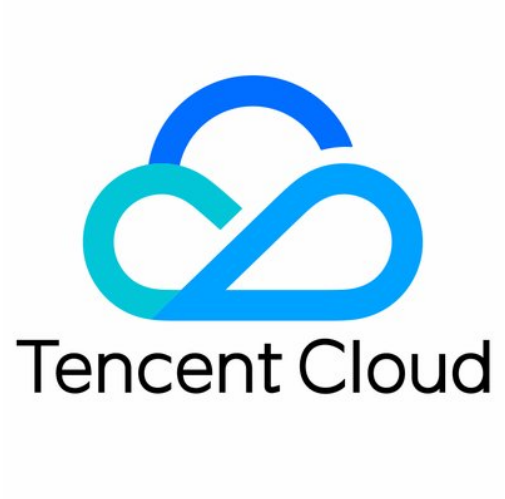
TencentDB for TcaplusDB

Practical Tutorial

Product Documentation



Contents

Practical Tutorial
Best Practice for Table Structure Design

Last updated：2024-12-04 10:16:20

Database design is important in the software development lifecycle. This document provides guidelines and best practices for database design.

Guidelines for Table Structure Definition

A field or table name should not contain any special characters, and the recommended length is no more than 32 bytes.
A field or table name should be meaningful and avoid misleading abbreviations if possible.
You should choose the right data types and not lose the needed precision of numbers.
Primary keys and shardkeys should be highly discrete for easier load balancing and scaling.
The number of indexes you create depends on the queries you need. Indexes should not have the same definition with primary keys.
We recommend the INT type for IP addresses of your business.
We recommend the LONG type for storing time data in seconds.
To guarantee operation atomicity, we recommend that logically-related fields be in the same table.
If you have high requirements on database performance, you can adopt a data redundancy design.
Primary keys and shardkeys should be highly discrete for easier scaling.
Primary keys should be readable for easier queries and troubleshooting. We do not recommend binary primary keys.
You should use brief primary keys if possible, which makes queries quicker.
You should define the size of a field based on your needs. Avoid such situations where a field defined with a large size has a small value when it is actually used.
You should add comments to all tables and fields.

Guidelines for Game Database Design

If a database needs to generate globally unique IDs, we recommend the `increase` command.
You should avoid database overloads when designing the database architecture. For example, you can add a queuing mechanism.
You should store relevant data in the same table to avoid data inconsistency.
We do not recommend implementing all features in a single table. A feature independent from others needs its own table.

If a table is frequently used and has a large number of records, you need to design a profile table so that the application does not need to access data from the original table, which lightens the database load.
Since chat in the game lobby can share the memory and chat in a single game can be pushed in real time, we do not recommend using databases in such scenarios. But databases are suitable for offline private chat.
We recommend using the array type provided by databases to handle historical game data, emails, and reports. This data type supports enqueue/dequeue and `TopN` in the order in which the data is enqueued.
We recommend using the sort component in ranking list design. If the ranking feature is implemented on the game server, we recommend storing the ranking results in databases asynchronously.
A marginal and time-consuming operation in the game needs a different process, which can avoid affecting the main logic of the game server.
The development cycle of a game is long and logically complex. During the development process, the logical data structure is likely to change. For scalability and ease of maintenance, we recommend that mutable data be defined as BLOB data in the game database table design phase. This type of data is serialized before being stored into databases to avoid frequent changes to the database tables due to data structure changes.

TDR Table Definition

Primary keys should be so discrete that the requests from the game server can be distributed to multiple access layer nodes.
We do not recommend that primary keys are identical with index keys in table definition, because identical keys waste network and disk resources.
To define non-primary key fields as arrays, the `refer` attribute should be added (`count` is the defined size, and `refer` is the actual size), so that the `count` size can be expanded later, and the data footprint in network transmission and on disks can be reduced.
The nesting depth of fields is limited to three and non-primary key fields should be at the first nesting level.
Binary primary key fields are not recommended as they work inefficiently in troubleshooting.
Up to eight indexes can be defined per table. We recommend two or three indexes per table. Too many indexes will reduce database performance, so please define indexes based on your needs.

Protocol Buffers Table Definition

Primary keys should be so discrete that the requests from the game server can be distributed to multiple access layer nodes.
Non-primary key fields can be defined as nested structures. Data access will be compromised if the nesting level is too deep.

Examples of Deprecated Design

Designs unable to meet the demands

Such designs lead to a lot of modifications. For projects right before the go-live phase, the modification cost is even higher.

Low performance

There are too many constraints between tables with large amounts of data; SQL query statements are complex due to the lack of fields well-designed for queries; there is no effective method to deal with tables with large amounts of data; there are too many views or views are not used efficiently.

Loss of data integrity

Badly-Designed primary or foreign keys cause program errors after update and deletion operations; data that has been deleted or dropped is used.

Poor scalability

The table has high affinity with the business and lacks the ability to scale, adapt to changes, or meet new demands.

A lot of redundant junk data

A lot of redundant junk data consumes resources and reduces query efficiency.

Fields inefficient in calculation or statistics

There are no fields designed for calculation or statistics; the fields used for calculation and statistics are scattered across multiple tables, making the calculation and statistics process cumbersome or even impossible.

No detailed data records

There are no fields that can be used to track data changes and user operations, or analyze data.

Tight coupling between tables

Tables are highly dependent on each other. Changes in one table will affect others.

Badly-Designed fields

The length of a field is too short or the type of a field is difficult to change later. Such fields will reduce scalability.

Best Practice for Database Interaction

Last updated：2024-12-04 10:16:20

Best Practices for Game Servers

1. In use cases where not all fields need to be read from or written to, we recommend reading and updating only the required fields to avoid fetching invalid data. This can reduce the size of data transmitted over a network and reduce the number of disk reads and writes.
2. If the data record needs to be returned when a write operation is performed, we recommend setting `result_flag` as needed. For example, if the updated data record is required after an update operation is performed, the `result_flag` should be set to 2. If the data record is not required during a write operation, the `result_flag` is set to 0.
3. For the command words in batch processing tasks, such as `batchget` , `listgetall` , and `getbypartkey` , we recommend getting the data record based on `offest` and `limit` . We recommend setting the `limit` to 200 and enabling multi-package return on the game server.
4. For tables supporting the LIST data type, when a `listaddafter` operation is performed, an enqueue/dequeue rule should be set in case that the list is full. We recommend the rule to add a new record to the front and delete a record from the rear, or vise versa.
5. For tables supporting the LIST data type, a correct index should be passed for `listreplace` , `listdelete` , and `listbatchdelete` operations.
6. Before the game server reads data, make sure that the non-primary key fields of the data record to be read are not empty. For example, when there are three non-primary key fields: A, B and C, if the game server only got A and B, then it indicates that field C is empty. Please check the fields before reading a data record.
7. The game server should control the timeout locally, and we do not recommend determining the timeout based on the response package sent from the game server.
8. When the game server performs traversal, we recommend accessing data from secondary nodes at the storage layer, so as to avoid affecting the performance of primary nodes.
9. We do not recommend performing `memset` operations on large fields to avoid consuming CPU resources. You can set the values of some fields in large data structures to 0 before initialization.
10. When processing requests to TcaplusDB and responses from TcaplusDB, the game server should use the divide-and-conquer method to process partial requests sent to TcaplusDB first and then process responses returned from TcaplusDB.

Best Practices for System Design

1. Create separate tables for the frequently used fields and the fields on which a one-time atomic operation needs to be performed.
2. When the game server writes data back, we recommend limiting traffic and discretize the data by time.
3. You can modify the table structure while using a table, and a table data conversion plugin is provided.
4. Rollback to the point in time of your cold backup or to an exact time at the table/logging level in all-server/all-region mode is supported.
5. Nodes at the access layer and the storage layer can be dynamically scaled for both all-server/all-region mode and multi-server/multi-region mode. We recommend the multi-server/multi-region mode.
6. Fields with a logical relation should be merged into one table to avoid distributed transaction problems.
7. We recommend enabling the compression feature, including the compression of request/response packages and logs.