ALIBABA CLOUD

阿里云

表格存储Tablestore

最佳实践

文档版本：20220113

阿里云

法律声明

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

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

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

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

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

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

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

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

目录

1.表操作篇

本文将为您提供关于表设计的最佳实践。

如需了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

设计良好的主键

表格存储会根据表的分区键将表的数据自动切分成多个分区，每个分区调度到一台服务节点上。分区键的值是最小的分区单位，相同的分区键值下的数据无法再做切分。为了防止某一个分区键值的数据成为访问热点造成达到单机服务能力上限，应用程序需要让数据的分布和访问量的分布尽可能均匀。

表格存储会对表中的行按主键进行排序，合理设计主键可以让数据在分区上的分布更加均匀，从而能够充分地利用表格存储水平扩展的特点。

选取分区键时，建议遵循以下几个原则：

- 单个分区键值中的数据不宜过大，建议不超过 10 GB。

  > **说明** 单个分区键不超过10 GB 是为了避免访问热点，而不是数据存储的限制。

- 一张表内，不同分区键值中的数据在逻辑上是独立的。
- 访问压力不要集中在小范围连续的分区键值中。

使用示例

例如，有一张表中存储了某大学内所有学生使用学生卡消费的记录。主键列有学生卡 ID（CardID）、商家 ID（SellerID）、消费终端 ID（DeviceID）、订单号（OrderNumber）。同时我们有如下约定：

- 每一张学生卡对应一个 CardID，每一个商家对应一个 SellerID。
- 每一个消费终端对应 DeviceID，DeviceID 在全局是唯一的。
- 在每一个消费终端上产生的每一笔消费记录一个 OrderNumber。一个消费终端产生的 OrderNumber 是唯一的，但是在全局范围内 OrderNumber 不唯一。例如，不同的消费终端有可能产生两条完全不同的消费记录，但是它们的 OrderNumber 相同。
- 同一个消费终端产生的 OrderNumber 按时间排序，新的消费记录比老的消费记录拥有更大的 OrderNumber。
- 每笔消费记录均会被实时写入这张表中。

为高效利用表格存储，在设计表格存储的表的主键时，需考虑表的分区键：

分区方式	说明
使用 CardID 作为表的分区键	使用 CardID 作为表的分区键是一个比较好的选择。每天每张卡产生的消费记录数从总体上来讲是均匀的，每一个分区中的访问压力也应该是均匀的。以 CardID 作为表的分区键可以较好地利用预留读/写吞吐量资源。
使用 SellerID 作为表的分区键	使用 SellerID 作为表的分区键不是一个好的选择。因为学校内的商铺数量相对较少，同时一些商铺可能产生大量的消费记录成为热点，不利于访问压力的均匀分配。

分区方式	说明
使用 DeviceID 作为表的分区键	使用 DeviceID 作为表的分区键是一个比较好的选择。尽管每家商铺的消费记录数可能相差较大，但是每天每台消费终端上产生的消费记录数是可预期的。消费终端每天产生消费记录的条数取决于收银员操作的速度，这就决定了一台消费终端产生的消费记录数是受限的。因此，使用 DeviceID 作为表的分区键也可以保证访问压力的相对均匀。
使用 OrderNumber 作为表的分区键	使用 OrderNumber 作为表的分区键不是一个好的选择。因为 OrderNumber 是顺序增长的，因此在同一段时间内产生的消费订单的 OrderNumber 的值会集中在一个较小的范围内，这些消费订单记录会集中写入到个别的分区，以致预留读/写吞吐量没能得到高效利用。如果必须使用 OrderNumber 作为分区键，建议在 OrderNumber 上进行哈希散列，将哈希值作为 OrderNumber 的前缀，保证数据和访问压力的均匀。

总结

可以根据需求将 CardID 和 DeviceID 作为表的分区键，而不应该使用 SellerID 和 OrderNumber。之后再根据应用程序的实际需求来设计剩余的主键列。

通过拼接方式使用分区键

建议表格存储中表的同一分区键值下的数据量大小不超过 10 GB。如果您的表中单个分区键值的所有行的总数据量大小可能超过 10 GB，在设计表时可以将原来的多个主键列拼接成分区键。

使用示例

例如，上一小节中提到的学生卡消费记录表，假设主键为 DeviceID, SellerID, CardID, OrderNumber。DeviceID 是该表的分区键，单个 DeviceID 中所有行的数据量总大小可能超过 10 GB，可以将 DeviceID、SellerID 和 CardID 拼接作为表的第一个主键列（即分区键）。

原来的表如下所示：

DeviceID	SellerID	CardID	OrderNumber	attrs
16	'a100'	66661	200001	...
54	'a100'	6777	200003	...
54	'a1001'	6777	200004	...
167	'a101'	283408	200002	...

将 DeviceID、SellerID 和 CardID 拼接成分区键后的表如下所示：

CombineDeviceIDSellerIDCardID	OrderNumber	attrs
'16:a100:66661'	200001	...
'167:a101:283408'	200002	...

CombineDeviceIDSellerIDCardID	OrderNumber	attrs
'54:a1001:6777'	200004	...
'54:a100:6777'	200003	...

在原来的表中，Device=54 的两行是属于同一个分区键值为 54 下的两条消费记录。在新的表中，这两条消费记录拥有不同的分区键值。通过拼接多列主键列形成分区键的表减少了单个分区键值下的总数据量大小。

选择将 DeviceID、SellerID 和 CardID 拼接成分区键，而不选择将 DeviceID 和 SellerID 进行拼接的原因是，前一节提到的消费记录表约定所有 DeviceID 相同的消费记录其 SellerID 也相同，因此仅仅拼接 DeviceID 和 SellerID 并不能解决单个分区键值的数据量过大的问题。

但是，拼接主键列形成表有一些小瑕疵。DeviceID 是一个 Integer 类型的主键列，在原来的表中，DeviceID=54 的消费记录在 DeviceID=167 的前面。将前三列主键列拼接成 String 类型的主键列后，DeviceID=54 的消费记录在 DeviceID=167 的后面。假如应用程序需要范围读取 DeviceID 在 [15, 100) 之间所有的消费记录，上面的表无法满足需求。

为了应对这种状况，可以在 DeviceID 高位补 0。补 0 的个数取决于 DeviceID 最大位数。假设 DeviceID 的取值范围是 [0, 999999]，可以将 DeviceID 高位补 0 至 6 位后再进行拼接，得到的表如下所示：

CombineDeviceIDSellerIDCardID	OrderNumber	attrs
'000016:a100:66661'	200001	...
'000054:a1001:6777'	200004	...
'000054:a100:6777'	200003	...
'000167:a101:283408'	200002	...

经过高位补 0 后的表依然有一些问题。在原来的表中，DeviceID=54 的两行、SellerID='a1001' 的行应该在 SellerID='a100' 的后面。产生这种现象的原因是：'000054:a1001' 的字典序小于 '000054:a100:'，但是 'a1001' 的字典序大于 'a100'，连接符 `:` 影响了字典序。

在选取连接符时，应该选取比所有可用字符的 ASCII 码都小的字符作为连接符。在该表中，SellerID 的取值为数字、大小写英文字母。我们可以使用 `,` 作为连接符，因为 `,` 比所有 SellerID 可用字符的 ASCII 码都小。

使用 `,` 拼接后的表如下所示：

CombineDeviceiDSellerIDCardID	OrderNumber	attrs
'000016,a100,66661'	200001	...
'000054,a100,6777'	200003	...
'000054,a1001,6777'	200004	...
'000167,a101,283408'	200002	...

上面经过拼接形成的分区键的表的记录顺序就和原来的表保持一致了。

总结

当表中单个分区键值的所有行的数据量总大小可能超过 10 GB 时，可以将多个主键列拼接成分区键，以避免单分区键值的数据量大小限制。在拼接分区键时需要注意以下事项：

- 选取需要拼接的多个主键列，必须能有效地将原来表中相同的分区键值的记录，变成拥有不同分区键值的记录。
- 拼接 Integer 类型主键列时可以在高位补 0，保持记录的顺序一致。
- 选取连接符时需要考虑连接符对新的分区键的字典序的影响，选取比所有可用字符都小的连接符是一个比较安全的选择。

在分区键中加入哈希前缀

使用示例

尽量不要使用 OrderNumber 作为表的分区键。因为 OrderNumber 是顺序增长的，消费记录总是被写入最新的 OrderNumber 范围之内，旧的 OrderNumber 不再有写入压力，造成访问压力不均匀的现象，以致预留读/写吞吐量得不到高效利用。如果必须使用顺序增长的键值作为分区键，我们可以对分区键拼接哈希前缀，让相连的 OrderNumber 在表中随机分布，使访问压力分布均匀。

以 OrderNumber 为分区键的消费记录表如下所示：

OrderNumber	DeviceID	SellerID	CardID	attrs
200001	16	'a100'	66661	...
200002	167	'a101'	283408	...
200003	54	'a100'	6777	...
200004	54	'a1001'	6777	...
200005	66	'b304'	178994	...

对 OrderNumber 使用 md5 算法计算前缀（您也可以采取其他哈希散列算法），拼接成 HashOrderNumber。因为 md5 算法计算得到的哈希字符串可能过长，我们只需要取前几位就能达到让 OrderNumber 相连的记录在表中随机分布的目的。在这个例子中我们取前 4 位：

HashOrderNumber	DeviceID	SellerID	CardID	attrs
'2e38200004'	54	'a1001'	6777	...
'a5a9200003'	54	'a100'	6777	...
'c335200005'	66	'b304'	178994	...
'db6e200002	167	'a101'	283408	...
'ddba200001'	16	'a100'	66661	...

在后续访问消费记录时，使用相同的算法对 OrderNumber 计算哈希前缀，即可得到对应消费记录的 HashOrderNumber。在分区键中加入哈希前缀的弊端是，原来连续的记录会被打散，无法再使用 GetRange 操作读取一段范围内在逻辑上连续的记录。

并行写入数据

表格存储的表会被切分成多个分区，这些分区被分散在多个表格存储的服务器上。如果有一批数据要上传到表格存储中，同时这批数据是按主键排列顺序的，若按顺序写入数据，可能会导致写入压力集中在某个分区中，而其他的分区处于空闲状态，无法有效利用预留读/写吞吐量，影响数据导入速度。

可以采取以下任一措施来提升导入数据的速率：

- 将原始数据顺序打乱后再进行导入，以保证写入数据均匀地分配在各个分区上。
- 使用多个工作线程并行导入数据。把大的数据集合切分成很多个小集合，工作线程随机选取小集合进行数据导入。

区分冷数据和热数据

数据往往具有时效性。例如，存储消费记录表，近期产生的消费记录被访问的可能性较大，因为应用程序需要及时地对消费记录进行处理和统计，或者查询最近的消费记录。但是年代久远的消费记录被查询的可能性不大，这些数据渐渐成为冷数据，但仍然占用存储空间。

其次，表中存在大量冷数据会导致数据访问压力不均匀，从而导致表上配置的预留读/写吞吐量无法被充分利用。例如，已经毕业的学生的卡片，不会再产生消费记录。假如 CardID 是随着卡片申请时间递增的，以 CardID 作为分区键，会导致已经毕业的学生的 CardID 没有访问压力却被分配到预留读/写吞吐量，造成浪费。

为了解决这种问题，可以用不同的表来区分冷热数据，并设置不同的预留读/写吞吐量。例如，将消费记录按月份分表，每一个新的自然月就换一张新的表。当月的消费记录表需要不停写入新的消费记录，同时有查询操作。当月的消费记录表可以设置一个较大的预留读/写吞吐量配置来满足访问需求。前几个月的表由于不再写入新数据或者写入的新数据量较少，查询的请求较多，因此前几个月的消费记录表可以设置较小的预留写吞吐量，较大的预留读吞吐量。而历史超过一年的消费记录表，由于再被使用的可能性不大，可以设置较小的预留读/写吞吐量配置。已经超出维护年限的消费记录表可以将数据导出，存入 OSS（Object Storage Service）归档，或直接删除。

2.数据操作篇

本文将为您提供关于数据操作的最佳实践。

拆分属性列访问热度差异大的表

如果行的属性列较多，但是每次操作只访问一部分属性列，可以考虑将表拆分成多个表，将不同访问频率的属性列放到不同的表中。

例如，在商品管理系统中，每行存放商品数量、商品价格和商品简介。商品数量和商品价格均为占用空间较小的 Integer 类型，商品简介是 String 类型占用空间较大。大多数操作仅更新商品数量与商品价格，而不修改商品简介，所以商品简介的修改频率较低。这时候可以考虑将这个表拆分为两个，一个表存储商品数量和商品价格，另一个表存储商品简介。

压缩较大的属性列文本

如果属性列是较大的文本，应用程序可以考虑将属性列压缩之后再以 Binary 类型存储到表格存储中。这样做节省了空间、减少了访问的服务能力单元消耗，从而可以降低使用表格存储的成本。

将数据量超出限制的属性列存储到 OSS 中

表格存储限制单个属性列值不超过 2 MB。如需在单个属性列存储超过 2 MB 的数据 ，如图片、音乐、文件等，可以使用 OSS（Object Storage Service）对其进行存储。OSS是阿里云提供的开放存储服务，用以应对海量数据的存储和访问。OSS 的存储单价比表格存储更低，更适合存储文件。

如果应用程序不方便使用 OSS，可以将超过 2 MB 的单个值拆分成多个行，存储在表格存储中。

错误重试加入时间间隔

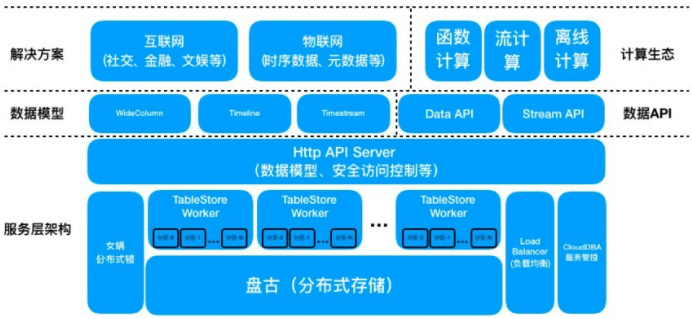
表格存储可能会遇到软硬件问题，导致应用程序的部分请求失败，并返回可重试的错误。建议应用程序遇到此类错误时等待一段时间后再进行重试，每两次重试之间应该加大时间间隔或引入随机时间间隔，避免重试的请求堆积到一个时间点上引发雪崩效应。

3.表设计最佳实践

3.1. 背景

了解表格存储表设计最佳实践的背景。选择使用表格存储后，根据实际业务场景，选择直接使用表格存储提供的数据模型或者根据最佳实践进行表设计。

为什么选择表格存储



- 09年阿里云成立之初立项，14年正式商业化
- 在内部承载阿里云、菜鸟、蚂蚁、天猫、钉钉、IoT、人工智能实验室等多个部门和领域的业务
- 历经多年双十一考验
- 同时在技术架构和产品层践行存储计算分离
- 多数据模型的设计，阿里云自研分布式多模型数据库
- 丰富的计算生态，与计算引擎的深度结合

如上图所示，表格存储提供了丰富、通用的功能，并具有如下优势：

- 零运维，即开即用，按量付费

  表格存储是阿里云上唯一一个Serverless的数据库，无需预定任何资源搭建服务，只需按使用量付费，简单易用，满足不同行业的大数据需求。

  如果使用针对大数据的开源数据库，则存在开源数据库依赖多，需要很多资源部署，运维成本较高等问题。

- 存储水平扩展，查询能力强

  表格存储可实现PB级数据的存储和查询，同时提供全局二级索引、多元索引等功能扩充查询能力，满足多种业务的负载和查询需求。

  如果采用组合使用MySQL、HBase、Elasticsearch等多款产品的方式实现不同业务的查询需求，则存在不同产品间数据同步负担。

- 模型和案例丰富，专家在线支持

  表格存储针对不同业务场景提供了相应的数据模型，简化业务的设计和开发，例如消息和Feed流提供Timeline模型，时序数据提供Timestream模型，科学大数据的多维网格数据提供Grid模型等。针对典型业务场景，提供丰富的场景案例和代码示例。

  建立表格存储技术支持群，技术专家实时在线进行技术支持和答疑工作，帮助您进行方案设计。

- 平台化功能演进

  表格存储不断的向着多模型和平台化的方向演进，可与各种生态对接，提供数据通道及场景化的解决方案。使用表格存储后，即可不断享受最新的技术红利。

为什么需要表设计最佳实践

表设计最佳实践可以帮助您在快速上手表格存储的同时，将表格存储的强大性能发挥到最优状态。

需要根据最佳实践进行表设计的原因如下：

- 数据规模大，应对海量数据仍需在数据库功能或者表设计上做一些取舍。

- 分布式数据库需要分区，而分区方式一般与用户表设计或者业务数据相关。

  数据规模小时，数据倾斜影响不大，而数据规模大到一定程度时，则需要考虑此问题。

- 数据库产品的功能丰富度和数据规模存在着矛盾。

  针对PB级数据设计的架构和功能，在查询丰富度上会有取舍，难以满足多样的查询需求。另一方面，选择灵活的查询方式可能意味着数据整体规模存在瓶颈，或者成本较高。因此不可避免的面临数据库选型或者是功能选型的问题。

- 数据库的整体性能与用户的使用方式、业务代码的质量等有很强的关系。

  有时，业务层感受到的数据库性能取决于业务层的DBA或者架构师对于数据库的了解和自身的代码水平。加深对数据库的了解，可以更好的应用此产品。

业务接入流程

当您有一块业务需要使用表格存储作为数据库时，通常会经历几个过程：了解表格存储（场景案例和文档），业务需求分析，直接对照方案实现、方案设计及编码实现（表结构和查询），接入业务数据测试上线。

根据业务场景能否直接使用表格存储的模型方案，实现过程分如下两种情况：

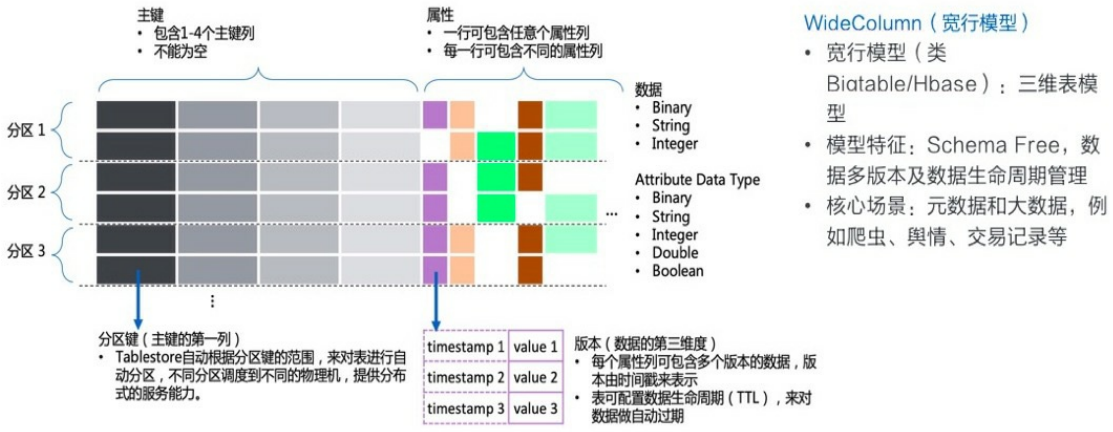
- 业务场景可以直接使用模型，例如Timeline、Timestream、Grid。此时直接使用模型，开发时也使用相应模型的SDK接口，无需进行表设计。
- 业务场景无法直接使用模型，此时需要根据业务需求和场景进行表结构和查询方案设计。

  对于简单的场景，例如简单的Key-Value模式进行存取，无需进行表设计；对于复杂的场景，例如数据规模很大且查询方式多样，此时需要进行表结构和查询方案的设计。

3.2. 索引介绍

本章节主要为您介绍表格存储的表引擎、全局二级索引、多元索引。

表引擎



如上图所示，表格存储的数据模型采用宽行模型。不同的分区可以加载到不同的机器上，实现水平扩展。下面为您分析表格存储的模型特性以及基于模型的查询方式。

- 多主键

表格存储的宽表模型包含多个主键列，多列主键列按照顺序共同构成一个主键，类似MySQL的联合主键，也可以把多列主键列拼接起来看作HBase的RowKey，每一列其实都只是整体主键的一部分。采用多列主键主要原因如下。

- 业务常需要多个字段来构成主键，如果只支持一个主键列，业务需要进行拼接，多列主键列避免了业务层做主键拼接和拆解。
- 第一个主键列是分区键，保证了分区键相同的行一定在同一个分区上。分区键可以帮助实现分区内事务（Transaction）、分区内自增列等功能。

> **说明** 主键的范围查询（GetRange接口）是指整体主键的范围，而非单独某一列的范围。

- 模型优势
  - 完全水平扩展，可支撑的读写并发和数据规模几乎无上限。表格存储可支持千万级的PTS/QPS，以及10PB级的存储量。当业务数据量大量上涨时，只要增加机器资源即可。同时，基于共享存储的架构也实现了动态负载均衡，不需要数据库层进行副本数据复制。
  - 提供了表模型。相比纯粹的Key-value数据库，表格存储具有列和多版本的概念，可以单独对某列进行读写。表模型也是一种比较通用的模型，可以方便与其他系统进行数据模型映射。
  - 表模型中，按照主键有序存储，而非Hash映射，因此支持主键的范围扫描。类似于HashMap与SortedMap的区别（此模型类似于SortedMap）。
  - Schema Free，即每行可以有不同的属性列，数据列个数也不限制。适合存储半结构化的数据。业务在运行过程中，可以进行任意的属性列变更。
  - 支持数据自动过期和多版本。每列都可以存储多个版本的值，每个值会有一个版本号，同时也是一个时间戳，如果设置了数据自动过期，就会按照这个时间戳来判断数据是否过期，后台对过期数据自动清理。
- 模型劣势
  - 数据查询依赖主键

    表格存储的数据模型类似于SortedMap，只能做点查和顺/逆序扫描，例如以下查询方式。

    - 主键点查

      通过已知主键，精确读取表上的一行。

    - 主键范围查

      按照顺序从开始主键（StartPrimaryKey）扫描到结束主键（EndPrimaryKey），或者逆序扫描。即对表进行顺序或逆序遍历，支持指定起始位置和结束位置。

    - 主键前缀范围查

      等价于主键范围查，主键前缀的一个范围可以转换成主键的一个范围，在表上进行顺序扫描即可。

  - 针对属性列的查询需要使用Filter

    Filter模式在过滤大量数据时效率有限，甚至变成全表扫描。数据查询的效率与底层扫描的数据量正相关，而底层扫描的数据量取决于数据分布和结构。数据默认仅按照主键有序存储，要按照某一属性列查询，符合条件的数据必然分布于全表的范围内，需要扫描后筛选。全表数据越多，扫描的数据量也就越大，效率也就越低。

在实际业务中，主键查询常常不能满足需求，而使用Filter在数据规模大的情况下效率很低，怎么解决这一问题呢？

数据查询的效率与底层扫描的数据量正相关，而Filter模式慢在符合条件的数据太分散，必须扫描大量的数据并从中筛选。解决这一问题有两种思路：

- 让符合条件的数据不再分散分布

  使用全局二级索引，将某列或某几列作为二级索引的主键。相当于通过数据冗余，直接把符合条件的数据预先排在一起，查询时直接精确定位和扫描，效率极高。

- 加快筛选的速度

  使用多元索引，多元索引底层提供了倒排索引、BKD-Tree等数据结构。以上面查询某属性列值为例，我们给这一列建立多元索引后，就会给这一列的值建立倒排索引，倒排索引实际上记录了某个值对应的所有主键的集合，即Value->List。那么要查询属性列为某个Value的所有记录时，直接通过倒排索引获取所有符合条件的主键，进行读取即可。本质上是加快了从海量数据中筛选数据的效率。

全局二级索引

全局二级索引采用的仍然是表引擎，主表建立了全局二级索引后，相当于多了一张索引表。索引表相当于给主表提供了另外一种排序的方式，即针对查询条件预先设计了一种数据分布，来加快数据查询的效率。索引的使用方式与主表类似，主要的查询方式仍然是上述的主键点查、主键范围查、主键前缀范围查。

例如我们有一张表存储文件的MD5和SHA1值，表结构如下。

FilePath(主键列)	MD5(属性列)	SHA1(属性列)
oss://abc/files/1.txt	0cc175b9c0f1b6a831c399e269772661	86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
oss://abc/files/2.txt	92eb5ffee6ae2fec3ad71c777531578f	e9d71f5ee7c92d6dc9e92ffdad17b8bd49418f98
oss://abc/files/3.txt	4a8a08f09d37b73795649038408b5f33	84a516841ba77a5b4648de2cd0dfcb30ea46dbb4

通过这张表，我们可以查询文件对应的MD5和SHA1值，但是通过MD5或SHA1反查文件名却不容易。我们可以给这张表建立两张全局二级索引表，表结构分别为：

索引1

MD5(主键列1)	FilePath(主键列2)
0cc175b9c0f1b6a831c399e269772661	oss://abc/files/1.txt
4a8a08f09d37b73795649038408b5f33	oss://abc/files/3.txt
92eb5ffee6ae2fec3ad71c777531578f	oss://abc/files/2.txt

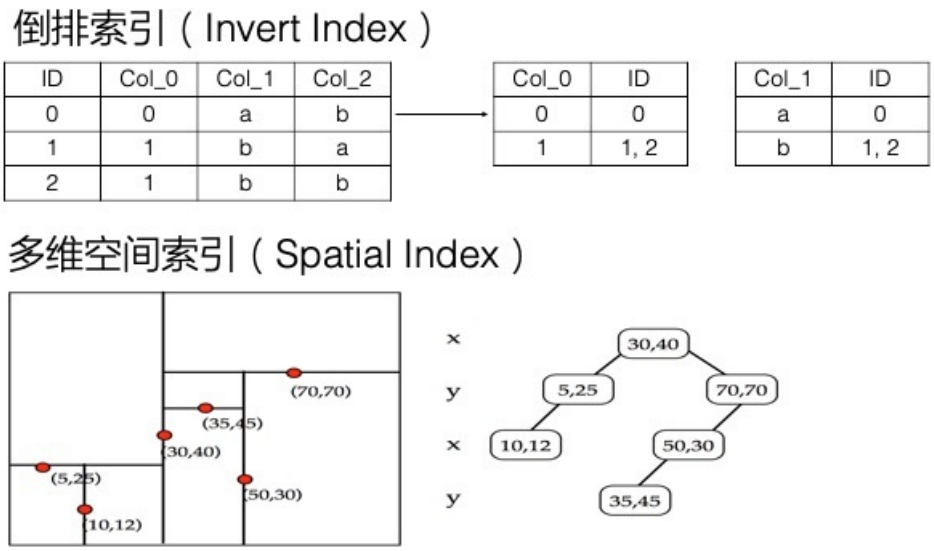
索引2

SHA1(主键列1)	FilePath(主键列2)
84a516841ba77a5b4648de2cd0dfcb30ea46dbb4	oss://abc/files/3.txt
86f7e437faa5a7fce15d1ddcb9eaeaea377667b8	oss://abc/files/1.txt
e9d71f5ee7c92d6dc9e92ffdad17b8bd49418f98	oss://abc/files/2.txt

为了确保主键的唯一性，全局二级索引会将原主键的主键列也放到主键列中，例如上面的FilePath列。有了上面两张索引表，就可以通过主键前缀范围查的方式里精确定位某个MD5/SHA1对应的文件名了。

多元索引

多元索引引擎相比于表引擎，底层增加了倒排索引、多维空间索引等，支持多条件组合查询、模糊查询、地理空间查询，以及全文索引等，还提供一些统计聚合能力。



倒排索引不仅可以解决单列值的检索问题，还可以解决多条件组合查询的问题。例如下表为一个订单记录。

订单号	订单（md5）（主键）	消费者编号	消费者姓名	售货员编号	售货员姓名	产品编号	产品名	产品品牌	产品类型	下单时间	支付时间	支付状态	产品单价	数量	总价钱
o000000000000000	c49f5fd5baa33159accae0d3ecd749a7	c0019	消陈九	s0020	售楚十	p0003004	vivox21	vivo	手机	2018-07-17 21:00:00		否	2498.999	2	4997.998

上面一共16个字段，需要按照任意多个字段组合查询，例如查询某一售货员、某一产品类型、单价在xx元之上的所有记录。这样的排列组合会有非常多种，因此不太可能预先将任何一种查询条件的数据放到一起，来加快查询的效率，这需要建立很多的全局二级索引。而如果采用Filter模型，又很可能需要扫描全表，效率不高。折中的方式是，可以先对某个字段建立二级索引，缩小数据范围，再对其中数据进行Filter。那么有没有更好的方式呢？

多元索引可以很好的解决这一问题，而且只需要建立一个多元索引，将所有可能查询的列加入到这个多元索引中即可，加入的顺序也没有要求。多元索引中的每一列默认都会建立倒排，倒排就记录了Value到List的映射。针对多列的多个条件，在每列的倒排表中找到对应的List，这个称为一个倒排链，而筛选符合多个条件的数据即为计算多个倒排链的交并集，这里底层有着大量的优化，可以高效的实现这一操作。因此多元索引在处理多条件组合查询方面效率很高。

此外，多元索引还支持全文索引、模糊查询、地理空间查询等，以地理空间查询为例，多元索引通过底层的BKD-Tree结构，支持高效地查询一个地理多边形内的点，也支持按照地理位置排序、聚合统计等。

3.3. 表设计

本章节主要为您介绍表格存储表设计的最佳实践。

> **说明** 关于表格存储索引选择的最佳实践，参见存储和索引的引擎详解。

主键设计——数据散列

- 为什么需要数据散列

  数据散列是分布式数据系统中的通常要考虑的问题，散列的目的是让数据分布更均匀，避免热点。假设数据分布不均匀，会出现以下问题：

  - 数据写入和读取能力受限于单个分区的能力，或者是单机能力，存在明显瓶颈。
  - 在某些数据处理场景下，热点或者数据分布不均会导致明显的长尾效应，拖慢整体速度。
  - 某个数据系统或者模块往往仅仅是整个业务链路上的一环，热点会拖慢整个上下游的速度。

  通常来讲，分布式数据库系统中，理想的数据和负载情况是：数据均匀分布，水平方式切分为很多分区，分布在不同机器上，读写压力也水平分散，每个请求的压力仅覆盖局部的一小部分，而不是整体。这种模式下完全水平扩展，业务压力增加，只需要增加机器资源即可。

  在业务设计上，应当尽量避免会导致数据热点的设计，在未来负载可支撑的情况下兼顾业务需求。有时业务需求会与数据均匀相矛盾，例如要按照时间全局有序的查询整个表最近写入的数据，那就与数据写入分散的原则有一些冲突。如果要让新写入的数据都集中在一起，系统就存在扩展瓶颈，那么在当前以及未来能否支撑这样的负载，未来系统的最大的写入TPS可能是多少，这些就是业务架构师需要考虑的问题。如果这种模式无法支持未来的负载，就意味着必须更改设计，否则是为当下或者将来埋坑。

  另一方面辩证的来看，如果业务需求很低，例如对于表格存储的TPS/QPS都在1000以下，整体数据在10GB下，且未来也不会有大的增长，那么热点问题也不是一个需要花太多精力考虑的问题，这样的负载单分区完全可以支撑。实际上，单分区也可以达到几万行/秒的写入，但是架构设计上不要依赖单分区能力，尽量控制在很安全的水平内。

- 常见热点问题

  下表一个常见的热点案例。这是一个简单的监控场景数据表，每次存储某个机器某个时间的一些指标值。这个表中主键有两列，分别是Timestamp和MachineIp。

Timestamp(主键列1，分区键)	MachineIp(主键列2)	Metrics(属性列)
1563617365001	10.10.10.1	{"cpu":10.0, "net_in": 10.0}
1563617365002	10.10.10.2	{"cpu":11.0, "net_in": 20.0}
1563617365003	10.10.10.3	{"cpu":12.0, "net_in": 30.0}

  这个表设计有很明显的热点问题，每次数据写入都是在表的末尾追加数据，而数据是按照分区键的范围进行分区的，也就是每次数据写入都会写入最后一个分区，而无法把写入负载平衡到多台机器上。这种情况下，单分区的写入能力就是整个表的写入能力上限，更重要的是，一旦发生热点，无法通过分片分裂来平衡负载，因为写入压力总是在写尾部。

- 解决方法一： 合理设置分区键

  针对上面例子中的热点问题，可以把MachineIp放到主键列的第一列，Timestamp放到第二列。这样就把写入压力按照Machine这个级别进行了分散，即写入的是每个Machine的尾部，大部分情况下也足够分散了。

  类似MachineIp这种分区键是很常见的一种分区方式，即把某个业务上比较分散的Key放到第一列，例如UserId，DeviceId、OrderId等等。这种模式只要这个Key本身比较分散，一般无太大问题。

  > **说明** 有一种局部热点情况，假设10.10.0.0/16这个网段的机器写入量很大，而表格存储是按照分区键的一个范围进行分区的，刚好这些机器又都分在一个分区内，会不会产生热点呢？如果写入压力超过或接近单分区的上限，确实是一个热点，但是表格存储具备自动负载均衡的能力，会自动将这个分区进行切分（Split），使得压力平均到两个分区上，如果仍不够会继续进行切分。

- 解决方法二： 拼接MD5

  上面的例子中，也可以通过拼接MD5的方式将IP本身的顺序打散，这种方式也较为常用。例如对IP计算，然后在IP前面接的前4位，如下图所示。

MachineIp(主键列1，分区键)	Timestamp(主键列2)	Metrics(属性列)
7552_10.10.10.2	1563617365000	{"cpu":10.0, "net_in": 10.0}
7552_10.10.10.2	1563617365001	{"cpu":10.0, "net_in": 10.0}
8d9c_10.10.10.3	1563617365003	{"cpu":10.0, "net_in": 10.0}
8d9c_10.10.10.3	1563617365004	{"cpu":10.0, "net_in": 10.0}
e5a3_10.10.10.1	1563617365000	{"cpu":10.0, "net_in": 10.0}

  这种方式可以避免某个IP段的热点，同时在结构上具备一定的模式，即以16进制字符串开头，这样有利于系统进行预分区。

- 主键顺序建议

  上面的例子中，很多用户以Timestamp作为第一个主键列，希望数据全局按照时间排序，这样可以按照时间范围进行查询，但是这种方式导致了尾部热点，给系统能力扩展带来了瓶颈。

  您可以通过以下方法解决对顺序性的要求。

  - 方法一：业务设计上避免对全局顺序性的要求，改为局部顺序性，例如上面的例子，在某个MachineIp下，数据仍是按照Timestamp有序的。即先通过一个字段来打散数据，再按照某种顺序查询。
  - 方法二：业务上采用分桶的方式，分散写入压力，在读取时从每个桶查询再合并。例如采用16个桶，每次写入时把时间对16取模，余数放到第一个主键列（0到15），时间放到第二个主键列。在读取时，从每个桶读取后合并即可。这种方式可以把压力分散到不同的桶上，能有多么分散取决于有多少个桶。在读取时需要对所有桶进行读取，可以采用并行读取的方式进行加速。
  - 方法三：表上仍然采用均匀的方式写入数据，给表建立一个多元索引，通过多元索引进行Timestamp等字段的有序查询。多元索引在数据分布上本身会进行一次散列，分成多个分片，查询时自动从多个分片合并数据。

其它主键设计建议

- 同一个分区键（分区键为第一列主键列）下数据不宜过多，例如10GB内（无硬性限制），因为相同分区键的行无法再进行分裂。这里相同分区键，指定的是对于第一列主键列的某个确定的值。对应上文的例子，即某个机器下的数据尽量不要超过10GB。

- 主键列的长度限制为1KB，尽可能使用较短的主键，有利于加快查询速度。

属性列设计

- 表格存储支持宽行，即一行可以非常宽，例如几十万个属性列。但是很宽的行，如果一次性读取，可能会读不出（超时），需要指定列或者分页读取某些列。因此，原则上不太建议非常宽的行（万列以上），可能会使某些功能受限。
- 属性列有2MB的限制，如果要保存的数据超过这一限制，需要把数据拆分成多列保存。

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。

3.4. 索引选择

本章节主要为您介绍如何选择表格存储的查询方式以及索引常见组合方案。

是否需要使用索引

以下情况您可以不使用索引进行查询：

- 如果基于主键和主键范围查询的功能已经可以满足业务需求，那么不需要建立索引。
- 如果对某个范围内进行筛选，范围内数据量不大或者查询频率不高，可以使用Filter，不需要建立索引。
- 如果是某种复杂查询，执行频率较低、对延迟不敏感，可以考虑通过DLA（数据湖分析）服务访问表格存储，使用SQL进行查询。

索引对比

- 全局二级索引

  一个全局二级索引是一个索引表，类似于主表，其提供了另一种数据分布方式（另一种主键排序方式），索引表可支撑的数据规模与主表相同。一个索引对应一种查询条件，预先将符合查询条件的数据排列在一起，查询效率很高，适合固定维度的查询。

  > **说明** 全局二级索引的主键设计也同样需要考虑散列问题。

- 多元索引

  一个多元索引是一系列数据结构的组合，其中的每一列都支持建立倒排索引等结构，查询时可以按照其中任意一列进行排序。一个多元索引可以支持多种查询条件，不需要对不同查询条件建立多个多元索引。相比全局二级索引，还支持多条件组合查询、模糊查询、全文索引、地理位置查询等。如果查询需求过于多变，需要多组合查询，则适合使用多元索引。多元索引本质上是通过各种数据结构加快了数据的筛选过程，功能丰富，但在数据按照某种固定顺序读取这种场景上，效率不如全局二级索引。

  多元索引的查询效率与倒排链长度等因素相关，即查询性能与整个表的全量数据规模有关，在数据规模达到百亿行以上时，建议使用RoutingKey对数据进行分片，查询时也通过指定RoutingKey查询来减少查询涉及到的数据量。简而言之，查询灵活度和数据规模不可兼得。

  多元索引适合如下的一些场景：

  - 需要对查询做任意组合
  - 需要有or的查询，如X=a OR X=b的场景
  - 需要GEO地理位置查询
  - 需要对字符串分词查询

两种索引的对比如下表：

对比项	全局二级索引	多元索引
查询灵活度	只可使用创建索引时指定的索引组合进行查询。如需其他组合需创建新的索引表。	可对索引字段做任意组合查询。
查询性能	通过索引键可定位到对应的”分片“，性能佳。	需要查询所有的分片。
大范围扫描	支持，性能与表引擎一致。	支持，但性能低于表引擎及全局二级索引。
数据可见延迟	毫秒级。	秒级。
分词查询	不支持。	支持。
GEO地理位置查询	不支持。	支持。
数据一致性	最终一致。	最终一致。

常见索引组合方案

每个业务都希望具备丰富的查询功能，但是在数据规模很大的情况下，灵活的查询意味着成本。例如万亿行数据的规模，使用表引擎，成本低；建立多元索引，查询灵活，但费用非常高昂；全局二级索引成本较低，但是只适合固定维度的查询。

常见的超大规模数据都带有一些时间属性，例如大量设备产生的数据（监控数据）或者人产生的数据（消息、行为数据等），这类数据非常适合采用表格存储进行存储。针对这类数据建立索引的组合方案：

- 对元数据表建立多元索引，全量数据表不建立索引或采用全局二级索引。
  - 元数据表可以是产生数据的主体表，例如设备信息表、用户信息表等。在时序模型中，产生数据的主体也可以认为是一个时间线，这条线会不断的产生新的点。
  - 表格存储的时序数据模型（Timestream）采用的也是类似的方式，对时序数据中的时间线建立一张表，专门用来记录时间线的元数据，每个时间线一行。时间线表建立多元索引，用来做时间线检索，而全量数据则不建立索引。在检索到时间线后，对某个时间线下的数据进行范围扫描，来读取这个时间线的数据。
- 热数据建立多元索引，冷数据不建立索引或者采用全局二级索引。
  - 很多情况下仅需要对非常热的数据进行多种维度查询，对冷数据采取固定维度查询（全局二级索引）即可。冷热分离可以给业务提供更高的性价比。
  - 目前多元索引还不支持TTL，需要业务层区分热数据和冷数据。

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎通过钉钉加入用户群11789671（表格存储技术交流群）或23307953（表格存储技术交流群-2）联系我们。