Key-Value DB Feasability Study and Implementation

Federico Fregosi, Mirko Laruina,
Riccardo Mancini, Gianmarco Petrelli

18/12/2019

Contents

1 Feasability study

Since there are few entities in our database, we decided to implement the whole database as a key-value database. By doing so, we learned how to manage different kinds of relationships between entities in a key-value database.

In the conversion from a relational database to a key-value one, we decided to proceed as follows:

1. identify a naive design for the keys
2. check the design again the use-cases to check problems or inefficiencies
3. add additional keys in order to fix the problems or speed up some use-cases
4. merge all changes in the final design

1.1 Naive design

We decided to generate ids incrementally by saving in the database the next id to be used. This value will be updated each time a new element is generated.

Another design choice we made is to represent the many to many relationship between users and chats through a list of chat members within the chat inside a structured attribute of the chat entity. Furthermore, we'll later see that this is not sufficient in order to provide efficient read operations and the redoundant list of chats of each user will be added as a structured attribute of the user entity.

Finally, the chat messages are saved under the chats and have an id that is unique only in the chat. Since messages cannot be deleted, no list is necessary.

The naive conversion is shown in the next table. A word preceeded by a dollar ($) indicates a value (that is the id in every case).

Key	Value
user:$userId:username	Users.username
user:$userId:password	Users.password
session:$sessionId:expiry	Sessions.expiry
session:$sessionId:userId	Sessions.userId
users:nextId	Next id for a user
chat:$chatId:name	Chats.name
chat:$chatId:lastActivity	Chats.lastActivity
chat:$chatId:admin	Chats.adminId
chat:$chatId:messages:nextId	Next id for a chat
chat:$chatId:message:$messageId:text	Messages.text
chat:$chatId:message:$messageId:timestamp	Messages.timestamp
chat:$chatId:message:$messageId:sender	Messages.senderUserId
chats:nextId	Next id for a chat
chatmembers:$chatId:$userId	Empty. Present iff user is member of chat

1.2 Use-cases analysis

Since mapping between use-cases and DB-abstraction-layer methods is straight-forward, this analysis will take into consideration the methods of the database abstraction layer interface.

Let's first analyze the methods that give some problems:

1.2.1 getChats

```
/**
* Returns the list of chats for the user identified by
* the given userId.
*
* @return the list of chats or null in case of error.
*/
List<Chat> getChats(long userId);
```

The above methods is inefficient because, in order to retrieve the chat a user is member of, I should try all chat ids to see if the key exists. As an alternative, I could take advantage of the ordering property of LevelDB but the same problem would arise in the complimentary method that retrieves the chat members.

Since the number of chats a user is member of and the number of chat members are small in practice we decided to replace the intermediate entity with two structured attributes of chat and user, respectively (see the table below). The attributes hold a comma-separated list of ids. The redoundancy is necessary in order to make both operations of retrieval of chat members efficient.

chat:$chatId:members	List of chat members
user:$userId:chats	List of chat ids where the user is present

This implementation increases the complexity and the time required for all those other methods that modify said lists, namely:

```
boolean addChatMember(long chatId, long userId);
boolean removeChatMember(long chatId, long userId);
```

This is due to the fact that both lists must be downloaded, edited and uploaded back. However, this is not a big deal since addition and removal are fairly unfrequent operations.

Another operation that is involved by this change is *checkChatMember* (see below). Compared to the "naive" implementation, it is slower but the time penalty is not as significant as in the previous case, due to the fact that a chat holds few members in practice.

```
boolean checkChatMember(long chatId, long userId);
```

1.2.2 getUser

```
/**
* Returns user identified by the given username.
*
* @return the user in case of success, null otherwise.
*/
User getUser(String username);
```

This method is inefficient using the proposed naive implementation since it would require a linear search among all users. The use of a reverse index to map usernames to ids would make it more efficient, i.e.:

username:$username:userId	Users.userId

1.2.3 existsChat

```
/**
* Returns true if there exists a private chat between user1 and
* user2.
*
* NB: a private chat is a chat with only two members.
*
* @return {@code true} if it exists,
*         {@code false} otherwise or in case of errors.
*/
boolean existsChat(long user1, long user2);
```

This method can be implemented in a simple way iterating through the user's chats and its members. A more efficient way would be inserting a redundant list of private chats the user is involved in. However, since this is just a proof-of-concept implementation and this method gets called only on a not frequent operation (chat creation), we decided not to perform this optimization.

1.2.4 getChatMessages

```
/**
* Returns a list of messages for the given chat, in the given
* time range, up to the given number of elements, sorted in
* ascending message sending time.
*
* @param chatId id of the chat whose messages are to be retrieved
* @param from start of the time range (included). It can be null,
*        whose meaning is that there is no lower bound.
* @param to end of the time range (excluded). It can be null,
*        whose meaning is that there is no upper bound.
* @param n maximum number of elements to return.
*        If from is not null, messages are counted from
*        {@code from} up to n or {@code to}. If from is null,
*        messages are counted from {@code to} up to n or {@code from}.
* @return the list of messages or null in case of error.
*/
List<Message> getChatMessages(long chatId, long from, long to, int n);
```

Even though it may look tricky, this query is in fact quite trivial since message ids are strictly monotone and adjacent, due to the id generation strategy we imposed and the impossibility to delete messages.

To tell the truth, this API was originally thought to take a time range, making its implementation more complicated using a key-value database. However, during the first design phases we decided to change it in order to make its use more flexible among different database tecnologies.

Another problem with this API was its slowness due to the three operations required to read the message from the database (there are 3 fields to be fetched). This problem was highlighted by the benchmarks of section 3.1. The solution is to aggregate the three attributes in a single structured attribute since they are always fetched together.

1.2.5 Trivial methods

Let's finally list the methods whose implementation is trivial and presents no issues using the proposed structure[1].

```
long addChatMessage(Message message);
long createChat(String name, long adminId, List<Long> userIds);
boolean deleteChat(long chatId);
Chat getChat(long chatId);
long createUser(User user);
long getUserFromSession(String sessionId);
boolean setUserSession(UserSession user);
boolean removeSession(String sessionId);
```

[1] If you're interested in their implementation, you can have a look at the code in the *LevelDBAdapter* class.

1.3 Final design

Key	Value
user:$userId:username	Users.username
user:$userId:password	Users.password
user:$userId:chats	List of chat ids where the user is present
username:$username:userId	Users.userId
session:$sessionId:expiry	Sessions.expiry
session:$sessionId:userId	Sessions.userId
users:nextId	Next id for a user
chat:$chatId:name	Chats.name
chat:$chatId:lastActivity	Chats.lastActivity
chat:$chatId:members	List of chat members
chat:$chatId:admin	Chats.adminId
chat:$chatId:messages:nextId	Next id for a chat
chat:$chatId:message:$messageId	Messages.*
chats:nextId	Next id for a chat

2 Implementation

The Java implementation has been done using *LevelDB*[2]. It is a simple Key-Value database, providing just three methods to access the store: *put*, *get* and *delete*.

Since we included a Database-abstraction layer in our application server, the implementation required just adding a new Java class implementing the *DatabaseAdapter* interface.

3 Testing and Evaluation

The *Key-Value* implementation went under the same testing methods as the JPA one (unit tests and final integration test). Furthermore, since our Java backend provides a convenient HTTP API, we were able to use the open-source software Siege[3] in order to benchmark the different database backends as described in the following section. This was also a way to test the robustness of our implementation.

3.1 Benchmarking

The benchmark was carried out using Siege. We simulated one user sending multiple requests and we measured the average *transactions per second* (tps) for 5000 requests over three runs.

[2]More information at: `https://github.com/google/leveldb`
[3]More information at: `https://www.joedog.org/siege-home/`

Disclaimer The tests were not done in a rigorous way but they were executed just to get an idea on the capabilities of the different database technologies.

Database/Implementation	get chat list	send messages	get messages
SQL/plain	397 (309)	217	13.05
SQL/JPA	278 (225)	136	9.69
LevelDB/no_aggregation	1022 (587)	1050	3.53
LevelDB/aggregated	1033 (527)	950	12.81

In the first column, the value between parenthesis represents the value for the first run, which is the minimum and has been exluded from the average, since it is the only one that didn't take advantage of caching (as the other two runs).

From the results it can be seen that:

- plain SQL is more efficient than JPA.
- Message field aggregation in LevelDB leads to more than 3x better performances.
- LevelDB is much faster than SQL in the first two operations, while shows similar performances in reading many values (as in the last operation).

We also tried increasing the number of concurrent users, obtaining an overall better tps (obviously), but showing the same ratios between different backends.