Table Store

Best Practices

MORE THAN JUST CLOUD | Alibaba Cloud

Best Practices

Table operations

This article details recommendations for optimizing your use of Alibaba Cloud Table Store.

A well-designed primary key

Table Store dynamically divides table data into partitions according to the partition key, and each partition is hosted on one server node. The partition key value is the smallest partition unit. The data on the same partition key value cannot be split. In this case, applications must balance data distribution and access distribution across partitions to leverage Table Store’ s capability.

Table Store sorts the rows in a table by the primary key. A well-designed primary key can better balance data distribution across partitions, making full use of the Table Store’ s high scalability.

When selecting a partition key, note the following:

- Data of all rows in one partition key value cannot exceed 10 GB. While 10 GB is not a hard limitation, but 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.

Example

Assume you have a table that stores records of students’ transactions using their student ID cards. In this scenario:

- Each student card corresponds to one CardID.

- Each seller corresponds to one SellerID.

- Each point-of-sale device corresponds to a DeviceID, which is globally unique.
- 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 Table Store, CardID or DeviceID are recommended as the table’ s partition key:

- Using CardID is strongly recommended as, generally, the number of purchase records for each card, 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.
- Using DeviceID is recommended as, 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 cashier’ s order processing speed, which determines the number of purchase records that can be generated by their purchase device per day. Therefore, DeviceID is suitable as the table’ s partition key to guarantee a balanced distribution of access pressure.

Using the SellerID and OrderNumber are not recommended. The SellerID is not recommended because it indicates the limited number of sellers available, and therefore, does not help to balance access pressure for each partition in scenarios in which a small number of sellers generate the majority of purchase records. 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.

> **Note:** If OrderNumber must be the partition key, you can hash it and use the resulting hash value as the OrderNumber prefixes. This process will allow for even distribution of the data and reduce distribution pressure.

Spliced partition key

For optimized Table Store 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 designing the table.

Example

As in the preceding student card purchase record example, 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 10 GB. In this case, splice DeviceID, SellerID, and CardID as the table’ s first primary key column (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 new table 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 DeviceID = 54 belong to two purchase records in the same partition key value of 54. In the newly created table, these two purchase records have different partition key values. By splicing multiple primary key columns to form a partition key, you can reduce the total data volume for each partition key value in the table.

Splicing the primary key columns to form a table presents some disadvantages. DeviceID is an integer-type primary key column. In the original table, the purchase records of DeviceID = 54 are listed 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 are listed after those of DeviceID = 167. If the application needs to read all purchase records from the DeviceID range [15, 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 range is [0, 999999], you can add zeros so that all DeviceIDs have 6 digits, and then splice. The resulting table is as follows:

CombineDeviceiDSellerIDCardID	OrderNumber	attrs
‘000016:a100:66661’	200001	...

‘000054:a1001:6777’	200004	...
‘000054:a100:6777’	200003	...
‘000167:a101:283408’	200002	...

However, even after padding zeros in front of the IDs, the table is still not fully optimized. This is because of the two rows with DeviceID = 54; the row with SellerID = ‘a1001’ is listed after SellerID = ‘a100’ . This discrepancy is caused by : as the connector, which influences the lexicographic order, meaning ‘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 uppercase and lowercase letters and digits. We recommend , as the connector, because the ASCII code for , is less than the ASCII code of all characters available for the SellerID.

Using , and then splicing, produces the following optimized table:

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

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 splicing the partition key, note the following:

- When choosing the primary key columns to splice, be sure that the original rows of the same partition key value have different partition key values after splicing.

- When splicing integer-type primary key columns, you can add zeros before the numbers to make the rows order remain the same.

- When selecting 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

Example

In the A well-designed primary key section, we recommend that OrderNumber is not used as the table’ s partition key. Since OrderNumbers increase sequentially, purchase records are always written in the latest OrderNumber range, meaning earlier OrderNumber ranges do not experience any written pressure. This causes an imbalance in access pressure resulting in inefficient use of the reserved read/write throughput. If a sequentially increasing key value needs to be used as the partition key, splice a hash prefix to the partition key. In this way, the OrderNumbers are randomly distributed throughout the table to better balance the access pressure.

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	...

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 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. One disadvantage of adding a hash prefix to the partition key is that the originally contiguous records are 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. If a batch of data is ordered by the primary key to be uploaded to Table Store, 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, while 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 either of the following methods to increase the data import speed:

- Disrupt the original data order and then import. Make sure that the written data is evenly distributed across each partition.

- Use multiple worker threads for parallel data import. 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

Mismanaged time sensitive data can create problems. Using the previous example of student transaction records, 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, old purchase records continue to occupy storage space and become cold. If a large volume of cold data is included in a table (such as CardIDs of students no longer enrolled, yet retained in the system), the reserved read/write throughput is ineffectively utilized, and results in unbalanced access pressure across the partitions.

To effectively manage time sensitive data, use different tables to separate cold and hot data. Set a different reserved read/write throughput for each of them. For example, purchase records may be divided into different tables according to month, with a new table being created for each month. The reserved read/write throughput can then be set for each table as follows:

- 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 (new purchase records have a higher chance of being queried than legacy data).

- A low reserved write throughput and a high reserved read throughput can be set for later 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 then be exported to restore in an OSS archive, or deleted.

Data operations

This article provides recommendations for optimizing Table Store data operations. Notably, it details how to effectively manage the attribute columns and application request errors.

Split tables among attribute columns

If a table’ s rows have many attribute columns, but each operation only accesses 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 with rows containing the item quantity, item price, and item description:

- Item quantities and prices are integer-type values that consume little storage space, but are modified frequently.

- Item descriptions are string-type values that consume more storage space, but are modified infrequently.

Because the majority of operations only require updating the integer-type values of item quantities and prices, the table can be split into two tables, one containing these two values, the other containing the string-type 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-type in Table Store. This process saves space and reduces the capacity units consumed by access operations, to reduce the cost of Table Store usage.

Store attribute columns in OSS

Table Store limits the size of a single attribute column to 2 MB. If you need to store a file that exceeds 2 MB, we recommend that you use Alibaba Cloud Object Storage Service (OSS). OSS is an alternative storage service capable of storing large files at lower costs compared to Alibaba Cloud Table Store.

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

Add error retry intervals

If an application’ s request fails and returns a **try again** error, we recommend that you wait a period of time before trying the request again. As a best practice, randomized or exponentially-increasing backoffs are helpful to avoid an avalanche effect. For more error information, see Table Store API.