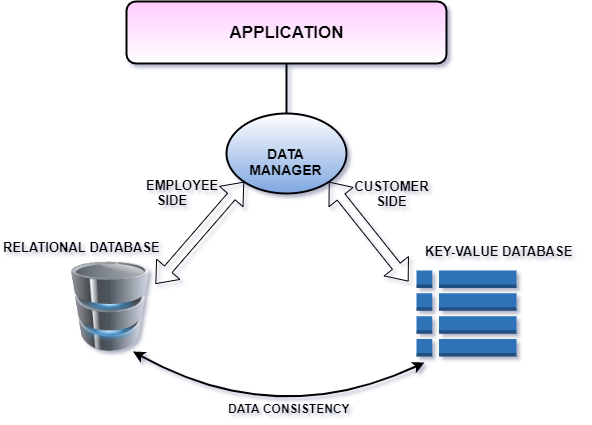
Key-Value model feasibility study

In this chapter we develop a feasibility study in order to decide if the translating of the application database (or a portion of it) into a Key-Value model could be beneficial for the application usage.

Considering the requirements of our application, an object-oriented paradigm for storing data would be generally preferred, since it guaranties data persistence thanks to ACID transactions. A Key-Value model could be implemented to a portion of the database with the idea of speed up some queries (that would require many Join operations between tables), ensuring a faster response time for the application. The response time issue is more critical considering the customer side of the application: a customer should always be able to access to an e-commerce server, to look up his orders and to make new ones.

The following figure describes on a general level the database model that could be implemented:



The idea is to maintain the relational database and insert a new Key-Value database side by side. The Key-Value database manages requests from the customer side, ensuring high response time, while the other requests are been managed by the relation database (using Hibernate, in our application).

In case the Key-Value database should not be available, the relational database will guarantee good functioning of the application: the customer can still access and make new orders. This approach also allow a customer to access and make new orders even when the relational database is not available, as long as the Key-Value database is still operative.

Consistency of the data remains an important requirement for the application, so it needs to be ensured (at least using an eventual consistency approach). Considering the application use cases, we can identify three types of operations:

1. Operations that don’t require consistency between the two databases. These operations have an effect only in the relational database, so they require to be managed only using Hibernate. These are UpdateSalary and all the view operations.
2. Operations that requires “*unilateral consistency*” : these operations cannot be made by a customer so they have to be managed in Hibernate, but still requires an update in the Key-Value database. Delete/Add User and Update Product are the operations that fall in this category.
3. Operations that requires “*bilateral consistency*”. These are all operations that can be made by a customer and have an effect on both the Key-Value database and the relational database. In our application the only operation that falls in this category is when a customer make a new Order.

Cost vs. Revenue analysis

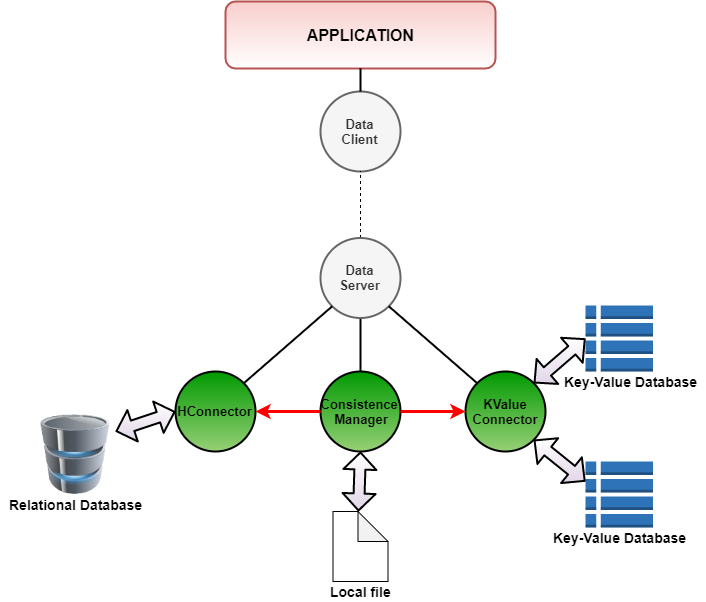
As part of the feasibility study, we analyse the pros and cons of implementing the Key-Value model previously described. On one hand the Key-Value database would ensure high response time for the customer side of the application and allow a customer to access on the application and make new orders. On the other hand, implementing such model would require the application to manage the consistency between the databases.

Our application is been built with the idea to represent a real world application, that could be used in working context, where allowing a customer to always buy new products is a critical priority. Considering also that in our application there is only one operation that requires “bilateral consistency” (Add Order), we decided to implement the Key-Value model described.

Key-Value model implementation

We decide to implement the Key-Value model using a Java port of the LevelDB library:

<https://github.com/dain/leveldb>

The following figure represents in details the architecture of our implementation:  


Three databases are been used: two Key-Value database and a MySQL one. The idea is to manage the Key-Value databases using a LevelDB while the MySQL database is managed using Hibernate. There is also a fourth database in the form of a document database saved in local file: this database is used to store data in gson format by the Consistence Manager module.

We used the hash function SHA1 for the keys and we divided the data in two Key-Value databases.

The data is stored in the key-value databases using the following key model:

* **user : names**  for a list of all the users (sul libro viene chiamata indice?)
* **user : “username”** for password of the user having username = “username”
* **user : “username” : order**  for the user order list
* **user : “username” : order : “idorder”** for the user order having id = “idorder”
* **user : “productName”** for the product having name = “productName”
* **prod : names** for the list of all available products
* **prod : “productName” : idstock** for the list of the stocks available having name = “productName”

All the values in the Key-Value database are gson objects (<https://github.com/google/gson>)

Consistency between databases is implemented through the module **Consistence Manager**. Every time an operation that requires consistency is requested, this module tries to update the other databases: in case it fails, the module saves the operation in an appropriate queue saved on a local file (associated to each correspondent database). Every time new request that requires consistency is made by a user, the module reads the queue and tries to execute the operations needed to guarantee data consistency between the databases. This is mandatory for Write operations, while Read operations can still be applied warning the user that the data shown may not be consistent.

In case of failure, when is not even possible to save the operation in the queue, the application proceedes to undo the operation on the database where the operation was already applied.

Final Application Architecture

