阿里云
表格存储

产品功能

文档版本：20181212

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。
斜体	表示参数、变量。	`bae log list --instanceid` *`Instance_ID`*
[]或者[a|b]	表示可选项，至多选择一个。	`ipconfig` *`[-all|-t]`*
{}或者{a|b}	表示必选项，至多选择一个。	`swich` *`{stand | slave}`*

目录

1 表格存储的表

创建表格存储的表时，需要指定表名、主键和预留读/写吞吐量。在传统数据库中，表具有预定义的结构信息，比如表的名称、主键、列名和类型，表中的所有行都具有相同的列集合。表格存储是NoSQL 数据库，除了主键需要格式外，没有其他的格式信息。本章将介绍表的概念和使用。

表名

表格存储的表的名称必须符合以下规范：

- 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。
- 长度在 1~255 字符之间。
- 同一个实例下不能有同名的表，但不同实例内的表名称可以相同。

主键

创建表格存储的表时必须指定表的主键。主键包含 1~4 个主键列，每个主键列都有名字和类型。表格存储对主键列的名字和类型都有限制，详细信息请参考主键和属性。

表格存储根据表的主键索引数据，表中的行按照它们的主键进行升序排序。

配置预留读/写吞吐量

为确保应用程序获得稳定、低延时的服务，应用程序可以在创建表的时候指定预留读/写吞吐量。如果预留读/写吞吐量不为 0，表格存储根据预留读/写吞吐量来为表分配相应的资源，满足应用程序的预留吞吐量需求，同时根据配置的预留读/写吞吐量收取相应费用。应用程序可以根据自身的业务需求动态上调和下调表的预留读/写吞吐量。预留读/写吞吐量通过读服务能力单元和写服务能力单元这两个数值来设置。

说明：

容量型实例下的表不支持预留读/写吞吐量。

通过 UpdateTable 操作可以更新表的预留读/写吞吐量。预留读/写吞吐量的更新有如下规则：

- 一张表上的两次更新的间隔必须大于 2 分钟。例如，12:43 AM 更新了某个表的预留读/写吞吐量，那么只有在 12:45 AM 之后才能再次更新该表的预留读/写吞吐量。更新间隔必须大于 2 分

钟的限制是针对表的，在 12:43 AM ~ 12:45 AM 之间，用户可以更新其他表的预留读/写吞吐量。

- 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，但是两次调整之间的时间间隔必须大于 2 分钟。调整预留读/写吞吐量的定义是，只要读服务能力单元或写服务能力单元配置其中一项进行了更新，则此次操作被视为对表进行了更新。
- 预留读/写吞吐量调整完毕后 1 分钟内生效。

对于访问消耗的读/写吞吐量中，超出预留读/写吞吐量的部分，会计入按量读/写吞吐量，并根据按量读/写吞吐量单价进行计费。

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，在用户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。

分区键下的数据量限制

表格存储按照分区键的范围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。为了防止分区过大无法切分，建议单个分区键值下的所有行数据大小之和不要超过 10 GB。

建表后加载时间

表格存储的表在被创建之后需要 1 分钟进行加载，在此期间对该表的读/写数据操作均会失败。应用程序应该等待表加载完毕后再进行数据操作。

最佳实践

表格存储表的最佳实践

使用表格存储的 SDK 进行表操作

JAVA-SDK -- 表操作

Python-SDK -- 表操作

2 表格存储的数据操作

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是行，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，则这一行旧的数据会先被删除，再新写入一行。
  - UpdateRow：更新一行的数据。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 单位写服务能力单元和 1 单位读服务能力单元。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 单位写服务能力单元和 1 单位读服务能力单元。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 单位写服务能力单元。请参见 *DeleteRow* 详解。

写操作会根据指定的 condition 情况消耗一定的读服务能力单元。

示例

下面举例说明单行写操作的写服务能力单元和读服务能力单元的计算。

示例 1，使用 PutRow 进行如下行写入操作：

```
//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/写服务能力单元的消耗情况如下：

- 将 condition 设置为 EXPECT_EXIST 时：消耗的写服务能力单元为 4322 Byte 除以 4 KB 向上取整，消耗的读服务能力单元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 个单位的写服务能力单元和 1 个单位的读服务能力单元。
- 将 condition 设置为 IGNORE 时：消耗的写服务能力单元为 4322 Byte 除以 4 KB 向上取整，消耗 0 个读服务能力单元。该 PutRow 操作消耗 2 个单位的写服务能力单元和 0 个单位的读服务能力单元。
- 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 个单位的写服务能力单元和 1 个单位的读服务能力单元。

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
```

读/写服务能力单元的消耗情况如下：

- 将 condition 设置为 IGNORE 时：消耗的写服务能力单元为 922 Byte 除以 4 KB 向上取整，消耗 0 个读服务能力单元。该 UpdateRow 操作消耗 1个单位的写服务能力单元和 0 个读服务能力单元。
- 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 个单位的写服务能力单元和 1 个单位的读服务能力单元。

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)}
}
```

读/写服务能力单元的消耗情况如下：

- 将 condition 设置为 EXPECT_EXIST 时：消耗的写服务能力单元为 4322 Byte 除以 4 KB 向上取整，消耗的读服务能力单元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 个单位的写服务能力单元和 1 个单位的读服务能力单元。
- 将 condition 设置为 IGNORE 时：消耗的写服务能力单元为 4322 Byte 除以 4 KB 向上取整，消耗 0 个读服务能力单元，该 UpdateRow 操作消耗 2 个单位的写服务能力单元和 0 个单位的读服务能力单元。

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 1 个单位服务能力单元。因此，该 DeleteRow 操作消耗 1 个单位的写服务能力单元。

读/写服务能力单元的消耗情况如下：

- 将 condition 设置为 EXPECT_EXIST 时：消耗的写服务能力单元为该行主键数据大小 10 Byte 除以 4 KB 向上取整，消耗的读服务能力单元为该主键数据大小 10 Byte 除以 4 KB 向上取整。该 DeleteRow 操作消耗 1 个单位的写服务能力单元和 1 个单位的读服务能力单元。
- 将 condition 设置为 IGNORE 时：消耗的写服务能力单元为该行主键数据大小 10 Byte 除以 4 KB 向上取整，消耗 0 个读服务能力单元。该 DeleteRow 操作消耗 1 个单位的写服务能力单元和 0 个单位的读服务能力单元。

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计算读服务能力单元。将行数据大小除以 4 KB 向上取整作为本次读取操作消耗的读服务能力单元。如果操作指定的行不存在，则消耗 1 单位读服务能力单元，单行读取操作不会消耗写服务能力单元。

示例

使用 GetRow 读取一行消耗的写服务能力单元的计算：

```
//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'}
}
```

消耗的读服务能力单元为 1216 Byte 除以 4 KB 向上取整，该 GetRow 操作消耗 1 个单位的读服务能力单元。

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，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，应用程序也必须要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，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，应用程序也必须要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。

表格存储计算读取，区间起始点到下一条未读数据的起始点的，所有行主键数据大小与实际读取的属性列数据大小之和。将数据大小之和除以 4 KB 向上取整计算消耗的读服务能力单元。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占用数据大小为 330 Byte，则消耗的读服务能力单元为 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 操作消耗的读服务能力单元计算：

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 个单位的读服务能力单元。

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

3 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。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，可用于 *PutRow*、*UpdateRow*、*DeleteRow* 和 *BatchWriteRow* 中。

列判断条件包括行存在性条件和列条件：

- 行存在性条件分为 `IGNORE`、`EXPECT_EXIS` 和 `EXPECT_NOT_EXIST`，分别代表忽略、期望存在和期望不存在。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的操作示例如下：

```
// 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 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

表格存储的主键列自增主要存在以下限制。

- 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。

- 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。

5 使用过滤器

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 Filter 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 Filter 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 SingleColumnValueFilter 和 CompositeColumnValueFilter，这两个 Filter 都是基于参考列的列值决定是否过滤某行。

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

API 文档参考：*CompositeColumnValueFilter* 和 *SingleColumnValueFilter*。

- 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);
```

Filter 是对读取后的结果再进行一次过滤，所以 SingleColumnValueFilter 或者 CompositeC olumnValueFilter 中的参考列必须在读取的结果内。如果用户指定了要读取的列，但其中不包含参考列，则 Filter 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，即用户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 Filter 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 200 KB 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 20 KB，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 4 MB 的限制，当在该次返回的 5000 行或者 4 MB 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。

6 原子计数器

原子计数器是表格存储提供的一个新特性，即将列当成一个原子计数器来使用。这样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。

当要给列值+1或其他增量值时，在传统设计中（即没有原子计数器情况下），您需要执行Read-Modify-Write(RMW)操作。首先您需要从服务端读取列的旧值，然后在客户端完成列值的修改，最后将修改后的列值写回服务端。此时，在并发多客户端修改同一行时会引发一致性的问题。

针对一致性问题，目前的解决方案是：通过一个行级事务，在执行RMW前，首先通过启动该行的一个行事务获得行锁，然后在这个事务内完成RMW的操作。通过该方案可以实现单行多客户端修改的强一致性，但是原子计数器写入性能将会受到较大影响。RMW实际对应两轮网络请求操作，因此会导致比较大的性能开销。

为了解决由强一致性导致的写入性能开销的问题，我们在表格存储中引入了原子计数器。一个RMW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meta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。

7 局部事务

局部事务是表格存储提供的一个新特性，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。

在未引入局部事务功能前，表格存储保证单行的写入是原子的，但未保证单行多次读写和多行读写的原子性，从而限制了用户的使用场景。

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，大大扩展了使用场景。

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
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。

限制项

- 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);
```

8 Stream增量数据流

8.1 概述

Table Store Stream 是一个数据通道，用于获取 Table Store 表中的增量数据。

您可以使用 Table Store Stream API 来获取这些修改内容。增量数据的重要性不言而喻，有了增量数据，可以进行增量数据流的实时增量分析、数据增量同步等。

原理

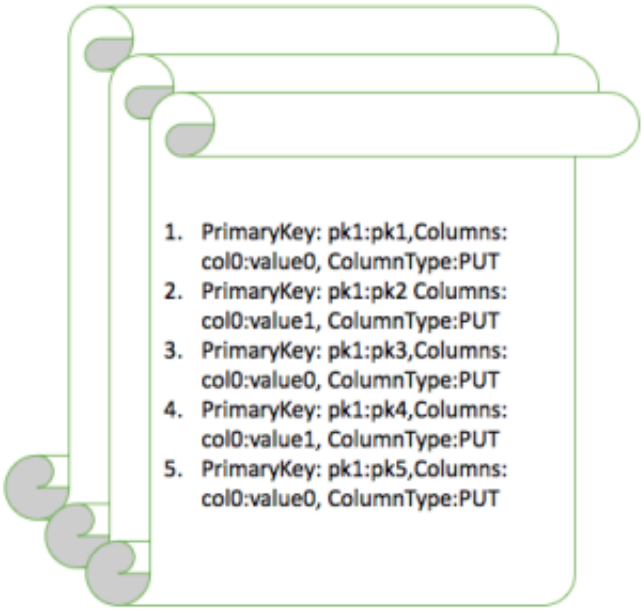
表格存储作为一款分布式 NoSQL 数据库，当有写操作（包括 put，delete，update）传入时，会把对应的修改记录存放在表格存储的 commit log 中，同时数据库也会定期做 checkpoint，旧的 commit 日志会被清除。

开启 Stream后，日志文件会被保存。在设置的保存期限内，可以通过 Stream 提供的通道读取这些增量数据。

由于表格存储的分区特性，同一分区的操作会共享一个 commit log，所以获取增量数据也是按照分区维度获取。

开启 Stream 时，会生成一个当前日志偏移量（即 iterator）并记录下来。用户可以通过 `GetShardIterator` 接口获取当前分区的 iterator，在后续读取该分区增量数据时传入这个 iterator，Stream 通道就可以知道从哪一行日志记录开始返回增量内容。返回增量内容的同时，Stream 也会返回一个新的偏移量，用于后续的读取。整个过程可以类比我们分页读取数据，iterator 就是分页的偏移量。

例如，我们有顺序地生成一批数据库日志文件，如下所示：



当我们在文件 A，第 3 行开始开启 Stream，则这个 iterator 就可以用来标识文件 A 的第 3 行。读取数据时传入这个 iterator，就能读到从上图中第三个操作 pk3 开始的后续修改操作。

Stream API 也提供了接口关闭这个数据通道。当用户再次开启时，Stream 会根据本次开启时间的日志偏移量重新生成一个当前分区的 iterator，我们可以用这个 iterator 读取该分区当前时间点之后的增量数据。

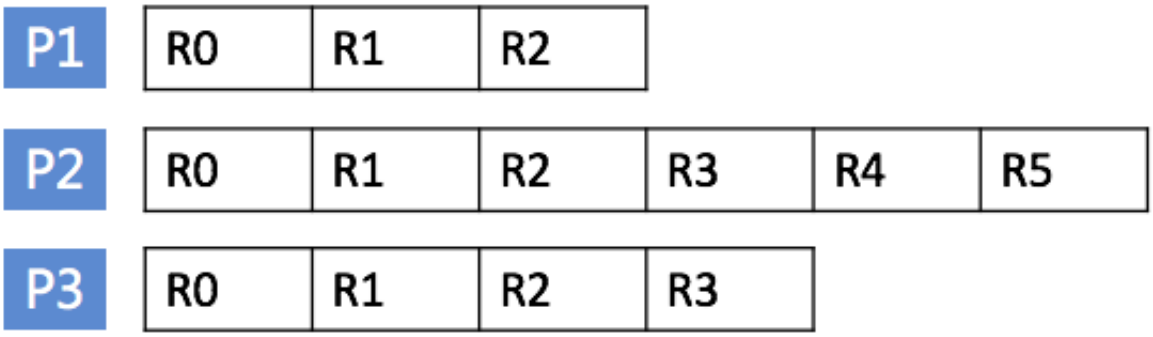
由于写操作会发生在同一个主键上，为了确保数据的一致性，对同一个主键的写操作需要顺序读取。然而，在读取增量数据之前，我们并不知道在哪些主键上发生了修改，所以读取增量数据的接口按照分区 id 来划分。用户可以通过罗列当前表下的所有分区来读取整张表的增量数据。Stream 通道会保证同分区内的写操作是顺序返回的，即，只要在同一分区内按照顺序读取，就可以保证相同主键的数据的一致性。如果持续读取所有分区的 Stream 数据，也就确保了整张表的增量数据都被会被读取到。

用户可以在创建表的时候开启 Stream，或者通过 `UpdateTable` 来开启或者关闭 Stream。当有一个 `put`，`update` 或者 `delete` 操作发生，一条修改记录会被写入 Stream，数据包含用户修改行的主键以及修改内容。

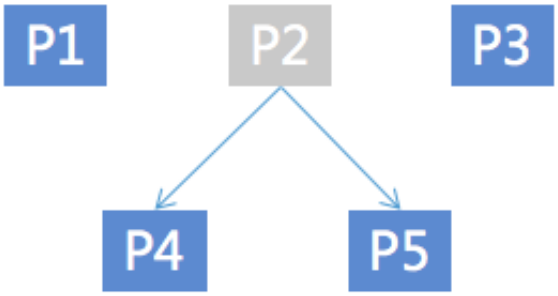
说明：

- 每个修改记录在 Stream 中存在且仅存在一次。
- 在每一个 shard 内，Stream 按照用户的实际操作顺序进行修改。但是不同 shard 的数据，不保证顺序。

示例



如上图所示，当前表有三个分区。图中的每一行表示一个分区，每一列表示当前分区的一次更新操作。每个分区维护自己的更新记录。我们可以通过 `DescribeStream` 接口得到分区的信息，然后顺序读取各自分区的数据。但系统会根据负载进行分区的分裂或者合并。分区的合并、分裂操作会生成新的分区，老的分区也将不会再产生新的增量数据。



上图中，分区 P2 发生了一次分裂，变成分区 P4 和分区 P5。我们可以并行读取分区 P4 和分区 P5 的数据，并不互相影响。但是在开始读取分区 P4 和分区 P5 之前，需要确保分区 P2 的增量数据已经全部读取完毕。

例如上图中，当开始读取分区 P4 的记录 R6 时，需要保证分区 P2 的 R5 已经被读取完毕，当 R5 被读完后，分区 P2 不会再生成新的数据。

8.2 Stream API/SDK

API

- 打开/关闭 Steam

  用户可以在创建表的时候设置 Stream 是否开启，也可以通过 `UpdateTable` 来开启或者关闭 Stream。`CreateTable` 和 `UpdateTable` 中新增了 `StreamSpecification` 参数，表示 Stream 的相关参数：

  - enable_stream：Stream 是否打开。
  - expiration_time：Stream 数据的过期时间，较早的修改记录将会被删除。

- 读取修改记录

  读取 Stream 数据的流程如下：

  1. 调用 `ListStreams` 获取当前表的 Stream 信息，例如 StreamID。详细信息请参见*ListStream*。
  2. 调用 `DescribeStream` 获取当前 Stream 的数据分片信息，例如 shard 的列表，每个 shard 记录又包含父 shard 信息、shardID 信息等。详细信息请参见*DescribeStream*。

3. 获取 StreamID 和 shardID 后，通过 `GetShardIterator` 获取当前 shard 的读 iterator 值，这个值标记着读取该 shard 记录的起始位置。详细信息请参见 *GetShardIterator*。
4. 调用 `GetStreamRecord` 来读取具体的修改记录，每次调用会返回新的 iterator，用于下次读取。详细信息请参见*GetStreamRecord*。

> 注意：
>
> - 同一个主键下的操作必须有序。在同一个 shard 下，Stream 做了这个保证。但是 shard 会出现分裂、合并等操作，所以您在读取某个 shard 的数据时，需要确保他的父 shard 以及 parent_sibling 的数据已经被读取。
> - 当读到空的 NextShardIterator 的时候，说明当前 shard 的增量数据已经全部读完，通常情况是该 shard 已经进入 inactive 的状态（发生分裂或者合并）。当一个 shard 已经被全部读完以后，您可以重新调用 `DescribeStream` 获取新的 shard 信息。

SDK

为了方便您使用 Stream API，Table Store 的 Java SDK 已经支持 Stream 接口。详细信息请参见 *Java SDK*。

8.3 Stream Client

基于 Table Store Stream API 以及 Table Store SDK，您可以使用 API 或者 SDK 读取 Stream 的记录。在实时获取增量数据的时候，需要注意分区的信息不是静态的。分区可能会分裂、合并。当分区发生变化后，需要处理分区之间的依赖关系以确保单主键上数据顺序读取。同时，如果您的数据是由多客户端并发生成，为了提高导出增量数据的效率，也需要有多消费端并行读取各个分区的增量记录。

Stream Client 可以解决 Stream 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 Stream Client 后，您只需要关心每条记录的处理逻辑即可。

下面内容将介绍 Stream Client 的原理，以及如何使用 Stream Client 高效打造适合自身业务的数据通道。

Stream Client 原理

为了方便做任务调度，以及记录当前每个分区的读取进度，Stream Client 使用了 Table Store 的一张表来记录这些信息，您可以自行选定表名，但是要确保该表名没有其他业务在使用。

Stream Client 中为每个分区定义了一个租约（lease），每个租约的拥有者叫做 worker。租约用来记录一个分区的增量数据消费者（即 worker）和读取进度信息。当一个新的消费端启动后，worker 会初始化，检查分区和租约信息，为没有相应租约的分区创建租约。当因为分裂或合并产生新的分区时，Stream Client 会在表中插入一条租约记录。新的记录会被某一个 Stream Client 的 worker 抢到并不断处理，如果有新的 worker 加入则可能会做负载均衡，调度至新 worker 处理。

租约记录的 schema 如下所示：

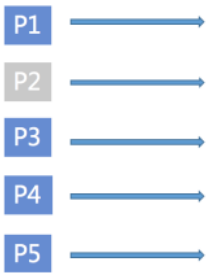
参数	说明
主键 StreamId	当前处理 Stream 的 Id。
主键 StatusType	当前 lease 的 Key。
主键 StatusValue	当前 lease 对应的分区的 Id。
属性 Checkpoint	记录当前分区 Stream 数据消费的位置（用户故障恢复）
属性 LeaseCounter	表示乐观锁。每个 lease 的 owner 会持续更新这个 counter 值，用来续租表示继续占有当前的 lease。
属性 LeaseOwner	拥有当前租约的 worker 名。
属性 LeaseStealer	在负载均衡的时候，表示准备挪至哪个 worker。
属性 ParentShardIds	当前 shard 的父分区信息。在 worker 消费当前 shard 时，保证父分区的 stream 数据已经被消费。

示例

下图是典型的使用 Stream Client 消费增量数据的分布式架构：

在这张图中，worker1 和 worker2 是两个基于 Stream Client 的消费端，例如在 ECS 上启动的进程。数据源不断地读写 Table Store 中的表，这张表初期有分区 P1、P2 和 P3。随着访问量和数据量的增大，分区 P2 发生了一次分裂，产生了 P4 和 P5。worker1 在初始阶段会消费 P1 的数据，worker2 消费 P2 和 p3 的数据，当 P2 发生分裂后，新的分区 P4 会被分配给 worker1，分区 P5 分配给 worker2。但 Stream Client 会确保 P2 的 R5 记录已经被消费完成后，再开始消费 P4 和 P5 的数据。如果这时部署了一个新的消费端 worker3，那可能发生的一件事情是 worker2 上的某个分区会被 worker3 调度走，由此产生负载均衡。

在上述场景下，Stream Client 会在表中生成如下租约信息：



StreamId	StatusType	StatusValue	Checkpoint	Counter	owner	Stealer	ParentShardIds
Table_123456	LeaseKey	ShardId1	checkpoint1	101	worker1		
Table_123456	LeaseKey	ShardId2	checkpoint2	95	worker2		
Table_123456	LeaseKey	ShardId3	checkpoint3	102	worker2		
Table_123456	LeaseKey	ShardId4	checkpoint4	55	worker1		p2
Table_123456	LeaseKey	ShardId5	checkpoint5	55	worker2		p2

Stream Client 中的 worker 是抽象消费 Stream 数据的载体，每一个分区会被分配一个 worker 即 lease 的拥有者，拥有者会不断地通过心跳，即更新 LeaseCounter 来续约当前的分区租约，通常每一个 Stream 消费端会起一个 worker，worker 在完成初始化后获取当前需要处理的分区信息，同时 worker 会维护自己的线程池，并发地循环拉取所拥有的各分区的增量数据。worker 初始化流程如下：

1. 读取 Table Store 的配置，对内部访问 Table Store 的客户端进行初始化。

2. 获取对应表的 Stream 信息，并初始化租约管理类。租约管理类会同步租约信息，为新分区创建租约记录。
3. 初始化分区同步类，分区同步类会维持当前持有的分区心跳。
4. 循环获取当前 worker 持有的分区的增量数据。

下载 Stream Client

- 下载安装 *JAR* 包
- Maven：

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

说明：

Stream Client 的代码是开源的，您可以下载源码了解原理，也欢迎您将基于 Stream 的优秀 Sample 代码分享给我们。

使用 Stream Client 接口

为了方便您使用 Stream Client 消费 Stream 数据、隐藏分区读取逻辑和调度逻辑，Stream Client 提供了 `IRecordProcessor` 接口。Stream Client 的 worker 会在拉取到 stream 数据后调用 `processRecords` 函数来触发用户的处理数据逻辑。

```
public interface IRecordProcessor {
    void initialize(InitializationInput initializationInput);
    void processRecords(ProcessRecordsInput processRecordsInput);
    void shutdown(ShutdownInput shutdownInput);
}
```

参数解释如下：

参数	说明
void initialize(InitializationInput initializationInput);	用于初始化一个读取任务，表示 stream client 准备开始读取某个 shard 的数据。
void processRecords(ProcessRecordsInput processRecordsInput);	表示具体读取到数据后用户希望如何处理这批记录。在 `ProcessRecordsInput` 中有一个 `getCheckpointer` 函数可以得到一个

参数	说明
	`IRecordProcessorCheckpointer`。这个接口是框架提供给用户用来做 checkpoint 的接口，用户可以自行决定多久做一次 checkpoint。
void shutdown(ShutdownInput shutdownInput);	表示结束某个 shard 的读取任务。

 说明：

- 因为读取任务所在机器不同，进程可能会遇到各种类型的错误，例如因为环境因素重启，需要定期对处理完的数据做记录（checkpoint）。当任务重启后，会接着上次的 checkpoint 继续往后做。也就是说，在极端情况下，Stream Client 不保证 `ProcessRecordsInput` 里传给您的记录只有一次，只会保证数据至少传一次，且记录的顺序不变。如果出现局部数据重复发送的情况，需要您注意业务的处理逻辑。
- 如果您希望减少在出错情况下数据的重复处理，可以增加做 checkpoint 的频率。但是过于频繁的 checkpoint 会降低系统的吞吐量，请根据自身业务特点决定做 checkpoint 的频率。
- 如果您发现增量数据不能及时被消费，可以增加消费端的资源，例如用更多的节点来读取 stream 记录。

下面给出了一个简单的示例，使用 Stream Client 来实时获取增量数据，并在控制台输出增量数据。

```
public class StreamSample {
   class RecordProcessor implements IRecordProcessor {

        private long creationTime = System.currentTimeMillis();
        private String workerIdentifier;

        public RecordProcessor(String workerIdentifier) {
            this.workerIdentifier = workerIdentifier;
        }

        public void initialize(InitializationInput initializationInput)
 {
            // Trace some info before start the query like stream info
etc.
        }

        public void processRecords(ProcessRecordsInput processRec
ordsInput) {
            List<StreamRecord> records = processRecordsInput.getRecords
();

            if(records.size() == 0) {
                // No more records we can wait for the next query
                System.out.println("no more records");
            }
```

```
            for (int i = 0; i < records.size(); i++) {
                System.out.println("records:" + records.get(i));
            }

            // Since we don't persist the stream record we can skip
blow step
            System.out.println(processRecordsInput.getCheckpointer().
getLargestPermittedCheckpointValue());
            try {
                processRecordsInput.getCheckpointer().checkpoint();
            } catch (ShutdownException e) {
                e.printStackTrace();
            } catch (StreamClientException e) {
                e.printStackTrace();
            } catch (DependencyException e) {
                e.printStackTrace();
            }
        }

        public void shutdown(ShutdownInput shutdownInput) {
            // finish the query task and trace the shutdown reason
            System.out.println(shutdownInput.getShutdownReason());
        }
    }

    class RecordProcessorFactory implements IRecordProcessorFactory {

        private final String workerIdentifier;

        public RecordProcessorFactory(String workerIdentifier) {
            this.workerIdentifier = workerIdentifier;
        }

        public IRecordProcessor createProcessor() {
            return new StreamSample.RecordProcessor(workerIdentifier);
        }
    }

    public Worker getNewWorker(String workerIdentifier) {
        // Please replace with your table info
        final String endPoint = "";
        final String accessId = "";
        final String accessKey = "";
        final String instanceName = "";

        StreamConfig streamConfig = new StreamConfig();
        streamConfig.setOTSClient(new SyncClient(endPoint, accessId,
accessKey,
                instanceName));
        streamConfig.setDataTableName("teststream");
        streamConfig.setStatusTableName("statusTable");

        Worker worker = new Worker(workerIdentifier, new ClientConfig
(), streamConfig,
                new StreamSample.RecordProcessorFactory(workerIden
tifier), Executors.newCachedThreadPool(), null);
        return worker;
    }

    public static void main(String[] args) throws InterruptedException
 {
        StreamSample test = new StreamSample();
```

```
        Worker worker1 = test.getNewWorker("worker1");
        Thread thread1 = new Thread(worker1);
        thread1.start();
    }
}
```

9 HBase 支持

9.1 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，用户不再需要关心 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：

	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。
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	需要自行保障集群的可用性。

	表格存储	自建HBase集群
	于 99.9%，数据可靠性达 99.99999999%。	
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。

9.2 Table Store HBase Client 支持的功能

表格存储与 HBase 的 API 区别

作为 NoSQL 数据库服务，表格存储对用户屏蔽了数据表分裂、Dump、Compact、Region Server 等底层相关的细节，用户只需要关心数据的使用。因此，虽然在数据模型及功能上相近，但表格存储并不完全与 HBase 相同，两者在 API 上有所区别。也正因如此，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

功能	支持情况
IsolationLevel:READ_UNCOMMITTED	不支持
IsolationLevel:STRONG	支持
IsolationLevel:TIMELINE	不支持

- Scan

  表格存储保证数据的强一致性，在数据写入 API 收到 HTTP 200 状态码（OK）的回复时，数据即被持久化到所有的备份上，这些数据能够马上被 Scan 读到。

功能	支持情况
指定 start、stop 确定扫描范围	支持
如果不指定扫描范围，默认扫描全局	支持
prefix filter	支持
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

功能	支持情况
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
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。

9.3 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。

9.4 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，但是也有一些事项需要注意。

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。

9.5 迁移较早版本的 HBase

Table Store HBase Client 目前支持 HBase Client 1.0.0 及以上版本的 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，用户需要管理 Connection 的生命周期，并在使用完以后将它 close。

最新的代码如下所示：

```
Connection connection = ConnectionFactory.createConnection(config);
// ...
connection.close();
```

TableName 类

1.0.0 之前版本的 HBase 中，用户在创建表时可以使用 String 类型的表名，但是 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，HBaseAdmin 类被新接口`org.apache.hadoop.hbase.client.Admin`取代。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();
```

9.6 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，主要包括下列操作：

- 配置依赖
- 连接表格存储
- 创建表
- 写数据
- 读数据
- 扫描数据
- 删表

代码位置

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。

使用 HBase API

- 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>
  ```

  更高级的配置请参考从 *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_FAMILY_NAME));

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

10 多元索引

10.1 简介

Table Store作为阿里云自研多模型数据库，基于其强大的分布式能力，我们致力于将其打造成一个在线数据平台。截至目前，Table Store已经可以提供大规模数据存储、高效的读写访问能力。

现在我们新推出的多元索引就是解决目前只能通过主键查询，而无法通过属性列查询、模糊查询和排序等查询能力不足的问题。多元索引又名SearchIndex，是一种高级索引结构，基于该索引可以实现众多高级查询功能。包括多字段ad-hoc查询、模糊查询、全文检索、排序、范围查询、嵌套查询、空间查询等。

目前我们已经开始在特定区域开放邀测，有兴趣的用户可以到阿里云官网表格存储控制台申请邀测，多元索引（SearchIndex）邀测阶段使用免费。

索引同步

开通多元索引的索引能力后，如果为某个表创建了多元索引（SearchIndex），则后续写入这个表中的数据会先写入存储模块。写成功后会立即返回用户数据写入成功，同时另一个异步线程会从存储模块读取新写入的数据然后创建SearchIndex，这个过程是一个异步过程，目前延迟在毫秒到秒级别。我们接下来会提供RPO监控，便于实时查看索引创建的延迟情况。采用异步方式创建多元索引并不影响、也不会阻塞Table Store原有的写入能力。

局限性

多元索引具备高级索引能力。和原有的Table Store KeyValue存储模块相比，有部分功能暂不支持，详细列表如下：

- TTL：多元索引部分的索引不支持TTL功能，目前如果原表开通了TTL，则不能创建多元索引（SearchIndex）。
- MaxVersion：SearchIndex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（SearchIndex）。
- 自定义版本号：如果在单版本中，用户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。

场景

Table Store提供了多元索引能力后，可以在众多场景中发挥更大的价值，比如：

- 元数据
- 时空数据
- 时序数据
- 全文检索

更多详情请参见*Table Store*发布多元索引功能#打造统一的在线数据平台文档。

10.2 功能介绍

不管是关系型数据库、还是NoSQL数据库，最基础的查询方式都是基于主键去查询。如果需要通过其他属性列去查询，就需要创建索引。为了使阿里云在线数据平台的功能更丰富，我们支持倒排索引以及其他一些索引：

- 倒排索引：是搜索系统中多种查询能力的基础结构，可以极大优化查询功能。为此，Table Store提供了倒排索引能力，用户为某些属性列建立了倒排索引后，可以基于这些倒排索引实现多字段自由组合的ad-hoc查询。同时也不用担心性别、年龄、枚举等选择性较差的字段问题了。
- 多维空间索引：是一种用于地理位置等多维空间查询的数据结构。一般都用于时空数据场景，可以极大提高空间查询的性能。为此，Table Store提供了多维空间索引。目前基于多维空间索引提供了地理位置的查询能力，包括附近的人、以及矩形、多边形等范围内的点等常见的地理查询，为大数据筛选、车联网和移动应用提供更丰富的一站式数据查询能力。
- 列式正排索引：同时为了更好地支持排序、统计聚合等功能，也提供了列式存储的正排索引。

基于上述三种基础索引能力，多元索引为用户提供下列功能：

非主键列的查询

之前的表格存储（Table Store）仅支持按主键列前缀查询数据，不支持非主键列前缀的查询，对用户有较大的使用限制。有了多元索引后，可以支持非主键列的查询。仅需要对要查询的列（Column）建立多元索引（CreateSearchIndex接口），即可通过该列的值查询数据。

多字段自由组合查询（Ad-hoc Query）

不管是NoSQL数据库系统，还是关系型数据库系统，都不能很好地支持多字段自由组合查询（ad-hoc query）。比如有一个表有4列，如果没有提前知道查询方式，或者需要支持多个字段的查询，在关系型数据库中可能需要建立7个二级索引。后期如果再增加第5列，那么索引个数可能会超过10个。而在多元索引中，只需要对这4个字段建立索引，同时还支持And、Or、Not等关系符号。

地理位置查询（GIS）

在车联网、物联网等位置相关的应用中，地理位置查询是一个极强的需求。比如查询附近的车，多边形范围内的点（电子围栏等）。目前表格存储多元索引提供了地理位置查询功能，用户可以基于此功能实现附近N公里内的车、矩形框或多边形框内的点等基本查询，为用户开发地理位置相关大数据应用提供了极大的便利性。

排序

排序是在线数据的一个常见需求，比如选择最多、最大、最小和最新等条件的数据。Table Store同样提供了强大的排序能力，包括正序、逆序、单条件、多条件等排序功能，为您存储在Table Store中的数据提供全局的多种排序能力。

全文检索

有了倒排索引后，Table Store也提供了分词能力。基于此，用户可以实现全文检索能力。但目前只有数据召回能力，不提供相关性能力。目前提供两种分词方式，单字分词（single_word）和最大语义分词（max_word）。

模糊查询

模糊查询是关系型数据库的一个强大功能，基于like语法可以实现很多易用性极高的功能。但是在分布式数据库中（比如HBase）目前仍无法提供模糊查询。现在Table Store提供了模糊查询能力，仅需为该属性列创建倒排索引，便可模糊查询该字段。

前缀查询

有了模糊查询能力后，Table Store也提供了前缀查询功能。

嵌套查询

在线数据中，除了扁平化的单层结构外，还存在一些更复杂的多层结构场景，比如图片标签。假设某个系统中存储了大量图片，每个图片都有多个实体，比如房子、轿车、人等。在每个图片中，这些实体占的位置、空间大小都不同，所以每个实体的价值（score）也不一样。这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，嵌套查询功能为复杂数据的建模提供了极大的便利性。

去重能力

基于以上的查询功能查询到结构后，有可能某个属性的数据非常多，导致结果多样性比较差。有了去重能力后，可以限制某个属性在一次结果中的最多个数，这样就能获取更好的结果多样性。

数据总行数

多元索引每次查询时都会返回这次请求命令中的数据行数。如果指定一个空查询条件，此时所有创建了索引的数据都符合该条件，返回的数据总行数就是表中已创建了索引的数据总行数。如果用户停止写入，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。这个功能适用于数据校验，运营等场景。

如果您在使用多元索引（SearchIndex）过程中遇到任何问题，可以通过以下方式联系我们：

- 阿里云官网控制台提交工单
- 表格存储技术交流钉钉群：11789671

10.3 使用指南

准备工作

在使用多元索引（SearchIndex）功能前，我们还需要做一些准备：

- 申请邀测：由于多元索引功能还在邀测阶段，所以需要申请邀测，具体请见官网控制台的相关页面。
- 准备SDK：目前仅Java SDK和Go SDK支持多元索引功能，请参考下文选择合适的SDK版本。

支持多元索引的SDK版本

Java SDK：

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>tablestore</artifactId>
  <version>4.7.4</version>
</dependency>
```

Go SDK：

请使用最新版tablestore go sdk：

```
go get -u github.com/aliyun/aliyun-tablestore-go-sdk
```

使用说明

- 多元索引是作为表上附加的搜索索引。一个表可以有多个多元索引，使用多元索引功能前必须先创建好多元索引。

- 创建好多元索引后，系统会自动进行索引的构建。如果表中原来已有数据，则会先完成全量数据的构建，再转入增量数据构建过程。进入增量构建过程后，用户每次新写入表中的数据，都会自动进入到多元索引中构建索引，但会存在一定的延迟。
- 多元索引提供了丰富的查询功能，各种查询功能的说明和示例代码见下文。

SearchIndex的FieldType类型

创建多元索引时需要指定索引的结构信息，包括每一列的类型(FieldType)，FieldType支持以下几种：

- Long：64位整型
- DOUBLE：64位浮点型
- BOOLEAN：布尔类型
- KEYWORD：字符串类型，与TEXT的区别是KEYWORD作为一个整体，不分词
- TEXT：分词字符串类型，与KEYWORD的区别是TEXT会进行分词
- GEO_POINT：地理位置类型，用经纬度数值("lat,lon")或者GeoHash字符串表示。需要特别注意，当用类似"-20,120"这种方式表示时，维度在前，经度在后。

SearchIndex支持的Query类型

- MatchAllQuery

  匹配所有行，常用于查询表中数据总行数，或者查看表中任意几条数据。
- MatchQuery

  匹配查询。采用近似匹配的方式查询表中的数据。比如查询的值为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
- MatchPhraseQuery

  短语匹配查询。短语匹配查询与匹配查询类似，但是要求查询的短语必须完整的按照顺序匹配。比如查询的值为“this is”，可以匹配到“...，this is tablestore”，"this is a table”。但是无法匹配到" this table is ..."以及"is this a table"。
- TermQuery

  精确查询。采用完整精确匹配的方式查询表中的数据，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。
- PrefixQuery

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。

- RangeQuery

  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。

- WildcardQuery

  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”。目前不支持以星号开头。

- BoolQuery

  复合条件查询。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，则要求这些子查询条件都不能满足。

- GeoBoundingBoxQuery

  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。

- GeoDistanceQuery

  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。

- GeoPolygonQuery

  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。

创建一个多元索引

如果要在某张表上使用Search功能，那么首先需要在这张表上创建一个多元索引。创建多元索引时，需要指定表名(TableName)、索引名(IndexName)、索引的结构信息(IndexSchema)。

- TableName：要创建的多元索引所属的表名。
- IndexName：要创建的多元索引的名字。
- IndexSchema：包含IndexSetting(索引设置)和FieldSchemas(Index的所有字段的设置)

- IndexSetting：RoutingFields(高级功能，可选配置)：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：FieldSchema的列表，每个FieldSchema包含以下内容：
  - FieldName(必选)：String。要索引的字段名，即列名。可以是主键列，也可以是属性列。
  - FieldType(必选)：字段类型，详见SearchIndex字段类型一节。
  - Index(可选)：bool值。是否开启索引。
  - IndexOptions(可选)：索引的配置选项。
  - Analyzer(可选)：分词器设置。
  - EnableSortAndAgg(可选)：bool值。是否开启排序与统计功能。
  - Store(可选)：bool值。是否在多元索引中附加存储该字段的值。开启后，可以直接从多元索引中读取该字段的值，而不必反查主表，可用于查询性能优化。

```
/**
 *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
```

```
import (
 "fmt"
 "github.com/aliyun/aliyun-tablestore-go-sdk/tablestore"
 "github.com/golang/protobuf/proto"
)

/**
 *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)。
 */
func CreateSearchIndex(client *tablestore.TableStoreClient, tableName
string, indexName string) {
 request := &tablestore.CreateSearchIndexRequest{}
```

```
 request.TableName = tableName // 设置表名
 request.IndexName = indexName // 设置索引名

 schemas := []*tablestore.FieldSchema{}
 field1 := &tablestore.FieldSchema{
  FieldName: proto.String("Col_Keyword"), // 设置字段名，使用proto.String
用于获取字符串指针
  FieldType: tablestore.FieldType_KEYWORD, // 设置字段类型
  Index:     proto.Bool(true), // 设置开启索引
  EnableSortAndAgg: proto.Bool(true), // 设置开启排序与统计功能
 }
 field2 := &tablestore.FieldSchema{
  FieldName: proto.String("Col_Long"),
  FieldType: tablestore.FieldType_LONG,
  Index:     proto.Bool(true),
  EnableSortAndAgg: proto.Bool(true),
 }
 schemas = append(schemas, field1, field2)

 request.IndexSchema = &tablestore.IndexSchema{
  FieldSchemas: schemas, // 设置SearchIndex包含的字段
 }
 resp, err := client.CreateSearchIndex(request) // 调用client创建
SearchIndex
 if err != nil {
  fmt.Println("error :", err)
  return
 }
 fmt.Println("CreateSearchIndex finished, requestId:", resp.ResponseIn
fo.RequestId)
}
```

列出多元索引名称

ListSearchIndex用于列出一个表下的所有多元索引。

```
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
}
```

```
func ListSearchIndex(client *tablestore.TableStoreClient, tableName
string) {
 request := &tablestore.ListSearchIndexRequest{}
 request.TableName = tableName // 设置表名
 resp, err := client.ListSearchIndex(request) // 获取表下所有SearchIndex
 if err != nil {
  fmt.Println("error: ", err)
  return
 }
 for _, info := range resp.IndexInfo {
  fmt.Printf("%#v\n", info) // 输出SearchIndex的信息
 }
```

```
 fmt.Println("ListSearchIndex finished, requestId:", resp.ResponseInfo
.RequestId)
}
```

查询多元索引的描述信息

DescribeSearchIndex用于查询多元索引的描述信息，包括多元索引的字段信息以及一些索引配置等。

```
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;
}
```

```
func DescribeSearchIndex(client *tablestore.TableStoreClient,
tableName string, indexName string) {
 request := &tablestore.DescribeSearchIndexRequest{}
 request.TableName = tableName  // 设置表名
 request.IndexName = indexName  // 设置索引名
 resp, err := client.DescribeSearchIndex(request)
 if err != nil {
  fmt.Println("error: ", err)
  return
 }
 fmt.Println("FieldSchemas:")
 for _, schema := range resp.Schema.FieldSchemas {
  fmt.Printf("%s\n", schema) // 输出SearchIndex中字段的schema信息
 }
 fmt.Println("DescribeSearchIndex finished, requestId: ", resp.
ResponseInfo.RequestId)
}
```

删除多元索引

DeleteSearchIndex用于删除一个多元索引。

```
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
}
```

```
func DeleteSearchIndex(client *tablestore.TableStoreClient, tableName
string, indexName string) {
 request := &tablestore.DeleteSearchIndexRequest{}
 request.TableName = tableName // 设置表名
 request.IndexName = indexName // 设置索引名
```

```
 resp, err := client.DeleteSearchIndex(request) // 调用client删除对应的多
元索引

 if err != nil {
  fmt.Println("error: ", err)
  return
 }
 fmt.Println("DeleteSearchIndex finished, requestId: ", resp.
ResponseInfo.RequestId)
}
```

MatchAllQuery

MatchAllQuery用于匹配所有行，常用于查询表中数据总行数，或者查看表中任意几条数据。

```
/**
 * 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();
    searchQuery.setQuery(new MatchAllQuery()); // 设置查询类型为
MatchAllQuery
    /**
     * MatchAllQuery结果中的TotalCount可以表示表中数据的总行数，
     * 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);
    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());
}
```

```
/**
 * 通过MatchAllQuery查询表中数据的总行数
 */
func MatchAllQuery(client *tablestore.TableStoreClient, tableName
string, indexName string) {
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.MatchAllQuery{} // 设置查询类型为MatchAllQuery
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchQuery.SetLimit(0) // 设置Limit为0，不获取具体数据
```

```
 searchRequest.SetSearchQuery(searchQuery)
 searchResponse, err := client.Search(searchRequest)
 if err != nil { // 判断异常
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess)
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 总行数
}
```

MatchQuery

MatchQuery采用近似匹配的方式查询表中的数据。

比如查询的值为"this is", 可以匹配到“... , this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。

```
/**
 * 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());
}
```

```
/**
 *  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。
 */
func MatchQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
```

```
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.MatchQuery{} // 设置查询类型为MatchQuery
 query.FieldName = "Col_Keyword" // 设置要匹配的字段
 query.Text = "hangzhou" // 设置要匹配的值
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchQuery.SetOffset(0) // 设置offset为0
 searchQuery.SetLimit(20) // 设置limit为20，表示最多返回20条数据
 searchRequest.SetSearchQuery(searchQuery)
 searchResponse, err := client.Search(searchRequest)
 if err != nil { // 判断异常
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows)) // 返回的行数
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody)) // 不设置columnsToGet，默认只返
回主键
 }
 // 设置返回所有列
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err = client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
```

```
}
```

MatchPhraseQuery

使用**MatchPhraseQuery**进行短语匹配查询。短语匹配查询与匹配查询类似，但是要求查询的短语必须完整的按照顺序匹配。比如查询的值为“this is”，可以匹配到“...，this is tablestore”、"this is a table”，但是无法匹配到"this table is ..."以及"is this a table"。

```
/**
 * 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());
}
```

```
/**
 * 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。
 */
func MatchPhraseQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.MatchPhraseQuery{} // 设置查询类型为MatchPhraseQuery
 query.FieldName = "Col_Text"  // 设置要匹配的字段
 query.Text = "hangzhou shanghai"  // 设置要匹配的值
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchQuery.SetOffset(0) // 设置offset为0
```

```
 searchQuery.SetLimit(20) // 设置limit为20，表示最多返回20条数据
 searchRequest.SetSearchQuery(searchQuery)
 searchResponse, err := client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
 // 设置返回所有列
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err = client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
}
```

TermQuery

TermQuery采用完整精确匹配的方式查询表中的数据，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。

```
/**
 * 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());
}
```

```
/**
 * 查询表中Col_Keyword这一列精确匹配"hangzhou"的数据。
 */
func TermQuery(client *tablestore.TableStoreClient, tableName string,
indexName string) {
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.TermQuery{} // 设置查询类型为TermQuery
 query.FieldName = "Col_Keyword" // 设置要匹配的字段
 query.Term = "hangzhou" // 设置要匹配的值
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchRequest.SetSearchQuery(searchQuery)
 // 设置返回所有列
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err := client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
}
```

PrefixQuery

PrefixQuery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。

```
/**
 * 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());
}
```

```
/**
 * 查询表中Col_Keyword这一列前缀为"hangzhou"的数据。
 */
func PrefixQuery(client *tablestore.TableStoreClient, tableName string
, indexName string) {
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.PrefixQuery{} // 设置查询类型为PrefixQuery
 query.FieldName = "Col_Keyword" // 设置要匹配的字段
 query.Prefix = "hangzhou" // 设置前缀
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchRequest.SetSearchQuery(searchQuery)
 // 设置返回所有列
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err := client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
```

```
  }
  fmt.Println("Row: ", string(jsonBody))
 }
}
```

RangeQuery

RangeQuery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。

```
/**
 * 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());
}
```

```
/**
 * 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。
 */
func RangeQuery(client *tablestore.TableStoreClient, tableName string
, indexName string) {
```

```
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 searchQuery := search.NewSearchQuery()
 rangeQuery := &search.RangeQuery{} // 设置查询类型为RangeQuery
 rangeQuery.FieldName = "Col_Long" // 设置针对哪个字段
 rangeQuery.GT(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
 searchQuery.SetSort(&search.Sort{
  []search.Sorter{
   &search.FieldSort{
    FieldName: "Col_Long",
    Order:     search.SortOrder_DESC.Enum(),
   },
  },
 })
 searchRequest.SetSearchQuery(searchQuery)
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err := client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
}
```

WildcardQuery

使用**WildcardQuery**进行通配符查询。

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”。目前不支持以星号开头。

```
/**
 * 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());
}
```

```
/**
 * 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
 */
func WildcardQuery(client *tablestore.TableStoreClient, tableName
string, indexName string) {
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.WildcardQuery{} // 设置查询类型为WildcardQuery
 query.FieldName = "Col_Keyword"
 query.Value = "hang*u"
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchRequest.SetSearchQuery(searchQuery)
 // 设置返回所有列
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err := client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
}
```

GeoBoundingBoxQuery

使用**GeoBoundingBoxQuery**进行地理边界框查询。

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。

```
/**
 * 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());
}
```

```
/**
 * 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。
 */
func GeoBoundingBoxQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.GeoBoundingBoxQuery{} // 设置查询类型为GeoBoundingBoxQuery
 query.FieldName = "Col_GeoPoint"  // 设置比较哪个字段的值
 query.TopLeft = "10,0"  // 设置矩形左上角
 query.BottomRight = "0,10" // 设置矩形右下角
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchRequest.SetSearchQuery(searchQuery)
 // 设置返回所有列
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err := client.Search(searchRequest)
 if err != nil {
```

```
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
}
```

GeoDistanceQuery

使用**GeoDistanceQuery**进行地理距离查询。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。

```
/**
 * 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());
}
```

```
/**
 * 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。
 */
func GeoDistanceQuery(client *tablestore.TableStoreClient, tableName
string, indexName string) {
 searchRequest := &tablestore.SearchRequest{}
```

```
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.GeoDistanceQuery{} // 设置查询类型为GeoDistanceQuery
 query.FieldName = "Col_GeoPoint"
 query.CenterPoint = "5,5"  // 设置中心点
 query.DistanceInMeter = 10000.0 // 设置到中心点的距离条件，不超过10000米
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchRequest.SetSearchQuery(searchQuery)
 // 设置返回所有列
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err := client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
}
```

GeoPolygonQuery

使用**GeoPolygonQuery**进行地理多边形查询。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。

```
/**
 * 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());
}
```

```
/**
 * 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。
 */
func GeoPolygonQuery(client *tablestore.TableStoreClient, tableName
string, indexName string) {
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
 query := &search.GeoPolygonQuery{} // 设置查询类型为GeoDistanceQuery
 query.FieldName = "Col_GeoPoint"
 query.Points = []string{"0,0","5,5","5,0"} // 设置多边形的顶点
 searchQuery := search.NewSearchQuery()
 searchQuery.SetQuery(query)
 searchRequest.SetSearchQuery(searchQuery)
 // 设置返回所有列
 searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
  ReturnAll:true,
 })
 searchResponse, err := client.Search(searchRequest)
 if err != nil {
  fmt.Printf("%#v", err)
  return
 }
 fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返回
结果是否完整
 fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
 fmt.Println("RowCount: ", len(searchResponse.Rows))
 for _, row := range searchResponse.Rows {
  jsonBody, err := json.Marshal(row)
  if err != nil {
   panic(err)
  }
  fmt.Println("Row: ", string(jsonBody))
 }
}
```

BoolQuery

使用**BoolQuery**进行复合条件查询。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。

```
/**
 * 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());
    }
}
```

```
/**
 * 通过BoolQuery进行复合条件查询。
 */
func BoolQuery(client *tablestore.TableStoreClient, tableName string,
indexName string) {
 searchRequest := &tablestore.SearchRequest{}
 searchRequest.SetTableName(tableName)
 searchRequest.SetIndexName(indexName)
```

```
 /**
  * 查询条件一：RangeQuery，Col_Long这一列的值要大于3
  */
 rangeQuery := &search.RangeQuery{}
 rangeQuery.FieldName = "Col_Long"
 rangeQuery.GT(3)

 /**
  * 查询条件二：MatchQuery，Col_Keyword这一列的值要匹配"hangzhou"
  */
 matchQuery := &search.MatchQuery{}
 matchQuery.FieldName = "Col_Keyword"
 matchQuery.Text = "hangzhou"

 {
  /**
   * 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 := &search.BoolQuery{
   MustQueries: []search.Query{
    rangeQuery,
    matchQuery,
   },
  }
  searchQuery := search.NewSearchQuery()
  searchQuery.SetQuery(boolQuery)
  searchRequest.SetSearchQuery(searchQuery)
  searchResponse, err := client.Search(searchRequest)
  if err != nil {
   fmt.Printf("%#v", err)
   return
  }
  fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返
回结果是否完整
  fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
  fmt.Println("RowCount: ", len(searchResponse.Rows))
 }
 {
  /**
   * 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
   */
  boolQuery := &search.BoolQuery{
   ShouldQueries: []search.Query{
    rangeQuery,
    matchQuery,
   },
   MinimumShouldMatch: proto.Int32(1),
  }
  searchQuery := search.NewSearchQuery()
  searchQuery.SetQuery(boolQuery)
  searchRequest.SetSearchQuery(searchQuery)
  searchResponse, err := client.Search(searchRequest)
  if err != nil {
   fmt.Printf("%#v", err)
   return
  }
  fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) // 查看返
回结果是否完整
  fmt.Println("TotalCount: ", searchResponse.TotalCount) // 匹配的总行数
```

```
    fmt.Println("RowCount: ", len(searchResponse.Rows))
  }
}
```

10.4 限制项

数值限制

Mapping：

名称	最大允许值	说明
Index Field数量	50	无
DocValues Field数量	50	无
Nested嵌套层数	1	最多嵌套一层nested
Nested Field数量	25	无
Table主键列长度之和	1000	所有PK列的长度累加后不超过1000
Table主键列中String长度	1000	所有PK列的长度累加后不超过1000
Table属性列中String长度（索引成Keyword）	4KB	无
Table属性列中String长度（索引成Text）	2MB	同Table中属性列长度限制

Search：

名称	最大允许值	说明
offset + limit	2000	如果超过请使用search_after
timeout	5s	无

类型限制

名称	允许值	禁止值
Index Field 类型	Table中的Double、String、Long、Boolean	Table中的Binary
	Index中Keyword、Text、Double、Long、Boolean	Index中Binary

名称	允许值	禁止值
DocValue类型	Table中Double、String、Long、Boolean	Table中Binary
	Index中Keyword、Long、Double、String	Index中Binary、Text
Store类型	Table中Double、String、Long、Boolean	Table中Binary
	Index中Keyword、Text、Double、Long、Boolean	Index中Binary
Geo Point	(lat, lng)：维度在前，经度在后	无

说明：

DocValue中的字段和排序相关，只有设置为true才支持排序，如果某个字段设置DocValues为false，则该字段不能再进行排序。

功能限制

名称	说明	替代方案
TTL：Time To Live	Table中支持TTL，多元索引中不支持TTL	按天、周或月创建Table，然后循环删除Table和Index即可
多版本	Table中支持多版本，多元索引中不支持多版本	版本号单独作为一个字段

其他限制

名称	值
功能开放区域	北京、上海、深圳、杭州

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，工单中请说明：场景、限制项名称、限制项的数量需求、为啥需要这么多大的原因，我们的工程师会在后续功能开发中优先考虑您的需求。

11 全局二级索引

11.1 使用前须知

使用表格存储全局二级索引前，您需了解全局二级索引中涉及的几个基本概念、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。

限制项

- 同一张主表下，最多建立16张索引表。
- 对于一张索引表，其索引列最多有四列（为主表PK以及主表预定义列的任意组合）。
- 索引列的类型为整型，字符串，二进制，布尔, 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。
- 索引表上不允许使用Stream功能。

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。

11.2 功能介绍

表格存储全局二级索引支持功能如下：

- 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PK3, PK2，没有属性列。
  - 索引列可以是PK3, 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Define0, 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)：即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。

目前表格存储的全局二级索引功能已经在张北公共云集群预发邀测，如要试用可以直接钉钉联系表格存储技术支持或者加入钉钉群 111789671。

11.3 使用场景

全局二级索引是表格存储提供的一个新特性。当您创建一张表时，其所有PK列构成了该表的一级索引：即指定完整PK就可以迅速查找到该PK所在行的数据。但在越来越多的业务场景中，需要对表的属性列、或者非首列PK进行条件上的查询。由于没有足够的索引信息，只能进行全表的扫描并配合条件过滤，来获取最终结果。当全表数据较多但最终结果很少时，全表扫描将浪费极大的资源。

Table Store 提供的二级索引功能（与*DynamoDB GSI*及*HBase Phoenix*方案类似），支持在指定列上建立索引，生成的索引表中数据按用户指定的索引列进行排序，主表的每一笔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。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
001	1532574644	123456	60

PK0	PK1	PK2	Defined0
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。

11.4 接口说明（以Java SDK为例）

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;
        }
    }
```

```
}
```

11.5 API/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*。

11.6 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);
    }
```