ALIBABA CLOUD

Alibaba Cloud

Tablestore

Best Practices

Document Version: 20201231

Alibaba Cloud

Legal disclaimer

Document conventions

Style	Description	Example
⚠ Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	⚠ **Danger:** Resetting will result in the loss of user configuration data.
🔔 Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	🔔 **Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
🔈 Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	🔈 **Notice:** If the weight is set to 0, the server no longer receives new requests.
? Note	A note indicates supplemental instructions, best practices, tips, and other content.	? **Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings> Network> Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Table operations

This topic provides you with best practices for table design.

A well-designed primary key

Tablestore dynamically divides table data into partitions based on the partition key, and each partition is hosted on one server node. The partition key is used as the minimum partition unit. Data under the same partition key value cannot split further. Applications must balance data distribution and access distribution across partitions to leverage the capability of Tabelstore.

Tablestore sorts the rows in a table by primary key. A well-designed primary key can better balance data distribution across partitions. This way, the high scalability of Tablestore can be fully utilized.

When you select a partition key, take note of the following items:

- Data of all rows in one partition key value cannot exceed 10 GB.

  Note Although 10 GB is not a hard limitation, this is recommended to avoid a hotspot.

- Data in different partition key value of the same table are logically independent.
- Do not concentrate the access load on a small range of consecutive partition key value.

Examples

For example, a table stores records of the transactions of all students using their student ID cards in a university. The primary key columns include CardID, SellerID, DeviceID, and OrderNumber. CardID indicates the ID of a student card, SellerID indicates the ID of a seller, DeviceID indicates the ID of a point-of-sale device, and OrderNumber indicates the number of an order. The records abide by the following conventions:

- Each student card corresponds to one CardID, and each seller corresponds to one SellerID.
- Each point-of-sale device corresponds to a globally unique DeviceID.
- For each purchase generated by a point-of-sale device, one OrderNumber is recorded. An OrderNumber generated by a device is unique to the device, but is not globally unique. For example, different point-of-sale devices may generate two separate purchase records using the same OrderNumber.
- Each OrderNumber generated by the same point-of-sale device has a different time stamp. New purchase records have larger sequential OrderNumbers than the previous purchase records.
- Every purchase record is written into the table in real time.

To optimize the use of Tablestore, consider the following partition keys when you design the primary key of a table in Tablestore:

Partitioning mode	Description
Use CardID as the partition key for the table	Using CardID as the partition key for the table is recommended. Generally, the number of purchase records for each card on each day is similar, thereby the access pressure for each partition is balanced. This allows for an efficient utilization of the reserved read/write throughput.

Partitioning mode	Description
Use SellerID as the partition key for the table	Using SellerID as the partition key for the table is not recommended. The number of sellers in a school is relatively small, and some sellers may generate a large number of purchase records and become hot spots. This does not help to balance access pressure for each partition.
Using DeviceID as the partition key for the table	Using DeviceID as the partition key for the table is recommended. Even though the number of purchase records for each seller per day varies, the number of purchase records generated by each purchase device per day can be estimated. This estimation is calculated based on a the order processing speed of a cashier, which determines the number of purchase records that can be generated by their purchase device per day. Therefore, DeviceID is suitable as the partition key of the table to guarantee a balanced distribution of access pressure.
Use OrderNumber as the partition key for the table	Using OrderNumber as the partition key for the table is not recommended. The OrderNumber is not recommended due to the sequential increase of purchase orders generated at the same time, resulting in grouped orders in the same time period. This restricts the effectiveness of the read /write throughput. If OrderNumber must be the partition key, you can hash it and use the resulting hash value as the OrderNumber prefix. This process will allow for even distribution of the data and reduce distribution pressure.

Summary

We recommend that you use CardID or DeviceID, instead of SellerID or OrderNumber, as the partition key of the table. After you specify the partition key, you can design the remaining primary key columns based on the actual requirements of your application.

Spliced partition key

For optimized Tablestore use, we recommend that the data volume of a single partition key value does not exceed 10 GB. If the total data volume for all rows in a single table partition key value exceeds 10 GB, you can splice multiple original primary key columns into a partition key when you design the table.

Example

As in the preceding student card purchase record example, assume the primary key columns are DeviceID, SellerID, CardID, and OrderNumber. DeviceID is the partition key for this table and the total data volume from all rows of a single DeviceID may exceed 10 GB. In this case, splice DeviceID, SellerID, and CardID as the first primary key column (partition key) of the table.

The following table shows the original table:

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

The following table shows the partition key created by splicing DeviceID, SellerID, and CardID:

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 whose DeviceIDs are 54 indicates two purchase records that have the same partition key value. In the newly created table, these two purchase records have different partition key values. You can reduce the total data volume for each partition key value in the table by splicing multiple primary key columns to form a partition key.

Create the partition key by splicing DeviceID, SellerID, and CardID instead of splicing DeviceID and SellerID. This is because in the table mentioned in the previous section, all purchase records with the same DeviceID have the same SellerID. The problem of too much data in a single partition cannot be resolved only by splicing DeviceID and SellerID.

Splicing the primary key columns to form a table presents some disadvantages. DeviceID is an INTEGER primary key column. In the original table, the purchase records whose DeviceIDs are 54 are listed before those whose DeviceIDs are 167. After splicing the first three primary key columns into a STRING primary key column, the purchase records whose DeviceIDs are 54 are listed after those whose DeviceIDs are 167. If the application must read all purchase records whose DeviceIDs range from 15 to 100, the preceding table is not optimal.

To address this situation, you can add zeros in front of the DeviceIDs. The number of zeros to add is determined by the maximum number of digits for DeviceIDs. If the DeviceID ranges from 0 to 999999, you can add zeros so that all DeviceIDs have 6 digits, and then splice. The following table shows the resulting table:

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

However, even after zeros are padded in front of the IDs, the table is still not fully optimized. This is because of the two rows whose DeviceIDs are 54 and the row whose SellerID is 'a1001' is listed after the row whose SellerID is 'a100'. This discrepancy is caused by `:` as the connector, which influences the lexicographic order. As a result, '000054:a1001' is lexicographically less than '000054:a100:', but 'a1001' is greater than 'a100'.

To resolve this issue, choose a character that is less than the ASCII code of all other available characters. In this table, the SellerID value uses letters and digits. We recommend that you use `,` as the connector because the ASCII code for `,` is less than the ASCII code of all characters available for the SellerID.

The following table shows the result after you use `,` and then splice the partition key.

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

In the preceding table produced by splicing the partition key, the record order is consistent with that of the original table.

Summary

If the total data size for all rows in a single partition key value exceeds 10 GB, you can splice multiple primary key columns to form a partition key to minimize the data size of an individual partition key value. When you splice the partition key, take note of the following items:

- When you choose the primary key columns to splice, make sure that the original rows of the same partition key value have different partition key values after splicing.
- When you splice INTEGER primary key columns, you can add zeros before the numbers to make the rows order remain the same.
- When you select a connector, consider its effect on the lexicographical order of the new partition key. The ideal method is to select a connector with an ASCII code that is less than all other available characters.

Add hash prefixes in partition key

Examples

We recommend that you do not use OrderNumber as the partition key for the table. Purchase records are always written in the latest OrderNumber range because OrderNumbers increase sequentially. As a result, earlier OrderNumber ranges do not experience any written pressure. This causes an imbalance in access pressure, which results in inefficient use of the reserved read/write throughput. If you must use a sequentially increasing key value as the partition key, splice a hash prefix to the partition key. This way, the OrderNumbers are randomly distributed throughout the table to better balance the access pressure.

The following table shows the purchase records using OrderNumber as the partition key.

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	...

As an example, for the OrderNumbers, you can use the md5 algorithm to calculate a prefix (other hashing algorithms are permitted) and splice it to create the HashOrderNumber. As the hash strings calculated by the md5 algorithm may be too long, you can take only the first few digits to achieve a random distribution of records of sequential OrderNumbers. In this example, the first 4 digits are used to produce the following table.

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 you subsequently access 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. One disadvantage of adding a hash prefix to the partition key is that the originally contiguous records are dispersed. As a result, the GetRange operation cannot be used to get a range of logically consecutive records.

Write data in parallel

When Tablestore tables are split into multiple partitions, these partitions are distributed across multiple Tablestore servers. If a batch of data is ordered by the primary key to be uploaded to Tablestore, and the data is written in the same order, this may concentrate the written pressure on a certain partition. This partition may have high pressure, whereas the other partitions remain idle. This operation does not fully utilize the reserved read/write throughput and may impact the data import speed.

To resolve this issue, use one of the following methods to increase the data import speed:

- Disrupt the original data order and import the data. Make sure that the written data is evenly distributed across each partition.
- Use multiple worker threads to import data in parallel. Split a large data set into multiple smaller sets. The worker threads then randomly selects a smaller set to import.

Distinguish cold data and hot data

Data are often time sensitive Use the student transaction records described in the preceding section as an example. Some purchase records may have a higher access probability because applications frequently query the latest record, and process and compile statistics based on the latest records. However, previous purchase records continue to occupy storage space and become cold.

If a large volume of cold data is included in a table, the reserved read/write throughput is ineffectively utilized, and results in unbalanced access pressure across the partitions. For example, the cards of students who graduated no longer generate purchase records. If the card IDs increase based on the date they were applied for and are used as the partition key, the records whose CardID belongs to graduate students do not have access pressure but waste reserved read/write throughput.

To effectively manage time sensitive data, use different tables to separate cold and hot data and set a different reserved read/write throughput for each table. For example, purchase records may be divided into different tables according to month. A new table is created for each month. New purchase records are constantly written to the table for the current month and query operations are also performed. A high reserved read/write throughput can be set for the table with the latest purchase records of the current month to satisfy its access needs. A low reserved write throughput and a high reserved read throughput can be set for previous tables of the past few months in which little or no new data is written, but queries are still performed. A low reserved read/write throughput can be set for tables that have exceeded their maintenance period such as historical records of a year or longer. These tables can be exported to restore in an OSS archive, or deleted.

2.Data operations

This topic provides you with best practices for data operations.

Split tables based on access frequency differences among attribute columns

If rows of a table have many attribute columns, but each operation accesses only a portion of these columns, you can split the table into multiple tables. The attribute columns of different access frequencies can be placed into different tables.

For example, in a merchandise management system, rows contain the item quantity, item price, and item description. Item quantities and prices are INTEGER values that consume little storage space. Item descriptions are STRING values that consume more storage space. Item descriptions are modified infrequently because most operations update item quantities and prices without modifying the item descriptions. The table can be split into two tables. One table contains item quantities and prices, and the other contains item descriptions.

Compress text-based attribute columns

If an attribute column contains a large amount of text, the attribute columns can be compressed and stored as BINARY data in Tablestore. This process saves space and reduces the capacity units consumed by access operations to reduce the cost of Tablestore usage.

Store attribute column in OSS

Tablestore limits the size of a single attribute column to 2 MB. If you must store a file that exceeds 2 MB, we recommend that you use Alibaba Cloud Object Storage Service (OSS).OSS is an open storage service provided by Alibaba Cloud to storage a large amount of data. Compared to Tablestore, OSS stores files at a lower unit price. Therefore, OSS is more suitable for storing objects.

If OSS cannot be used, the attribute column whose value is greater than 2 MB can be split into multiple smaller rows, and stored in Tablestore.

Add error retry intervals

Tablestore may encounter hardware or software problems, which result in some requests of the application to fail and return retries errors. If the request from an application fails and returns a try again error, we recommend that you wait a period of time before you try the request again. As a best practice, randomized or exponentially-increasing backoffs are helpful to avoid an avalanche effect.