PISA UNIVERSITY

TASK 1
LARGE-SCALE AND MULTI-STRUCTURED DATABASES

FEASIBILITY STUDY FOR A KEY-VALUE TRASLATION OF THE DATABASE MODEL

ACADEMIC YEAR 2019-2020

STEFANO PETROCCHI, ANDREA TUBAK, FRANCESCO RONCHIERI, ALESSANDRO MADONNA

SUMMARY

STUDY

APPLICATION

DESCRIPTION

PisaFlix is an application that helps people to find the best place to watch a movie within Pisa. It provides all of the information regarding cinemas, films that are going to be projected and projection schedules. Users can also express their opinion by writing comments and adding favorites both to cinemas and films.

LOAD

The following list contains some estimates useful to understand the order of magnitude of the load that will affect our application:

- Pisa is a medium sized Italian city and it has a population of nearly 100˙000 inhabitants (including foreign students).
  Is estimated that about 50% of people in Italy goes to cinema at least once a year.
- Are estimated 1˙000 daily active users.
- Seven cinemas are located in Pisa, one of them has multi-room.
- About 50 new film are estimated to be projected every month.
- Cinemas are open every day from 3:00 pm to 1:00 am and we estimate 5 projection for each room of a cinema every day.

These are the numbers to consider for the future scalability of the application.

KEY-VALUE MODEL

CHARACTERISTICS

A ***key-value*** store is a database which uses an array of *keys* where each key is associated with only one *value* in a collection. The *key-value* stores usually do not have query languages as in ***relational databases*** to retrieve data. They only provide some simple operations such as get, put and delete.

Key-value databases work in a very different way from *relational databases*. *RDBs* pre-define the data structure in the database as a series of tables containing fields with well-defined data types. Exposing the data types to the database program allows it to apply a number of optimizations. In contrast, *key-value* systems treat the data as a single opaque collection, which may have different fields for every record. This offers considerable flexibility and more closely follows modern concepts like object-oriented programming. Because optional values are not represented by placeholders or input parameters, as in most *RDBs*, *key-value* databases often use far
less memory to store the same database, which can lead to large performance gains in certain workloads. Furthermore, in contrast with the *RDBs* that provides an *ACID* transaction model, a *BASE* approach is better suited for *key-value* databases.

DISADVANTAGES

Disadvantages of *key-value* models:

- The only queries that are efficient are simple, one-row-at-a-time queries.
- Is not really a data model, indeed there is no association between attributes that form an entity.
- Is hard to use most ordinary SQL operations such as JOIN or GROUP BY.
- There isn’t a possibility to choose an appropriate SQL data type for the value.
- There isn’t a possibility to use many SQL constraints such as FOREIGN KEY or NOT NULL.
- A lot more application code is needed to reassemble collections of *key-value* pairs into objects.

SUITABLE DATA TYPES

Types of data suitable for storing in a key-value pair:

- **Data of indeterminate form:** For example, each HTML page is different. Defining a schema for such page is complex. Since relational databases expect a schema, it is not possible to store the HTML page. *Key-value* data-store does not require a schema and it would be a best fit for such data.
- **Data of huge size and quantity:** *RDBs* are optimized for small rows that supports table fitting within a single server. In contrary Key-value data-stores support storing large objects with huge quantity, spread across multiple servers.
- **Unrelated Data:** Application might require storing unrelated data which is not suitable to be stored in *RDBs*. Since Key-value data stores are not based on relations, storing such unrelated data is supported.

LEVELDB

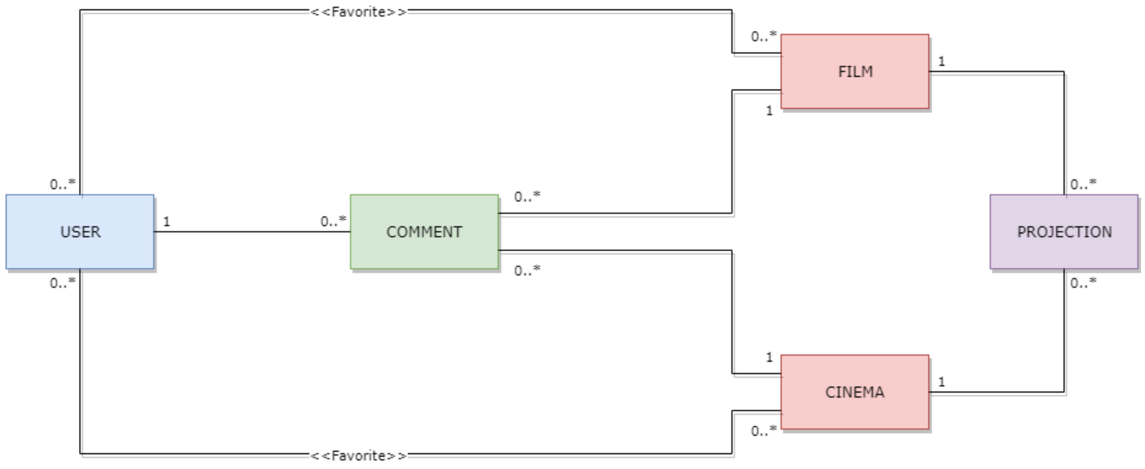
LevelDB is a *key-value* store built by Google. It can support an ordered mapping from string keys to string values. The core storage architecture of LevelDB is a log-structured merge tree (LSM), which is a write-optimized B-tree variant. It is optimized for large sequential writes as opposed to small random writes.

Key-value store supports the mapping from the key to the corresponding value. In SSTable the layout of key and value is managed as adjacent string sequence.

ENTITIES ANALYSES

In order to evaluate if some of the entities that populate our database are suitable to be stored on a key value database; let us analyse the structure and the scale of the number of transaction related to each entity.

Considering the database entities schema of the application:



USER

User entity represent a client or a not-technical administrator of the application. The information contained in this entity regards app login information, registry information and privilege level information. That information is small size data that can be naturally organized in a tabular form.

User is related with *Comment*, *Film* and *Cinema* entities.

An estimate of the typical transactions load for this entity in a day is:

- *Very small* number of ***write*** transactions (profile creation, deletion or modification).
- A number of ***read*** transactions less then active users (± *500* logs in).
- A number of new ***favourite*** cinema and film similar to that of log in.

FILM

Film entity represent a film and its description. The information contained in this entity can be of indeterminate form and dimension (i.e. for the description).

Film is related with *User*, *Comment* and *Projection* entities.

An estimate of the typical transactions load for this entity in a day is:

- *Very small* number of ***write*** transactions (small number of new films every month).
- A number of ***read*** transactions slightly greater than the number of active users (*> 1˙000*).

CINEMA

Cinema entity represent a cinema and few general information. That data is small sized and can be naturally organized in a tabular form.

Cinema is related with *User*, *Comment* and *Projection* entities.

An estimate of the typical transactions load for this entity in a day is:

- *Insignificant* number of ***write*** transactions.
- A number of ***read*** transactions slightly greater than the number of active users (*> 1˙000*).

COMMENT

Comment entity represent a user comment. The information contained in this entity can be of indeterminate dimension.

Comment can be related to cinemas and films but forms of comment that are nested or related to other entities could also be implemented.

An estimate of the typical transactions load for this entity in a day is:

- A number of ***write*** transactions similar to that of log in ($\pm$ *500*).
- A number of ***read*** transactions largely greater than the number of active users (*>>> 1˙000*).

PROJECTION

Projection entity represent the information of a single projection of a specific film in a specific cinema. That data is small sized and can be naturally organized in a tabular form.

Projection is related to *Film* and *Cinema* entities.

An estimate of the typical transactions load for this entity in a day is:

- $\pm$ *100* ***write*** transactions (five projection for a cinema room daily).
- A number of ***read*** transactions greater than the number of active users (*> 1˙000*).

CONCLUSION

ENTITIES SUITED TO A RELATIONAL DATABASE

From the entities analysis it has emerged that both read and write transactions regarding *User*, *Film* and *Cinema* entities will have a limited impact on the overall performance of the application. Even foreseeing a strong expansion of the application, a relational database will be able to handle this load just fine.

Furthermore, those entities are strongly related to each other; e.g. the possibility for a user to have favourite cinemas and films introduces the need to keep this relationship updated whenever a change occurs. In case of a Key value solution, this work would have to be done by application code which takes a considerable amount of time to develop.

User and *Cinema* contains small sized data and can be naturally organized in a tabular form.

> According to this study *User*, *Film* and *Cinema* entities are preferably mouldable through an ***RDB*** model because of their static structure and the reduced load that they entail in terms of transactions compared to the overall system.

KEY-VALUE RECOMMENDED ENTITIES

Given the fact that the vast majority of all read and write operations are related to the *Comment* and *Projection* entities, it is safe to say that the scalability of the application depends largely on the management of these entities. This is basically the end of the suitable traits for the *Projection* entity because old record of projections are almost useless and the operations that will work on them are going to die out over time. Moreover it is preferable to have consistent transactions for *Projection* in order to provide users correct and updated informations.

The *Comment* entity is much better suited for a Key-Value solution. The information contained in *Comment* can be of indeterminate form and dimension, plus, comments can be *eventually* consistent. Another important consideration is that we might want to change the structure of the *Comment* entity in order to support new ways of commenting stuff. For instance we might want to include emojis, gifs, stickers, images of different formats, audio files, etc. An RDB would have to be modified to accommodate each one of this changes, while our key-value store is already capable of handling all of this.

Having said all of that we have to admit that the *Comment* entity is not a perfect match either. The problem is that each comment is related with other entities, those relations can be handled by a key-value database at the cost of writing *ad hoc* code for cascades and joins management. We will have to implement some work around to make this operations fast enough to justify the use of a Key-Value database given the present implementation of comments.

> According to this study it is not recommended a *key-value* model also for *Comment* and *Projection* entities unless is expected a strong expansion of the application and an evolution of the *Comment* capabilites. In that case a **key-value** model could guarantee the necessary features to make the application scalable. Given its characteristics, *Comment* is more suited to be stored in a *key-value* database compared to all other entites; and this is why we are going to implement an hybrid solution just to store comments.