KBOX Assignment

Database Schema Design Document

İlker Çelik
03.11.2020

Schema Design

According to the project specifications, recording the profile views can be persisted to a database table (PROFILE_VIEWS) which consists of three columns.

Structure of the table given below :

PROFILE_VIEWS		
viewee_user_id	viewer_user_id	viewedAt
INT	INT	TIMESTAMP

As stated in the specifications, data in this table has only one access (query) pattern. Specifically accessing views of a viewee's profile within 30 days should be queried efficiently.

Since we expect to have more than 20 millions of rows for a single day, number of rows on this table will increase dramatically over time. This will have a huge impact on query performance and querying profile views of a user will be very costly.

Although the table grows significantly over time, we do not need to query or store profile view records which are older than 30 days.

With the information given above, following design decisions are made for efficiency and maintenance.

Design Decision 1 - Do not introduce PK column

Because PROFILE_VIEWS table will store log-like (immutable events) data it won't be referenced by another table or rows on this table does not need to be identified with a unique id. So that we do not need a primary key column on this table. Since primary keys are also unique indexes, maintaining indexes has some costs on some database operations like insert, update etc.

Pros : Increased insert performance
Cons : Decreased storage usage, some migration tools may not work

Design Decision 2 - Create a proper index

Since *viewee_user_id* and *viewedAt* will always be present in the where clause of the profile view query, **creating an index** on these fields will significantly improve query performance.

Pros : Fast query execution
Cons : Increased storage usage, may introduce a little bit latency for inserts

Design Decision 3 - Handling huge amounts of data

According to the project specification profile visits which are older than 30 days won't be queried. So we can tweak our design to get rid of those obsolete data.

There three alternatives for removing unnecessary profile view records.

Alternative - 1

It is possible to delete the records older than 30 days. To remove these records we can schedule a job which executes DELETE statements against our table. However this approach has following disadvantages.

- Deleting records from the table can cause storage fragmentation
- Deleting records from the table can cause index fragmentation. Requires rebuilding indices regularly which can make the application unusable.
- Deleting records does not automatically return storage space to the operating system. Database specific maintenance tools (shrink, vacuum etc) must be run periodically.

Alternative - 2

To control the size of our PROFILE_VIEWS table we can introduce more than one table for each month in a year.

For example PROFILE_VIEWS_2020_01, PROFILE_VIEWS_2020_02, PROFILE_VIEWS_2020_03 etc.

Since querying last 30 days can maximum span two months, a scheduled job that runs at the end of each month can drop the tables which hold obsolete data.

We also need to create these window tables upfront. This can be done by a scheduled job or manually once a year.

Although this approach also has some advantages for controlling row count, index size and storage, it has some drawbacks.

These drawbacks are listed below :

- Requires creating tables in advance
- Because the table names must be dynamically changed on application code level, it increases complexity and maintenance costs of our application.
- Querying records for the last 30 days may span more than one table. So it should also be handled on application level. This complicates our codebase and decreases readability and maintainability.

Alternative - 3

Many production ready databases (PostgreSQL, SQL Server, Oracle etc.) provide built-in table partitioning functionality. Using this functionality. In our case, we can use this functionality and create range based one-month window partitions for our PROFILE_VIEWS table. In this way removing partitions which hold old data (including indexes) can be dropped instantaneously with a scheduled job without affecting the application execution.

As opposed to the Alternative - 2 this approach does not need any special application level code, because database management systems which support table partitioning have partition-pruning feature that optimizes queries to not to look up unnecessary partitions.

The drawbacks of this approach are :

- Requires creating partitions in advance
- Some of the databases does not support table partitioning

In conclusion, among the alternatives listed above, it is best to go with **Alternative - 3**. To implement this approach, we can automate database maintenance tasks as follows.

1. Schedule a cron job running every month which one-month ranged partitions for PROFILE_VIEWS table
2. Schedule a cron job running every month which drops partitions older than 2 months.