表格存储

最佳实践

最佳实践

本章将会提供一些关于使用表格存储的表的建议。

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。分区键是最小的分区单位，一个分区键下的数据无法再做切分。为了防止某一个分区键的数据成为访问热点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：

单个分区键中的数据不宜过大，建议不超过 1 GB。

说明：1 GB 是限制为了避免访问热点，而不是数据存储的限制。

同一张表不同分区键中的数据在逻辑上独立。

访问压力不要集中在小范围连续的分区键中。

使用示例

例如，有一张表，里面存储的是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，每一个分区键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。
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

建议表格存储表的单个分区下的数据量大小不超过 1 GB。如果您的表中单个分区键的所有行的总数据量大小可能超过 1 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 1 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	…
‘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。通过拼接多列主键列形成分区键的表减少了单个分区键下的总数据量大小。

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

当表中单个分区键的所有行的数据量总大小可能超过 1 GB 时，可以将多个主键列拼接成分区键，以避免单分区键的数据量大小限制。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的记录，变成拥有不同分区键的记录。

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

在前面**设计良好的主键**一节中已经提到，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键上。
- 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，在**设计良好的主键**一节中提到的存储消费记录表，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。

本章将会提供一些关于表格存储的数据操作的建议。

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。如果需要存储单个值超过 2 MB 的需求，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（请参见表格存储的错误信息）。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。