The challenges of key-value stores

Gjorgjina Cenikj
Faculty of Computer Science and Engineering,
Ss. Cyril and Methodius University,
Skopje, Republic of North Macedonia
gjorgjina.cenikj@students.finki.ukim.mk

Dushica Jankovikj
Faculty of Computer Science and Engineering,
Ss. Cyril and Methodius University,
Skopje, Republic of North Macedonia
dushica.jankovikj@students.finki.ukim.mk

Oliver Dimitriov
Faculty of Computer Science and Engineering,
Ss. Cyril and Methodius University,
Skopje, Republic of North Macedonia
oliver.dimitriov@students.finki.ukim.mk

***Abstract*— The escalation of flexibility, scalability, and elasticity demands for data storage solutions has implored the quest of finding an alternative to the traditional relational databases, which in turn, lead to the popularization of NoSQL databases. This paper takes a deeper look into the key-value stores, and accentuates their strengths and weaknesses. The performance of Redis, LevelDB, and Oracle NoSQL is compared to that of the PostgreSQL database. The results affirm the inability of key-value stores to achieve comparable performance on queries that are typically performed on relational databases, such as inserting and deleting records, sorting and querying that involves several join operations. The results indicate that LevelDB outperforms Redis and Oracle NoSQL in most scenarios. The evaluation of the modeling capabilities of each database reveals additional challenges that one needs to be aware of when choosing which key-value store is best suited for their requirements.**

Keywords*— *NoSQL databases, key-value, Redis, LevelDB, Oracle

I. Introduction

Key-value stores are one example of NoSQL databases, which u.e method based on key-value pairs, where each key uniquely identifies the value. Both the keys and the values can be of different data types, from simple strings, to complex BLOBs, depending on the concrete implementation.

The main advantage of key-value databases is their high partitioning and horizontal scaling potential. Due to the fact that most of them do not use a predefined schema, they are considered to be one of the most flexible NoSQL database types, which give the application a complete control over the stored data, with next to no restrictions. Furthermore, since no placeholder values are assigned for the optional attributes, they are more efficient as far as the memory needed to store the data is concerned.

One of the major disadvantages of the typical key-value databases is the inefficiency of querying through the values. The standard implementations do not use a query language, but instead merely provide options for key-value pair addition and removal. Extending the possibilities of manipulating the data using conventional SQL queries imparts the benefits of speed, availability, and scalability, while keeping the familiar relational language.

According to the CAP theorem, it is impossible for a distributed computer system to simultaneously provide consistency, availability and partition tolerance. While relational databases guarantee consistency and availability, the key-value databases guarantee availability and partitioning tolerance, while sacrificing consistency.

Some of the most well known key-value databases are: Dynamo, Hazelcast, LevelDB, Riak, Redis, Oracle, Voldemort и RocksDB. In this project, we take a look at LevelDB, Redis and Oracle NoSQL. We provide a short overview of the capabilities of each dataset in Section 2, and present the most related research in Section 3. A detailed description of the databases, method and proposed models is provided in Section 4, the discussion of the evaluation results are presented in Section 5, while Section 6 concludes the study.

II. Background

A. Redis

Redis is an in-memory data structure project implementing a distributed, in-memory key-value database. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, etc. The basic functionalities it offers are put, get and delete, i.e storing, retrieving and deleting key-value pairs.

Although Redis is an in-memory database, it can persist data on disk. However, the disk storage format is not suitable for querying, as its sole purpose is reconstructing the data in memory, once the system is rebooted.

B. LevelDB

LevelDB is an embeddable, light-weight key value store developed by Google, which offers support for many programming languages such as C++, NodeJS and Java. This database stores the keys and values on-disk in a sorted fashion in the form of byte arrays. The data itself is organised by the core storage architecture which is represented by a log-structured merge tree (LSM). This type of write-optimised storage system makes levelDB appropriate for large sequential (batch) updates instead of sparse random writes.

LevelDB is not a SQL database, signifying it doesn't have a relational data model, nor does it support SQL queries or indexes. The key/value store supports a simple flat mapping from a key to a value, so the creation of data models and data relational handling is considered only in the higher abstraction levels.

LevelDB is published with respect to the New BSD License which ensures that the wider technical community can use this storage engine. This database optimises storage size and bandwidth predominantly with the utilization of compression algorithms such as Google's Snappy and LZ4. The compression showed useful for the research paper since we were dealing with raw text such as JSON and XML. A possible disadvantage of LevelDB is that it requires quite a few disk seeks for information retrieval.

C. Oracle NoSQL

Oracle NoSQL is a distributed database that is typically applied in web applications, where it is used as the primary or as an auxiliary backend database. It supports the majority of the most popular data types. The latest versions provide the option of using tabular structures with an additional level of abstraction that simplifies the modeling and querying with the use of the familiar SQL syntax.

The concept of tables in Oracle NoSQL is to some extent similar to that of SQL tables, in the sense that the tables are composed of rows with predefined, named columns. Each table must have at least one attribute that forms part of the primary key, which uniquely identifies each table row. Some of the methods that refer to multi row operations, allow or even demand the use of partial primary keys, which are keys where the values of some of the key's attributes are not specified.

Contrary to the SQL implementation, each table can have nested, indexable child tables with an unlimited number of rows. The primary key of each child table is composed of their parent's and their own primary keys, meaning that each table implicitly contains the primary keys of all of their ancestors. There are no limitations in regards to the number of child tables present, nor the amount of nesting possible, i.e. the depth of the hierarchy.

Alternatively, lower-level data can be represented with the Record data type, and it is recommended to use this approach in the case of a fixed and low number of attributes. Indexes are an alternative way of querying tables through values that are not necessarily part of the primary key. Using indexes, it is possible to query rows that have different primary keys, but share another common characteristic.

III. Related work

Redis has previously been used as a representative of the key-value family of NoSQL databases, to compare their functionalities to traditional relational databases [1]. While this work has a wider scope, and takes into account all types of non-relational databases, it is primarily concerned with their general capabilities, without going into too much detail about any particular type. Our work is completely focused on the key-value databases, and the difference between their concrete implementations.

Apart from this, Redis' capabilities have also been compared to Riak's, [2] and it has been pointed out that Redis excels in situations where the data is subjected to rapid changes.

A study that bears more similarities to ours, [3] compares the performance of Redis and Oracle NoSQL, and finds that while Oracle NoSQL provides more convenient querying methods, Redis consistently outperforms it in terms of execution time. While this paper also investigates document stores and extensible record stores, ours takes into account one additional key-value database, LevelDB.

IV. Methodology

We setup an instance of each database and consider several different data representation models, in order to be able to fully explore the modeling capabilities offered by the corresponding database. All of the models are based on the same dataset, which is necessary for the comparison of their performance on different types of queries, which is presented in the next section.

A. Dataset

The used dataset refers to completed auctions, where each row contains data about the seller, item, auction, shipping, payment types, buyer protection, and bid history. The original dataset was given in 4 XML files: '321gone', 'ebay', 'ubid' and 'yahoo'. As a preliminary step, each file was converted to JSON format, which is better suited for reading with java libraries. Owing to the fact that all of the fields were originally of type string, additional preprocessing was required before the database insertion took place.

B. Redis

For the purpose of the paper, we made a Java application in order to import the data and run some queries on it. The project setup is straight forward, all that is required is to add a dependency for the Redis Client and make a connection with the Redis-server. In order to import the data in Redis, we transformed the given datasets from xml into key-value pairs, where the keys have the format shown below:

$database_name:$table_name:$primary_key_value:$attribute_name

The key is made of four main parts. The first two parameters are the database and table name. After them comes the primary key, and finally separated with colons are listed the names of the attributes.

C. LevelDB

For the purposes of this research paper the Java implementation of the database was utilised and tested. The api used for this project is part of the project dain/leveldb which offers the core functionalities: get(key), put(key,value), and delete(key).

The entity of interest in this research paper is Listing. For the handling of its nested structure and related attributes, three experimental models were built:

- L_natural model - This model is the most straightforward approach. Here, every complex object within the nested structure of the Listing entity is represented as a separate class. Even though this is an intuitive approach, it might be an overcompatmentalisation of an otherwise rather simple entity with not that many depth levels. The query execution for this model is simple, since no serialisation is included in the nested classes.
- L_flat model - This model is an example of an oversimplification. Here only the leaf nodes of the xml object tree are included as attributes of the Listing model. Even though this increases the accessibility of data for certain keys, some structural information is lost in the process. To an individual familiar with the structure and key characteristics, the performance of this model and the previous one is identical.
- L_formatted model - A model identical to the first one when talking about object structure, but different in the data that it carries. This model ensures input data goes through a formatting phase before being written in the database. This makes the database less error prone and value data types become a universally known fact. The downside here is that some data types are not primitive and might require serialization in order to be written as a byte array value for a key - such as lists, sets, maps.

Aside from the previously mentioned models, some further experimentation was executed by the inclusion of serialisation of objects of deeper levels. What this means is

that the depth of an entity is being decreased by representing the class attributes of non primitive data types as serialised java objects. This process gives a flatter structure, with a major downside - time consuming value extraction which has detrimental effects on performance.

D. Oracle NoSQL

Several models were considered for structuring the data in the OracleNoSQL database. All of the models use the appropriate data type for each field, which required some additional preprocessing effort before the database imports, since the original dataset represented all of the information as strings.

The O_flat model is obtained by moving all of the attributes that would represent leaves in the json tree of the dataset into the root element, listing. This way, the table contains only attributes of simple types, without any further nesting. The column names were created so that they represent the hierarchical organization of the original structure, so that queries can be easily executed if one is familiar with this nomenclature.

The two remaining models, named O_record and O_child, use the Record data type to represent the seller_info, bid_history and item_info fields. They only differ in the representation of the auction_info field.The O_record model relies on the use of the Record data type to represent the auction_info field, while the O_child model represents it using a child table, where all of the attributes are represented using simple data types. The keys of the parent listing table are implicitly added to the auction_info child table, as one would add foreign keys in a relational database.

V. Results

For the purposes of comparison with relational databases, two baseline models were created using PostgreSQL. The P_flat model is created by placing all of the data in a single table, so that no joins are required in order to retrieve the data. The P_nested model involves placing every JSON object in its respective table. This way, the tables item_info, seller_info, high_bidder, bid_history, auction_info and finally listing_nested were created. The P_nested model has foreign keys to every other respective table, which might be redundant since all relations are one-to-one.

The experiments were conducted on machines with 16GB of RAM and i7 processors, with the difference of the machine used for the Redis experiments having 2 cores and 4 threads, as opposed to the 4 cores and 8 threads on the machine used for the LevelDB and Oracle NoSQL experiments. We measured the time needed for importing the data, and executed 8 additional queries that are standard filtering (Q1-Q6) and sorting queries (Q7,Q8) that are typically executed on relational databases. Fig.1 features an overview of the performance of the previously presented models and databases, measured as the time in milliseconds needed to import the data and execute the queries.

TABLE I. Query execution time for each proposed model

Query	Database			
	Oracle NoSQL	***Redis***	***LevelDB***	***PostgreSQL***
Import	O_flat: 3001 + 269 O_record: 2983 + 293 O_child: 5115 + 414	469	L_natural: 94.66 L_flat: 91.29 L_formatted: 103.42	P_flat: 64 + 472 P_nested: 116 + 84
Q1	O_flat: 421^{O}, 30^{J} O_record: 189^{J} O_child: 267^{J}	48	L_natural: 29.04 L_formatted: 39.43	P_flat: 38 P_nested: 37
Q2	O_flat: 404^{O} O_record: 168^{J} O_child: 427^{O}	49	L_natural: 28.35 L_formatted: 35.27	P_flat: 47 P_nested: 32
Q3	all models: 144^{O}, 15^{J}	23	L_natural: 39.3 L_formatted: 28.59	P_flat: 44 P_nested: 31
Q4	O_flat: 441^{O}, 13^{I} O_record, O_child: 22^{J}	150	L_natural: 44.7 L_formatted: 31.23	P_flat: 31 P_nested: 25
Q5	all models: 168^{J}	39	L_natural: 31.24 L_formatted: 25.55	P_flat: 31 P_nested: 33
Q6	O_flat: 396^{O}, 39^{J} O_record: 140^{J} O_child: 391^{O}, 33^{J}	42	L_natural - 26.63 L_formatted: 26.06	P_flat: 36 P_nested: 31
Q7	O_flat: 153^{J}, 274^{I} O_record: 162^{J} O_child: 363^{J}, 630^{I}	144	L_natural: 30.67 L_formatted: 20.2	P_flat: 47 P_nested: 33
Q8	O_flat: 145^{J}, 260^{I} O_record: 20^{J} O_child: 356^{J}, 633^{I}	152	L_natural: 33.88 L_formatted: 21.53	P_flat: 47 P_nested: 32

Fig. 1. Execution times in miliseconds required for the data import and 8 executed queries, for all of the proposed models and databases. The superscripts next to the results of the Oracle models refer to the time required when using the SQL syntax (O), java (J) or previously created indexes (I).

It comes as no surprise that the relational database outperforms the non-relational ones, since key-value stores are not meant to be used when accessing the data through non-key values is required.

A. Redis

In order to get a result for a query, one needs to manipulate the data. In our project, all manipulation and actions with the data in Redis and LevelDB was made using Java code, since the databases themselves provide no operations other than basic retrieval by key, and there is no alternative way to execute the queries.

The average execution time for the queries run on Redis is 80.87ms, the lowest one is 23ms and the highest one is 152ms. It is interesting to point out that while LevelDB and Redis have roughly similar execution times for the select queries, Redis is somewhat slower when it comes to the database creation and sorting queries.

B. LevelDB

Out of the three databases explored in this paper, LevelDB has the best performance on most of the queries, and manages to outperform the baseline PostgreSQL on the sorting queries and some of the filtering queries.

As far as the different proposed models are concerned, it can be noticed that the L_natural model that does no data

preprocessing executes the database population faster than the L_formatted model, because no time is spent on serialization. On account of this, the L_formatted model performs better on the data retrieval queries, because there is no need for formatting the data on the fly.

C. Oracle NoSQL

Oracle NoSQL does support the execution of some of the queries, and therefore, its results are accompanied by a superscript, which indicates the type of approach that the specified execution time refers to.

When analyzing the time needed to import the data, one can clearly note the significantly higher amount of time required by the O_child model. This is due to the fact that, unlike the other models, O_child requires the creation of an additional table for representing the auction_info.

A slightly less obvious observation is the lack of a result referring to the performance of the O_record model using a secondary index. The absence is a consequence of the inability to create indices based on the values encapsulated in a Record field, and it is one of the main disadvantages of this model. A related issue is the inability to query through these values, which is the reason why queries Q1, Q2, Q4, and Q6 had to be executed using java code.

The O_flat model supports the execution of all of the queries using Oracle's SQL syntax, but this model sacrifices the original data structure.

As far as execution time is concerned, the O_child model seems to yield the most inferior performance out of the 3 Oracle models. However, this model would be more memory efficient in the case of missing auction_info values, since both of the other models would store null values for all of its fields, while the O_child would waste no memory resources. This is also the only model that could represent a one-to-many relationship, if needed.

While it is rather obvious that the Table API used for the Oracle NoSQL database is slower than the other two databases, it is worth noting that the syntax the API provides is significantly more compact and convinient to anyone familiar with SQL.

VI. Conclusions

Through the review of Redis, LevelDB and Oracle NoSQL as representatives of the key-value databases, this work once again reaffirms the strengths, and especially the weaknesses of this type of non-relational databases. The obtained results indicate that the Table API used for the Oracle NoSQL database is slower than the Redis and LevelDB implementations. While relational databases outperformed on the dataset used in our study, the empirical results have noteworthy implications for future research that could involve the use of a different and larger dataset, analyzing additional queries and experimenting with different data models.

References

[1] M. Radoev, "A Comparison between Characteristics of NoSQL Databases and Traditional Databases," Computer Science and Information Technology 5.5, 2017, pp.149 - 153.

[2] C.A. Baron, "NoSQL Key-Value DBs Riak and Redis", Database Systems Journal, 6, issue 4, 2015, pp. 3-10.

[3] F. Bugiotti and L. Cabibbo, "A comparison of data models and APIs of NoSQL datastores," 21st Italian Symposium on Advanced Database Systems, 2013, pp. 63-74.