Table Store

Best Practices

MORE THAN JUST CLOUD | Alibaba Cloud

Best Practices

Table operations

This section provides some suggestions for using the table of Table Store.

A well-designed Primary Key

Table Store divides the table data into partitions dynamically according to Partition Keys, and each partition will be served by one server node. A Partition Key is the smallest partition unit. The data under a Partition Key cannot be split. Therefore, it is better for applications to make both data distribution and access distribution even in order to leverage Table Store’ s capability.

Table Store will sort the rows in a table by the Primary Key. A well-designed Primary Key can make the data distribution more even across partitions. This will make full use of the Table Store’ s high scalability.

When selecting a Partition Key, follow the principles below:

- Data of all rows in a single Partition Key should not be too large (Suggest not to exceed 1 GB. It is to avoid hot spot instead of actual data size limitation).
- The data in different Partition Keys of the same table are logically independent.
- The access pressure should not be concentrated on a small range of the consecutive Partition Keys.

Example:

If we have a table that stores all the students’ purchase records using their student cards in a university, the Primary Key columns would be the student card ID (CardID), the seller’ s ID (SellerID), the purchase terminal’ s ID (DeviceID) and the order number (OrderNumber). At the same time, we would have the following assumptions:

- Each student card corresponds to one CardID and each seller corresponds to one SellerID.

Each purchase terminal corresponds to a DeviceID which is globally unique.

For each purchase generated by a purchase terminal, one OrderNumber is recorded. The OrderNumber generated by a purchase terminal is unique, but this OrderNumber is not globally unique. For example, different purchase terminals may generate two completely different purchase records, but they may have the same OrderNumber.

The OrderNumbers generated by the same purchase terminal have different time stamp. The new purchase records will have larger sequential OrderNumbers than the previous purchase records.

Every purchase record will be written into this table in real time.

To use Table Store efficiently, pay attention to the table’ s Partition Key as follows when you design the table’ s Primary Key of Table Store.

Using CardID as the table’ s Partition Key

Using CardID as the table’ s Partition Key is a good choice. Generally, the number of purchase records for each card each day is relatively even, so the access pressure for each Partition Key will also be even. Using CardID as the table’ s Partition Key allows for good utilization of the reserved read/write throughput.

Use SellerID as the table’ s Partition Key

Using SellerID as the table’ s Partition Key is not a good choice. Because the number of on-campus sellers is relatively limited and, at the same time, some sellers may have a large number of purchase records and become a hot spot, which will not allow for an even allocation of the access pressure.

Use DeviceID as the table’ s Partition Key

Using DeviceID as the table’ s Partition Key is a good choice. Even though the number of purchase records for each seller may vary widely, the number of purchase records generated by each purchase terminal per day can be anticipated. The number of purchase records generated by a purchase terminal each day depends on the cashier’ s operating speed. This sets a limit on the number of purchase records that can be generated by a purchase terminal each day. Therefore, using DeviceID as the table’ s Partition Key can ensure a relatively even distribution of the access pressure.

Use OrderNumber as the table’ s Partition Key

> Using OrderNumber as the table’ s Partition Key is not a good choice. Because OrderNumbers increase sequentially, the OrderNumber values of purchase orders generated during same time period will be grouped in the same and narrow time period. These purchase order records will all be written in a single partition. Therefore, the reserved read/write throughput will not be used efficiently. If you must use OrderNumber as the Partition Key, you can hash it and use the hash value as the OrderNumber prefixes. This will ensure the even distribution of the data and access pressure.

In summary, CardID or DeviceID can be used as the table’ s Partition Key as needed, but SellerID or OrderNumber should not be used. And then you can design the remaining Primary Key columns based on the actual needs of the application.

Use the spliced Partition Keys

Table Store recommends that the data volume of a single partition should not exceed 1 GB. If the total data volume for all rows in a single table partition may exceed 1 GB, when designing the table, you can splice multiple original Primary Key columns into a Partition Key.

Example:

As in the student card purchase record example above, assume the Primary Key columns are [DeviceID, SellerID, CardID, OrderNumber]. DeviceID is the Partition Key for this table and the total data volume from all rows of a single DeviceID may exceed 1 GB. In this case, we can splice DeviceID, SellerID and CardID as the table’ s first Primary Key column (that is the Partition Key).

The original table is shown as follows:

DeviceID	SellerID	CardID	OrderNumber	attrs
16	‘a100’	66661	200001	…
54	‘a100’	6777	200003	…
54	‘a1001’	6777	200004	…
167	‘a101’	283408	200002	…

After splicing DeviceID, SellerID and CardID to create the Partition Key, the table produced is shown as follows:

CombineDeviceIDSellerIDCardID	OrderNumber	attrs
‘16:a100:66661’	200001	…
‘167:a101:283408’	200002	…
‘54:a1001:6777’	200004	…
‘54:a100:6777’	200003	…

In the original table, the two rows for Device = 54 belong to two purchase records in the same Partition Key 54. In the new table, these two purchase records have different Partition Keys. By splicing multiple Primary Key columns to form a Partition Key, you can reduce the total data volume for each Partition Key in the table.

We choose to splice DeviceID, SellerID and CardID into a Partition Key, rather than just DeviceID and SellerID. Because in the table mentioned in the previous section, all purchase records with the same DeviceID would also have the same SellerID. Only splicing DeviceID and SellerID would not solve the problem of too much data in a single Partition Key.

However, splicing the Primary Key columns to form a table has some drawbacks. DeviceID is an integer-type Primary Key column. In the original table, THE purchase records of DeviceID = 54 come before those of DeviceID = 167. After splicing the first three Primary Key columns into a string-type Primary Key column, the purchase records of DeviceID = 54 will come after those of DeviceID = 167. If the application needs to read all purchase records from the DeviceID range [15, 100), the above table will not be suitable.

In response to this situation, you can add 0s in front of the DeviceIDs. The number of added 0s can be determined by the maximum number of digits for DeviceIDs. If the DeviceID range is [0, 999999], you can add 0s so that all DeviceIDs have 6 digits and then splice. The resulting table will be as follows:

CombineDeviceiDSellerIDCardID	OrderNumber	attrs
‘000016:a100:66661’	200001	…
‘000054:a1001:6777’	200004	…
‘000054:a100:6777’	200003	…
‘000167:a101:283408’	200002	…

After padding 0s in front of the IDs, the table still has a problem. In the original table, of the two rows with DeviceID = 54, the row with SellerID = ‘a1001’ came after SellerID = ‘a100’. The discrepancy in the new table is caused by the fact that ‘000054:a1001’ is lexicographically less than ‘000054:a100:’, but ‘a1001’ is greater than ‘a100’. The connector “:” influences the lexicographic order. When selecting a connector, you should choose a character that is less than the ASCII code of all other available characters. In this table, the SellerID value uses numbers and uppercase/lowercase English letters. We can use “,” as the connector because “,” is less than the ASCII code of all characters available for the SellerID.

Using “,” and then splicing produces the following table:

CombineDeviceiDSellerIDCardID	OrderNumber	attrs
‘000016,a100,66661’	200001	…
‘000054,a100,6777’	200003	…
‘000054,a1001,6777’	200004	…

‘000167,a101,283408’	200002	…

In the above table created by splicing the Partition Key, the record order is consistent with that of the original table.

In conclusion, when the total data size for all rows in a single Partition Key may exceed 1 GB, you can splice multiple Primary Key columns to form a Partition Key in order to minimize the data size of an individual Primary Key. When splicing the Partition Key, pay attention to the following issues:

> When choosing the Primary Key columns to splice, be sure that the original rows of the same Partition Key will have different Partition Keys after splicing.
>
> When splicing integer-type Primary Key columns, you can add 0s before the numbers to ensure the rows’ order remains the same.
>
> When selecting a connector, you must consider its effect on the lexicographical order of the new Partition Key. The safe choice is to select a connector that is less than all other available characters.

Add hash prefixes in Partition Keys

Example:

In the well-designed Primary Key section, it’s mentioned that it is better not to use OrderNumber as the table’s Partition Key. Because the OrderNumbers increase sequentially, purchase records will always be written in the newest OrderNumber range, while the older OrderNumber ranges will not have any written pressure. This will cause an imbalance in access pressure so that the reserved read/write throughput is not used efficiently. If a sequentially increasing key value should be used as the Partition Key, we can splice a hash prefix to the Partition Key. In this way, the OrderNumbers will be randomly distributed throughout the table, evening out the access pressure distribution.

The purchase records table using OrderNumber as the Partition Key is as follows:

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	…

For the OrderNumbers, use the md5 algorithm to calculate a prefix (you may also use other hashing

algorithms) and splice it to create the HashOrderNumber. Because the hash strings calculated by the md5 algorithm may be too long, we only need to take the first few digits in order to achieve a random distribution of the records of sequential OrderNumbers. In this example, we take the first 4 digits and the table produced is as follows:

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	...

When subsequently accessing the purchase records, use the same algorithm to calculate the hash prefix of the OrderNumber to get the HashOrderNumber that corresponds to a purchase record. The downside of adding a hash prefix to the Partition Key is that the originally contiguous records will be dispersed. This means that the GetRange operation cannot be used to get a range of logically consecutive records.

Write data in parallel

When Table Store tables are split into multiple partitions, these partitions are distributed across multiple Table Store servers. When a batch of data must be uploaded to Table Store and it is ordered by a Primary Key, if the data is written in the same order, this may concentrate the written pressure on a certain partition, while the other partitions are idle. Thus, the operation will not make effective use of the reserved read/write throughput, which will impact the data import speed.

You can adopt any of the following methods to increase the data import speed:

Disrupt the original data order and then import. Ensure that the written data is evenly distributed across each Partition Key.

Use multiple worker threads for parallel data import. Split a large data set into many small sets. The worker threads will randomly select a small set to import.

Distinguish cold data and hot data

Data are often time sensitive. For example, for the table of purchase records in the well-designed

Primary Key section, the most recent purchase records may have a higher access probability since applications need to promptly process and compile statistics on purchase records or query the latest purchase records. However, the old purchase records are unlikely to be queried. These data gradually become cold, but still occupy storage space.

In addition, when there is a large volume of cold data in a table, it causes uneven access pressure, resulting in the inefficient use of the table’ s reserved read/write throughput. Back to the previous example, those cards belong to students who graduated will no longer generate purchase records. If the CardID numbers increase sequentially based on the card application date, then using CardID as the Partition Key means that the CardIDs of students who have graduated will have no access pressure, but still be allocated with the reserved read/write throughput. This would be a waste.

In order to solve this problem, we can use different tables to separate the cold and hot data, and set a different reserved read/write throughput for them. For example, purchase records may be divided into different tables by month, with a new table being created for each natural month. New purchase records will be constantly written in the table for the current month and query operations will also be performed. A high reserved read/write throughput may be set for the table of the current month to satisfy its access needs. For tables from the past few months, little or no new data will be written, but there will still be a lot of query requests. Therefore, these tables may be set with a low reserved write throughput and a high reserved read throughput. For the historical purchase record tables over a year old, there is little chance of use. Therefore, a low reserved read/write throughput can be set. The data from tables that have exceeded their maintenance period can be exported to restore in OSS (Alibaba Cloud Object Storage Service) archive or simply deleted.

Data operations

This section provides some suggestions on using the data operations of Table Store.

Split the tables with large access pressure variations among Attribute columns

If the rows have many Attribute columns but each operation only accesses a portion of these columns, you may consider splitting the table into multiple tables, with the Attribute columns of different access frequencies placed in different tables. For instance, in a merchandise management system, each row contains the item quantity, the item price and the item description. The item quantities and prices are integer-type values that take up little space, while item descriptions are string-type values that take up more space. The vast majority of operations will only update item quantities and prices, leaving the item description unchanged. The modification frequency of item descriptions is quite low. Therefore, you may consider splitting this table into two tables, one containing the item quantities and prices and the other containing the item descriptions.

Compress the Attribute column text of large amount

If the Attribute columns contain a large amount of text, the application should consider compressing the Attribute columns first and then storing them as the binary-type in Table Store. This will save space and reducing the capacity units consumed by access operations and the cost of using Table Store.

Store the Attribute columns with excessive data in OSS

Table Store limits the size of a single Attribute column up to 2 MB. If you need to store a single file whose size exceeds 2 MB (such as pictures, audio, or text), you can store it using OSS (Object Storage Service). OSS is an open storage service provided by Alibaba Cloud. It is used to store and access massive amounts of data. Unit storage prices in OSS are much lower than those in Table Store. This method is better for storing large files.

If it is inconvenient for the application to use OSS, individual values greater than 2MB can be split into multiple rows and stored in Table Store.

Add error retry intervals

Table Store may encounter hardware or software failures that cause some of an application’s requests to fail and return a “try again” error (for details, refer to Use Table Store’s API). It is recommended to wait for short period of time and then try again when an application encounters this type of error. Randomized or exponentially-increasing backoffs are in practice helpful to avoid the avalanche effects.