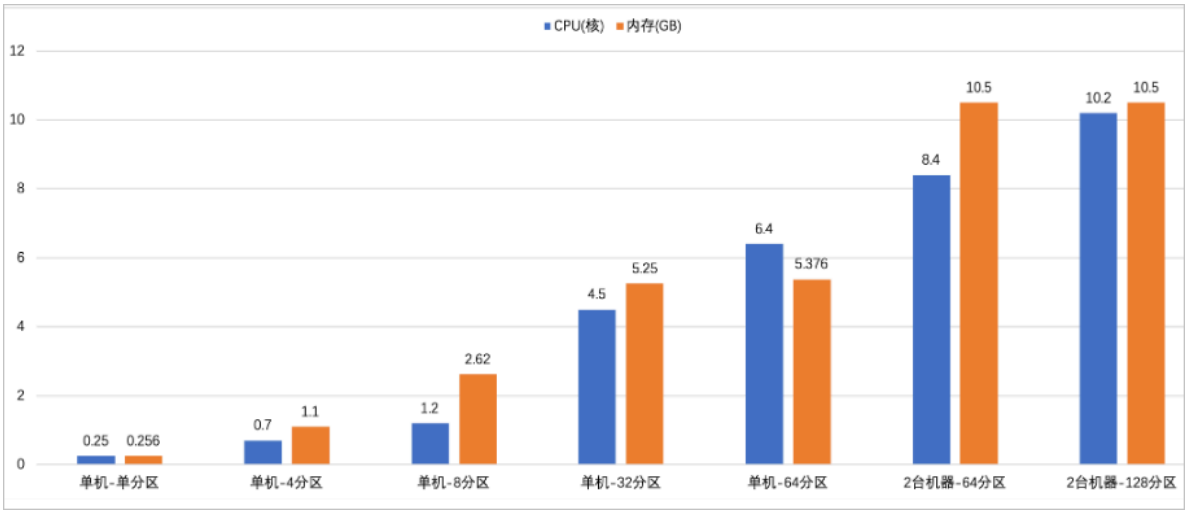
机-32分区场景下（42W）的两倍速率。而在最后的两台机器-128分区场景下，Tunnel增量同步的QPS也成功达到了100W行。



- 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。

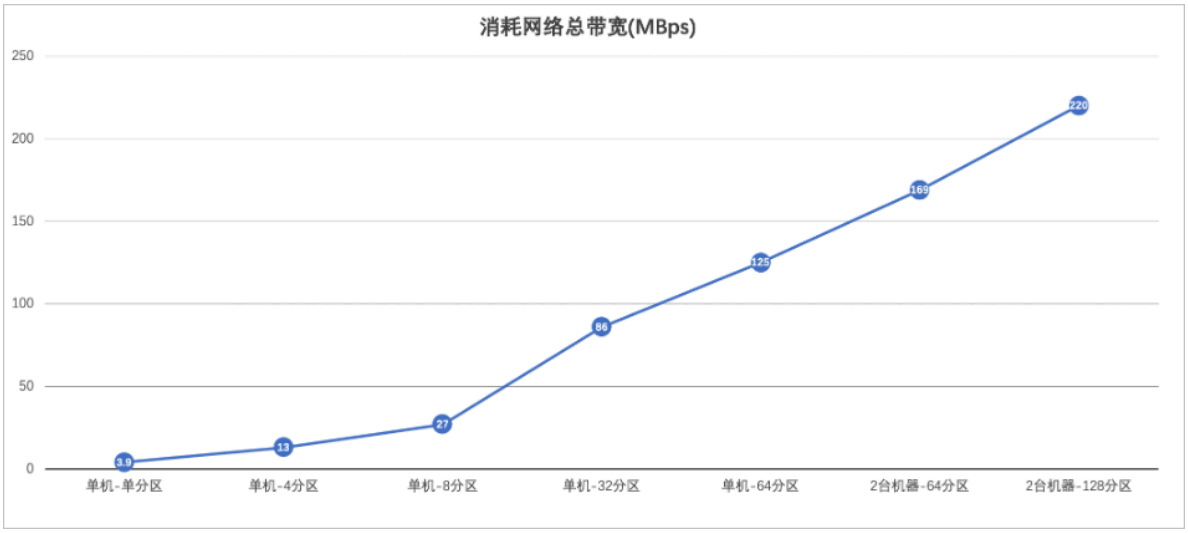
在单机-单分区场景下，消耗的 CPU 为0.25个单核 CPU。2台机器-128个分区的场景下，当同步 QPS 达到100W行时，消耗的 CPU 也仅为10.2个单核 CPU。从内存消耗方面看。在分区数较少时，CPU 和分区数呈线性关系，而在分区数增加较多（32个和64个）时，单机内存消耗基本维持在5.3GB左右。



- 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125MBps，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169MBps，和单机-32分区的86MBps的两倍近乎相等。而在两台机器-128分区的100W QPS场景中，总吞吐量也达到了220MBps。

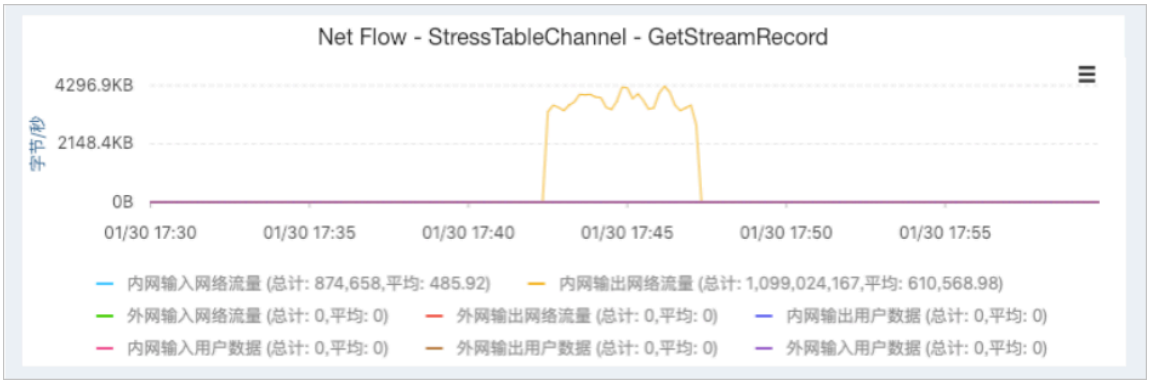


测试细则

- 单机单 Channel：1.9W 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0.256GB左右（测试机器内存为64GB）
  - 网络带宽消耗：4000KB/s左右

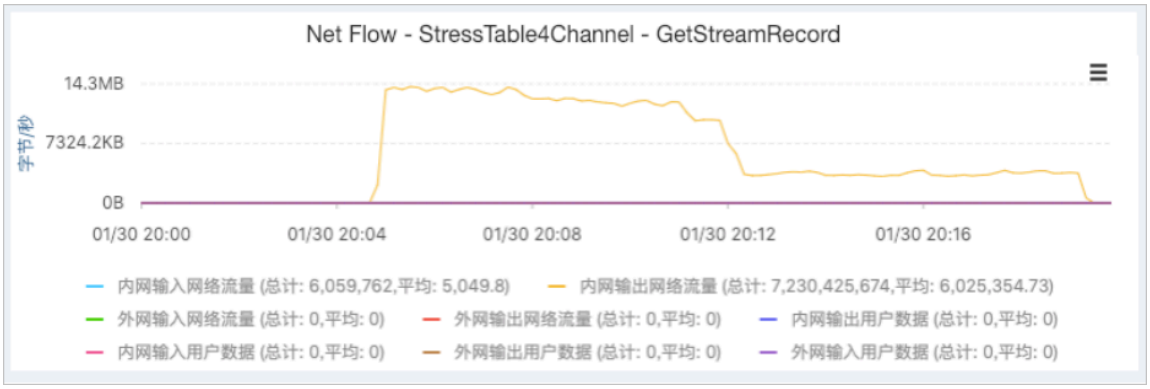


- 单机4分区：7W QPS

- 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1.1GB左右。（测试机器内存为64GB）
- 网络带宽消耗：13MBps 左右


Net Flow - StressTable4Channel - GetStreamRecord
14.3MB
7324.2KB
0B
字节/秒
01/30 20:00
01/30 20:04
01/30 20:08
01/30 20:12
01/30 20:16
内网输入网络流量 (总计: 6,059,762,平均: 5,049.8)
内网输出网络流量 (总计: 7,230,425,674,平均: 6,025,354.73)
外网输入网络流量 (总计: 0,平均: 0)
外网输出网络流量 (总计: 0,平均: 0)
内网输出用户数据 (总计: 0,平均: 0)
内网输入用户数据 (总计: 0,平均: 0)
外网输出用户数据 (总计: 0,平均: 0)
外网输入用户数据 (总计: 0,平均: 0)

- 单机8分区：13W QPS

通道分区列表

通道名：teststream2 通道分区总数：8

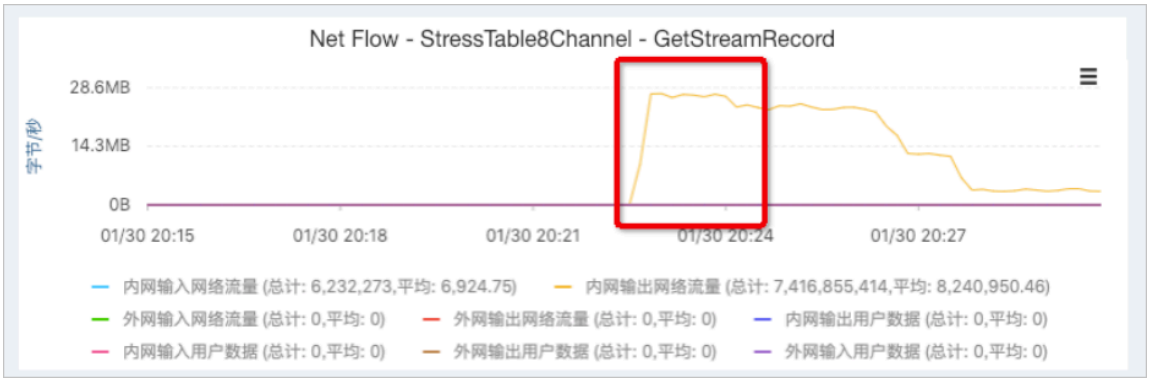
通道分区ID	客户端ID	类型	状态	消费统计	增量通道分区最新同步时间	操作
182bfcd3-c4e1-46d2-a9b8-0fa5dd3c3317_1548850307298607		增量	等待	0	1970-01-01 08:00:00	模拟消费
43b553d7-16ee-45fa-83a0-2591881bb429_1548850307298607		增量	等待	0	1970-01-01 08:00:00	模拟消费
54fffd56-cc51-41a4-bd11-24e6483cf6d0_1548850307298607		增量	等待	0	1970-01-01 08:00:00	模拟消费
6c051a78-e187-4f12-96e4-d581389f8212_1548850307298607		增量	等待	0	1970-01-01 08:00:00	模拟消费
905c6503-255b-43bd-aec1-42a7cb379237_1548850307298607		增量	等待	0	1970-01-01 08:00:00	模拟消费
b69601c6-67c3-446e-808a-8416ee921f0f_1548850307298607		增量	等待	0	1970-01-01 08:00:00	模拟消费
c1c4d3ca-60f2-47b3-a883-d69774ca41db_1548850307298607		增量	等待	0	1970-01-01 08:00:00	模拟消费
f47cc574-a459-4669-a833-867ed7b07c87_1548850307298607		增量	等待	0	1970-01-01 08:00:00	模拟消费

共有8条，每页显示：10条 « ‹ 1 › »

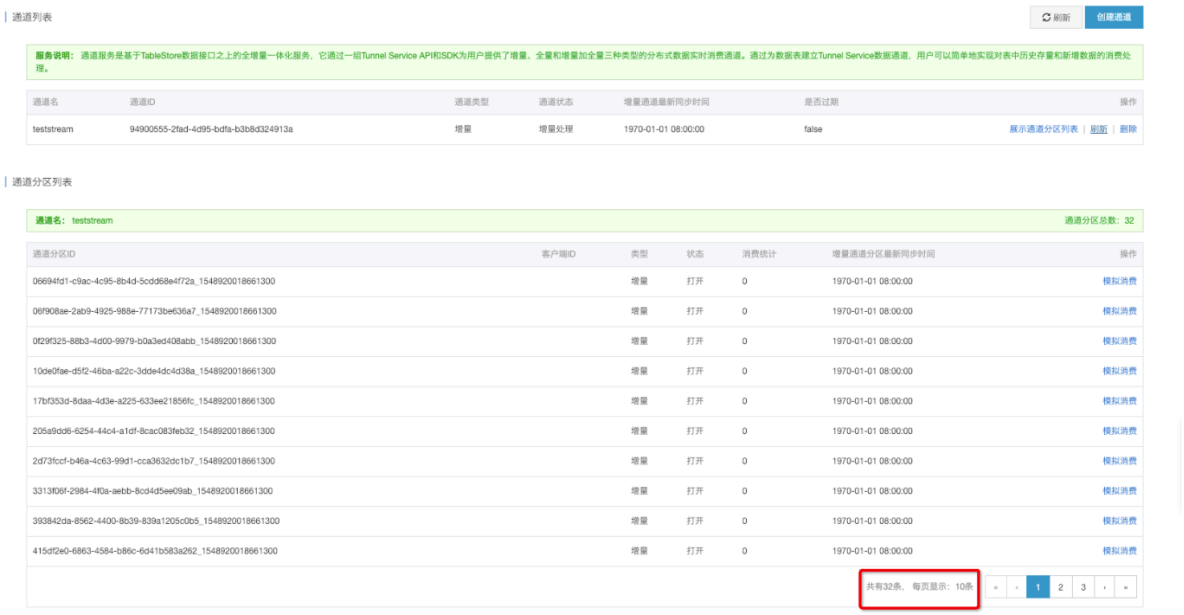
  - 测试时间：2019/1/30 20:20
  - QPS：稳定速率130000行/秒，峰值速率141644行/秒
  - Latency：7.69ms/1000行左右


```
{"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2.62GB左右（测试机器内存为64GB）
  - 消耗网络带宽：27MBps 左右


Net Flow - StressTable8Channel - GetStreamRecord
28.6MB
14.3MB
0B
字节/秒
01/30 20:15
01/30 20:18
01/30 20:21
01/30 20:24
01/30 20:27
内网输入网络流量 (总计: 6,232,273,平均: 6,924.75)
内网输出网络流量 (总计: 7,416,855,414,平均: 8,240,950.46)
外网输入网络流量 (总计: 0,平均: 0)
外网输出网络流量 (总计: 0,平均: 0)
内网输出用户数据 (总计: 0,平均: 0)
内网输入用户数据 (总计: 0,平均: 0)
外网输出用户数据 (总计: 0,平均: 0)
外网输入用户数据 (总计: 0,平均: 0)

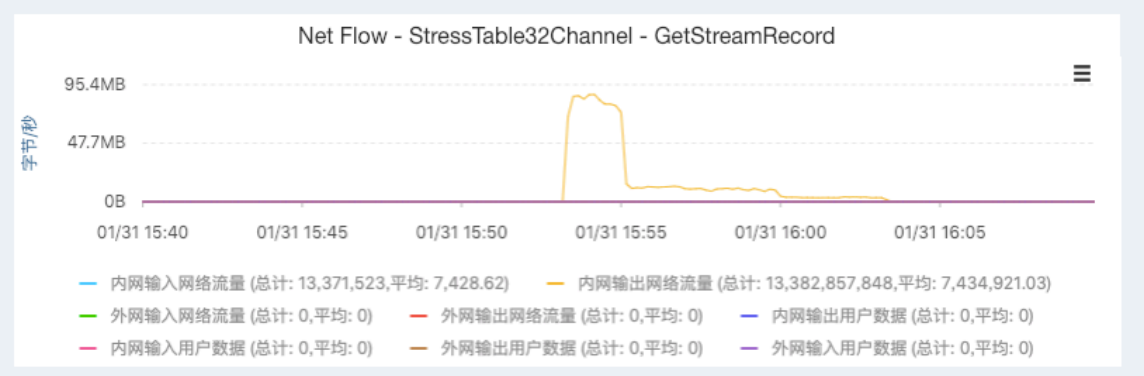
- 单机32分区：42W QPS



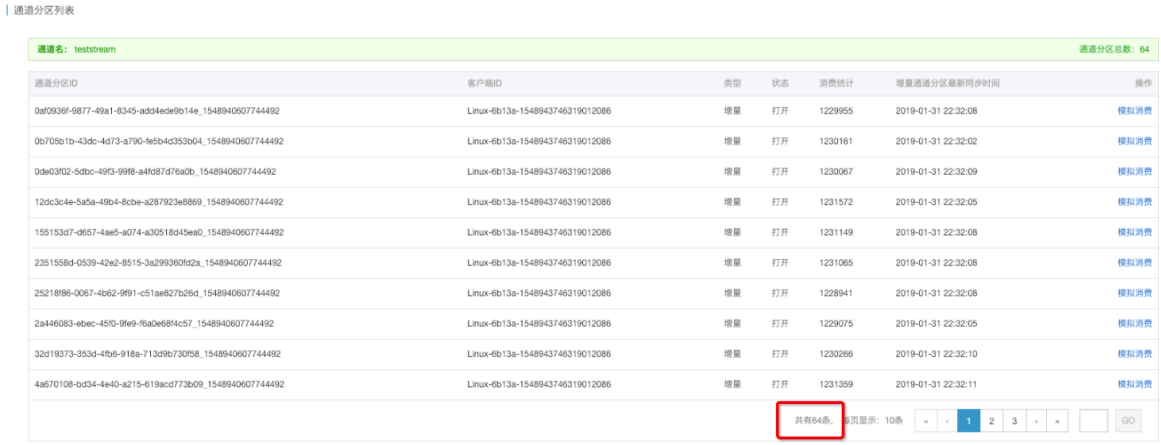
- 测试时间：2019/1/31 15:50
- QPS：稳定速率42W/s， 峰值速率447600行/s
- Latency：2.38ms/1000行


```
{"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5.25GB 左右（物理内存 64GB）
- 增量数据消耗网络带宽：86MBps 左右


Net Flow - StressTable32Channel - GetStreamRecord
95.4MB
47.7MB
0B
字节/秒
01/31 15:40
01/31 15:45
01/31 15:50
01/31 15:55
01/31 16:00
01/31 16:05
内网输入网络流量 (总计: 13,371,523,平均: 7,428.62)
内网输出网络流量 (总计: 13,382,857,848,平均: 7,434,921.03)
外网输入网络流量 (总计: 0,平均: 0)
外网输出网络流量 (总计: 0,平均: 0)
内网输出用户数据 (总计: 0,平均: 0)
内网输入用户数据 (总计: 0,平均: 0)
外网输出用户数据 (总计: 0,平均: 0)
外网输入用户数据 (总计: 0,平均: 0)

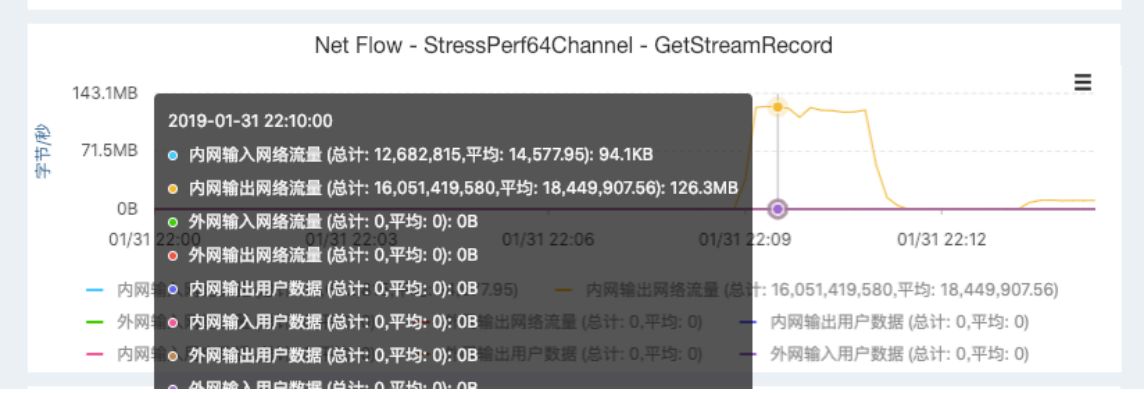
· 单机64分区（千兆网卡被打满）： 57W QPS



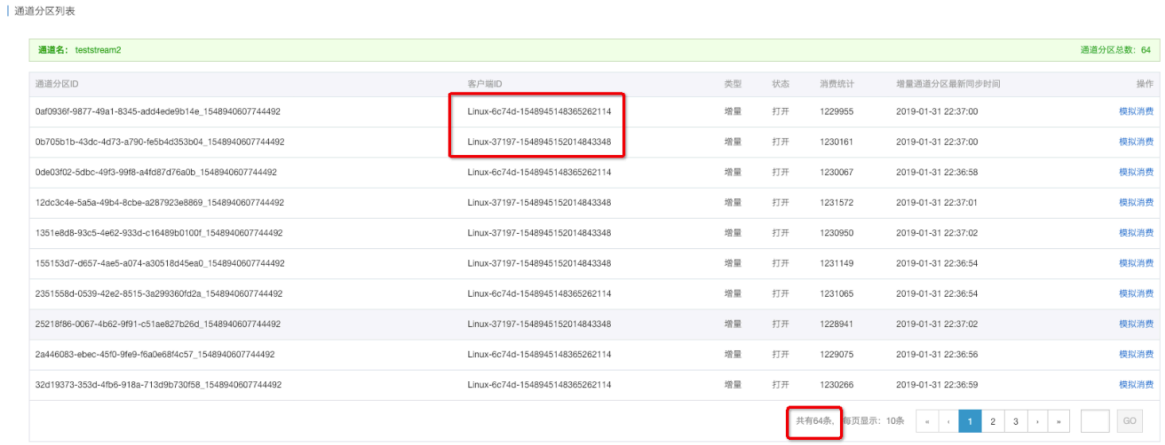
- 测试时间：2019/1/31 22:10
- QPS：稳定速率57W行/s左右，峰值速率581400行/s
- Latency：1.75ms/1000行左右


```
{"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5.376GB左右
- 增量数据消耗网络带宽：125MBps 左右（达到千兆网卡的速率极限）


Net Flow - StressPerf64Channel - GetStreamRecord
143.1MB
71.5MB
0B
字节/秒
01/31 22:00
01/31 22:03
01/31 22:06
01/31 22:09
01/31 22:12
2019-01-31 22:10:00
内网输入网络流量 (总计: 12,682,815,平均: 14,577.95): 94.1KB
内网输出网络流量 (总计: 16,051,419,580,平均: 18,449,907.56): 126.3MB
外网输入网络流量 (总计: 0,平均: 0): 0B
外网输出网络流量 (总计: 0,平均: 0): 0B
内网输出用户数据 (总计: 0,平均: 0): 0B
内网输入用户数据 (总计: 0,平均: 0): 0B
外网输出用户数据 (总计: 0,平均: 0): 0B
内网输出网络流量 (总计: 16,051,419,580,平均: 18,449,907.56)
内网输出用户数据 (总计: 0,平均: 0)
外网输入用户数据 (总计: 0,平均: 0)

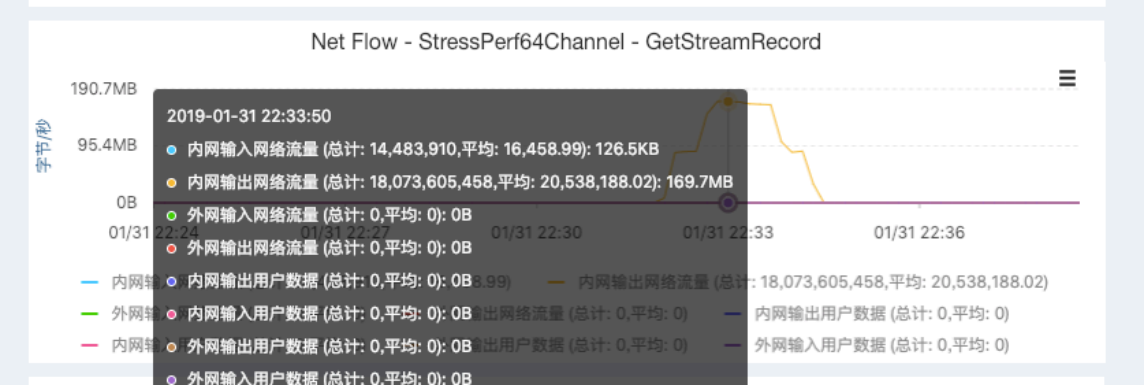
- 2台机器共同消费64分区：78W QPS



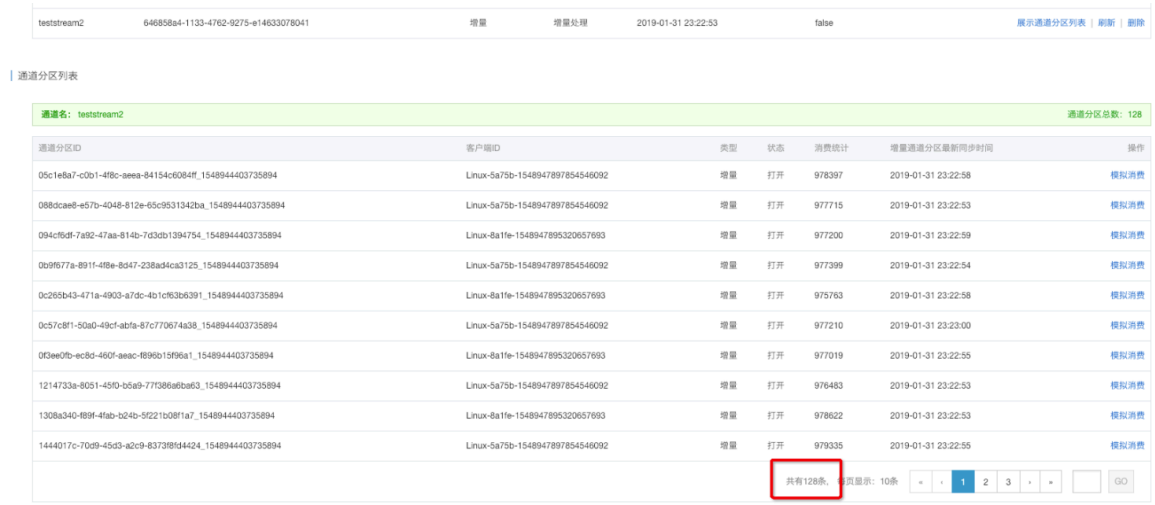
- 测试时间：2018/1/31 22:30
- QPS：每台稳定速率在39W行/s左右，总的稳定速率在78W行/s左右
- Latency：1.28ms/1000行

```
{"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%（10.5GB）
- 增量数据消耗总网络带宽：169MBps 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）


Net Flow - StressPerf64Channel - GetStreamRecord
190.7MB
95.4MB
0B
字节/秒
01/31 22:24
01/31 22:27
01/31 22:30
01/31 22:33
01/31 22:36
2019-01-31 22:33:50
内网输入网络流量 (总计: 14,483,910,平均: 16,458.99): 126.5KB
内网输出网络流量 (总计: 18,073,605,458,平均: 20,538,188.02): 169.7MB
外网输入网络流量 (总计: 0,平均: 0): 0B
外网输出网络流量 (总计: 0,平均: 0): 0B
内网输出用户数据 (总计: 0,平均: 0): 0B
内网输入用户数据 (总计: 0,平均: 0): 0B
外网输出用户数据 (总计: 0,平均: 0): 0B
外网输入用户数据 (总计: 0,平均: 0): 0B
内网输出网络流量 (总计: 18,073,605,458,平均: 20,538,188.02)
内网输出用户数据 (总计: 0,平均: 0)
外网输入用户数据 (总计: 0,平均: 0)

· 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）：100W QPS



- 测试时间：2018/1/31 23:20
- QPS：每台稳定速率在50W行/s左右，总的稳定速率在100W行/s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%（10.5GB）
- 增量数据消耗总网络带宽：220MBps 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 HBase 支持

11.1 Table Store HBase Client

除了使用现有的 SDK 以及 Restful API 来访问表格存储，我们还提供了 Table Store HBase Client。使用开源 HBase API 的 JAVA 应用可以通过 Table Store HBase Client 来直接访问表格存储服务。

Table Store 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。

Table Store 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>
```

由于表格存储是一个全托管的 NoSQL 数据库服务，当使用 Table Store HBase Client 之后，您不再需要关心 HBase Server 的相关事项，只需要通过 Client 暴露出来的接口进行表或者数据的操作即可。

相比自行搭建 HBase 服务，表格存储有着如下的优势：

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及 VPC、主/子账号功能，授权粒度达到表级别和 API 级别。	需要额外的安全机制。

对比项	表格存储	自建HBase集群
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。

11.2 Table Store HBase Client 支持的功能

本文主要为您介绍Table Store 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，Table Store 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

功能	支持情况
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
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
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

功能	支持情况
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数据分区，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。

11.3 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。

11.4 从 HBase 迁移到表格存储

Table Store 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 迁移到表格存储及注意事项。

依赖

Table Store HBase Client 1.2.0 版本中依赖了 HBase Client 1.2.0 版本和 Table Store 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 Table Store JAVA SDK，可以使用 exclusion 标签。下面示例中使用 HBase Client 1.2.1 版本和 Table Store JAVA SDK 4.2.0 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 Table Store 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 Table Store 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 Table Store 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，需要配置秘钥等信息。

  - **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。

11.5 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();
```

11.6 Hello World

本节描述如何使用 Table Store HBase Client 实现一个简单的 Hello World 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，目录位置是 *src/test/java/samples/HelloWorld.java*。

操作步骤

- 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>
  ```

  更多项目依赖配置方法，参见从 HBase 迁移到表格存储。

- 配置文件

  hbase-site.xml 中增加下列配置项：

  ```
  <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>
```

更高级的配置请参考从 HBase 迁移到表格存储。

- 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();
```

- 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
 默认值是 1，TimeToLive 默认值是 Integer.INF_MAX
        descriptor.addFamily(new HColumnDescriptor(COLUMN_FAM
ILY_NAME));

        // 通过 Admin 的 createTable 接口创建表
        System.out.println("Create table " + descriptor.getNameAsS
tring());
        admin.createTable(descriptor);
```

- 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，主键是 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，如果没有配置则默认是“f”，所以写入数据时 COLUMN_FAMILY_NAME 可以是空值
```

```
        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);
```

- 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_FAM
ILY_NAME, COLUMN_NAME));
        System.out.println("Get one row by row key");
        System.out.printf("\t%s = %s\n", Bytes.toString(ROW_KEY),
value);
```

- 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));
    }
```

- 删表

  使用 Admin API 删除一张表。

```
        print("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("HelloTable
store");
    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_FAM
ILY_NAME));
            System.out.println("Create table " + descriptor.getNameAsS
tring());
            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_VAL
UE);
            table.put(put);
            Result getResult = table.get(new Get(ROW_KEY));
            String value = Bytes.toString(getResult.getValue(
COLUMN_FAMILY_NAME, COLUMN_NAME));
            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 HelloTable
store: " + e.toString());
            System.exit(1);
        }
    }
}
```