Otto-von-Guericke-Universität Magdeburg

Faculty of Computer Science

Master's Thesis

Exploring Multi-Model Features of Redis

Author:

Aeman Nawaz Malik

July 18, 2018

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. Gabriel Campero Durand

Databases and Software Engineering Workgroup,
University of Magdeburg

Malik, Aeman Nawaz:
Exploring Multi-Model Features of Redis
Master's Thesis, Otto-von-Guericke-Universität Magdeburg, 2018.

Abstract

The world of data has seen a dramatic increase in diversity and volume of everyday data in recent years. This has led to emergence of various technologies to handle and manage this data. Different SQL and NoSQL databases are available to cater to a certain kind of data model, however, in complex applications, there is not only one kind of data to be managed. The concept of Multi-model databases addresses the needs of such applications where data belonging to different models is being managed at the same time. In this work, the multi-model features of Redis, a popular key-value store, are explored and evaluated. The two modules of Redis used in this work are ReJSON (document store) and Redis Graph (graph database). The core tasks of data ingestion, data transformation and data querying are assessed and performance of each data model is recorded in each case. We find that data load is done more efficient in Redis, and that a CRUD workload as embodied in the YCSB database can be best served by Redis. Redis Graph is shown to perform inefficiently for loading data, when compared to the other modules. However, when the workload contains short scans, we note that Redis Graph performs the best. We find that the cost of moving data across models can be at least twice as high as that of loading the data originally into Redis, even with the use of scripting features. We also observe that scripting overall leads to performance gains, giving Redis Graph a small competitive edge over ReJSON. We also note that work is required to understand the reasons for the benefits of scripting, such that this could be replicated in other operations with the modules (e.g to improve their data load). Finally, we compare the performance of the modules for more analytical queries. We use queries from the ArangoDB benchmark. We find that overall Redis Graph performs the best, stemming from its ability to express the queries in OpenCypher, its native graph query language, and from its storage structure that matches well the workload. In future work we seek to study cases that fit the performance characteristics of ReJSON.

This work assesses the ability of Redis to perform as a multi-model database and provides a guideline for steps required to be evaluated in the creation of a multi-model database (i.e., data load, data transformation and data querying). In future work we seek to contribute in the standardization of these steps, such that other multi-model databases could be evaluated, and such that the data lifecycle of working with these systems could be better understood and designed, helping users exploit the protean features that multi-model databases offer.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, M.Sc. Gabriel Campero Durand, for his continuous support, patience and immense guidance at every step. His untiring efforts to provide encouragement and moral support at every step have played a vital role in the success of this Thesis. I also thank Prof. Dr. rer. nat. habil. Gunter Saake for providing me with the opportunity to write my Master's thesis at his chair. I am very grateful to my husband for supporting me in every way during this journey and to my family and friends for being there when I needed their support. Last but not the least, I am thankful to my baby daughter, Sarah, who made this Thesis a roller coaster ride. I thank her for her patience and love that kept me going.

Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.
Magdeburg, July 18th 2018

_____

Aeman Malik

Contents

List of Figures

List of Tables

Listings

1. INTRODUCTION

Over the years, there have been significant advancements in the storage, communication and processing of data. Since the advent of the world wide web, the amount of data available and being generated online has increased continuously at an ever growing pace. In recent years, the three characteristics of such fast growing data have been described under the concept of *big data*; velocity, veracity and variety. This versatility of modern data in turn has given rise to different methods and data management tools to store and work with big data.

The concept of data storage goes back to even before the invention of computers. Since the 1970s, with the invention of databases, application-specific hand-crafted storage and management of data (e.g. transaction records and customer data) was replaced and made easier and more efficient. In fact, databases nowadays play an important role, sitting between operating systems and traditional applications, in helping developers build applications without being troubled by the complex requirements of efficient and consistent data management.

Today's data generated by real-time applications, interactive web applications and social networking websites is not only large in amount but is also unstructured, posing challenges to the traditional designs for data management solutions. Relational database management systems were and still are a suitable option for storing structured data. But the need to store this unstructured data of various kinds (documents, graphs, time-series, etc) has led to the rise of so-called NoSQL databases (an acronym which stands for not-only relational). These databases provide organizations with a chance to store separately different kinds of application data according to the required data models.

Building on the mainstream adoption of data models alternative to the relational model (e.g. JSON formats, graphs of different kinds, columnar-structured large-scale storage as exemplified by Cassandra), a large variety of storage systems have been evolved to suit the needs of different data models. Non-relational database systems per-design seek to provide developers with more flexibility, when compared to their relational counterparts.

The data generated by the WWW today is highly versatile and unstructured. In order to store, manipulate and access this kind of data, a flexible, schema-less alternative to RDBMS is required, and NoSQL databases provide such alternative. In spite of these benefits, when it comes to choosing a suitable database for an organizations data, there is no one-size-fits-all solution. Relational models can and do work well when the data is structured and the workload matches what RDBMSs offer. When data is unstructured and the workload has other requirements, then database systems supporting other data models could be an alternative. Practitioners have coined the possibility of doing so, as *polyglot persistence* [6].

So organizations and businesses nowadays can rely on both relational and non-relational storage systems for different kinds of data. For example, relational database systems are used in cases where the data to be stored is structured, requires consistency guarantees to be satisfied and does not need to be horizontally scaled (as RDBMs ensure ACID guarantees but have, in general, poor horizontal scalability [7]). In financial and e-commerce applications, it is preferred to use a relational database that fulfills ACID guarantees. In cases where the data to be stored has little to no structure, a suitable NoSQL storage system could be preferred. Also, NoSQL databases could be used in cases where consistency can be neglected to an extent in favour of high speed and performance.

In today's age of big data, modern enterprises contain large applications having various parts, each addressing a different kind of problem. These applications deal with lots of different data and hence could require both relational and non-relational databases according to the suitable data model. Every part of the system will use a different database that best suits its needs. Fowler and Sadalage [6], coined the term "Polyglot Persistence" to describe this approach. In such a system, for example, a relational database will be used for structured tabular data, a document database for unstructured data, a key/value data store for a distributed hash table which could be used for application caching, and for linked referential data a graph database. So, at the end, a federation of databases can be used within the same project. This gives rise to deployment and operational problems.

The assurance of an application using a diversity of databases to be fault tolerant is very challenging. Moreover, keeping such a system synchronized and consistent is also difficult when there is data duplication across databases. Furthermore there can be complexities in developing applications with the awareness of a large amount of underlying systems.

An alternative solution to dealing with multiple types of data distributed across several systems, is to include them in a single system. This is a solution provided by multi-model databases. Multi-model databases address the very issues posed by the approach of polyglot persistence. Duggan et al. study such approach in the BigDawg polystore system[8].

The concept of multi-model databases, as discussed by Lu and Holubova[4], can be explained briefly as an approach to incorporate multiple databases in a system using a single engine as the back-end (mind that this still might involve separate storage

sub-systems), and giving users the impression of a single system, thus minimizing developmental and operational complexities. These databases should ideally have one query language (though possibly with different dialects) and a single API that can be used on all data models. In today's age of big data, enterprises which prefer to adopt multi-model databases over federated polyglot systems enjoy a number of benefits, some of which are described as follows (based on the presentation of Lu and Holubova[4]):

1. Less operational complexity:

   In an environment where multiple databases are being used separately by an application, the complexity of the development process increases due to the requirement of several query languages, and explicit mappings between models. From the operational side there are also challenges, such as the need to provision for mixed workloads without a clear separation of requirements for each back-end. Furthermore, maintaining fault tolerance and data consistency can be challenging as well. Multi-model databases, on the other hand, reduce these operational challenges by managing data consistency and offering a single operational component for simpler provisioning.

2. More Reliability:

   In a polyglot approach adopted by large applications, if a single database fails, the whole system needs to stop and restore its functionality. This may require downtime, and expensive synchronization among databases. For some scenarios such costs are unfeasible (e.g. Web services might not be able to tolerate bad user experience). Multi-model databases are, at least from a conceptual perspective, more fault tolerant since the varied data stores are incorporated into a single database engine and hence experience less coordination issues.

3. Less costs:

   Using a number of databases, as in the polyglot approach, increases hardware, software and management costs for an application. Each database requires individual maintenance. Updates in each component require organizations to make appropriate updates and furthermore, to hire specialists of all the different databases. Multi-model databases, being a single synchronized system, could face less operational costs comparatively.

These and many other benefits of a multi-model database, which make them significant in today's era of diverse and voluminous data. Different databases today are presenting themselves as multi-model but there is not much information and understanding on how to use them as such. Specifically, there are scarce studies evaluating comparatively the cost of operations over different data models within a single multi-model database, therefore the life-cycle of data (including the migration across available data models to better serve workloads) is still poorly understood. As a consequence developers have no guideline nor best practices to manage their data across these models, when possible.

Similarly, the absence of such studies limits the development of multi-model storage engines capable of migrating between models for improving performance for workloads (i.e., using a data model as an index to speed-up the processing of some operations of another model).

Realizing such study is not specifically difficult, and we believe that it has not been done before solely because of the slow development of standards and available features for multi-model databases. In this work we exploit the fact that there are some early multi-model offerings in off-the-shelf systems, enabling us to evaluate the essential performance differences between model support in such systems.

1.1 Research Purpose

In this work we evaluate a multi-model database using a centralized storage. For this purpose we adopt Redis, an in-memory key-value data store with support for additional data models via pluggable features. Redis is explored and extended by the use of community-supported modules, in order to study its multi-model features. Redis, as mentioned, is basically a key-value store that contains various modules (explained in more detail in Chapter 2) which enhance its storage capabilities and give it a chance to be presented as a multi-model database. From a number of modules that it provides, in this work, two of them namely ReJSON (for documents) and Redis Graph (for graph data) are used. Thus, the data models targeted in this work are key/value data, documents and graphs. Ingestion of data into one model, conversion of data between models and querying of data has been done using queries from the ArangoDB benchmark (explained in more detail in Chapter 3). Performance for data ingestion and transformation has also been measured using the Yahoo! Cloud Serving Benchmark (YCSB), an OLTP benchmark. Hence, the general research questions that this work is founded on are:

1. What are the options to realize a multi-model database using a centralized storage?
2. How good is each model in performing the core tasks that a multi-model database requires (data ingestion, data transformation, and data querying)?

1.2 Research Methodology

To provide more structure to our research, we have followed a standard research methodology namely, the Waterfall Model [1]. This model is a sequential process model to carry out a research. The linearity of this model makes it simple and hence, easy to adopt. This model is shown in Figure.

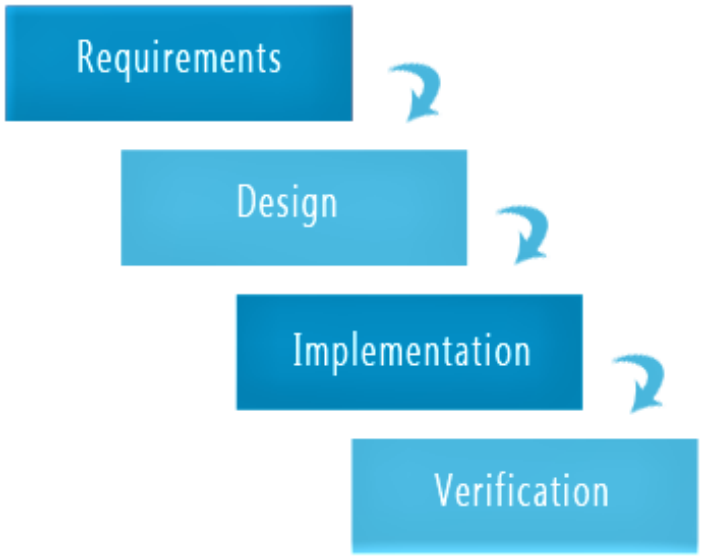


Figure 1.1: The Waterfall model for structured research

The steps followed in this model are explained briefly, as follows:

1. **Requirements Analysis:** In this phase, the requirements for the application are observed and analyzed. This phase is very important at the beginning of the research process, since the tools needed for the research and other necessities are noted down at this stage.

2. **Design:** After the requirements for the research are noted down, the implementation design is carried out. Here specific strategical questions need to be addressed. This design is a core foundation of the future implementation of the project.

3. **Implementation**: This is the development phase which consists of coding part the prototypical implementation. This is the stage where the product is actually being created.

4. **Testing and Verification**: At this stage, the research work done so far is tested and verified, its validity is checked and modifications are suggested if necessary. Apart from the validity checks, this stage encompasses the intended evaluation that is the focus of our work (i.e., from research question 2.).

[1] https://www.jitbm.com/Volume2No1/waterfall.pdf

We have used two iterations of the Waterfall model methodology, each for our two set of tests in this work.

1.3 Structure of Thesis

We structure this thesis as follows:

- Chapter 2 contains the method used to collect literature for this work. Also, the technical background containing a brief history of databases. and the problems that led to the rise of multi-model databases are explained.
- Chapter 3 consists of implementation techniques and details of how this research was carried out.
- Chapter 4 and Chapter 5 collect the results and our discussion for the specific tests carried out in this work.
- Chapter 6 discusses closely related works.
- Chapter 7 concludes the work done, describes some threats to validity and highlights areas in which work can be done in future.

2. TECHNICAL BACKGROUND

In this chapter, a brief history of databases is presented. The evolution of databases from relational to NoSQL is explained and the reasons that led to the emergence of multi-model databases are highlighted. Different relevant database models are also explained with examples.

- Section 2.1 presents the literature research methodology for this work
- Section 2.2 presents relevant history of databases and background of SQL and NoSQL databases with examples
- Section 2.3 highlights the key points of the Polyglot Persistence Concept
- Section 2.4 describes multi-model databases and provides their categorization
- Section 2.5 presents the challenges that are faced while designing a multi-model database
- Section 2.6 concludes the chapter

2.1 Requirements Analysis - The First Step

The first step of the waterfall model followed in this work is to analyze the requirements of the project. We have observed these requirements through comprehensive literature research and by scanning relevant technical background. The literature research and the study of the relevant background is given in the sections below.

2.1.1 Literature Research

In this section, we present an overview of the literature collection process for this research.

2.1.1.1 Research Phases

The collection of relevant literature is divided into two phases:

- **Initial Search Phase:** In this phase of collecting literature, the focus was on retrieving articles that were relevant to multi-model databases and Redis database in general. This selection was done based mostly on the primary search terms (discussed below). Detailed study of the papers, to confirm relevance with the topic, was not made in this phase.

  Primary Search Terms: The primary search terms used and the number of literature sources retrieved are given below:

  - NoSQL and multi-model databases: 57 sources retrieved
  - Redis and multi-model databases: 33 sources retrieved and 5 selected out of them
  - Multi-model and polyglot database: 4 sources retrieved

  In total, 94 sources (including journal articles, conference papers and books) were initially selected.

- **Refined Search Phase:** In this phase, exclusion of some literature sources collected in the initial phase was made. This exclusion was made according to a criteria.

  Exclusion Criteria: Bachelor and master thesis, unpublished works and incomplete works were discarded. The search terms were enhanced according to detailed study of the acquired resources. These refined search terms led to more relevant articles. Also, by detailed study of the sources collected initially, and through examination of their bibliography, 35 most relevant literature sources were chosen.

2.1.1.2 Scope

The scope defines the databases of literature available on the Internet that were chosen to select relevant articles. The database selected for the research is Google Scholar. The time period of the obtained literature lies between 1967 to 2018, with most articles being from 2011 onwards and only two literature sources from 1967.

2.2 A Brief History of Databases

From the earliest days of computers, a main focus of any application has been storage and manipulation of data. The history of databases is a tale of attempting to address the ever increasing complexity of everyday data and its storage requirements. From the time when data was simple and could be managed manually, to today's era of data deluge where every day 2.5 Quintillion bytes of data are created [9], systems and machines

have evolved immensely according to the varying data management needs. Today's era of information explosion brought with it a dire need to structure this information. For this purpose various data management systems were invented at different points in time. Index card can be considered as the predecessor of computerized databases. The process of card indexing was invented by Carl Linnaeus in around 1760 [10]. This method of putting down data on cards worked at the time but as information load increased this process, which had to be handled by a man, could not last. Around 1880, Herman Hollerith transformed the world of data storage by inventing the first electromechanical machine for storing data [11] . This machine used punch cards to store information and data was represented by holes on the punch cards. During that time, Hollerith's firm, along with three other companies, merged into what is now called International Business Machines (IBM) [11] . From then on, electromechanical data storage and retrieval continued for the next fifty years. Around 1950, IBM created a new machine called UNICAV I, which holds the special position of being the first digital computer for commercial utilization. It, however, soon proved to be inefficient for data storage tasks. A historical Conference on Data Systems Languages (CODASYL) [12] took place in 1959 which founded the concept of introducing a common computer programming language. This programming language was created and defined in the same year as Common Business-Oriented Language (COBOL) [13]. With the advent of programming languages, came the transformation from magnetic tapes to magnetic disks; thus allowing direct access to stored data. This evolution laid the foundation of the modern database management systems (DBMS) of today. The advent of computerized databases brought huge amount of ease to the programmers and developers as they did not have to worry about the arduous task of storage as this was handled by the DBMS. The database management systems in that period (1965-70) were categorized into two basic models: network model and hierarchical model. Both of these methods to store data had some restrictions and drawbacks. This was observed by Edgar F. Codd, who introduced the concept of a relational model in 1970. IBM developed a prototype relational database around 1974 called System R. System R adopted the structured query language (SQL) which was developed in 1980s. The world of data has been ruled by relational database management systems for more than 4 decades.

2.2.1 Relational Model

The relational model allows storage of data in the form of *relations* or tables. Relations, in turn, represent a set of tuples and attributes. A specific type or domain of values can be associated with each attribute. Examples of the database systems following relational model include MySQL, POSTGRESQL and Oracle.

2.2.1.1 Data Layout

A relation, in a relational model, consists of rows and columns. Records are represented in rows and attributes in columns. A tuple consists of a single row of a relation. One column contains the set of values for a specific attribute. Each row of the table can be uniquely identified by one or more keys or identifier attributes. Moreover, values which are not given for an attribute are termed as null values.

2.2.1.2 Integrity Constraints

In order to prove itself valid, a relation must satisfy some constraints and fulfill some rules. Firstly, the presence of one or more such attributes through which a tuple can be uniquely identified is mandatory in a relation. Furthermore, these key attributes cannot have identical values for any two tuples; thus making each tuple unique. These key attributes cannot be null or empty. Some attributes can have only certain specific values in real world; for example, age cannot be less than zero. The constraints of the real world are also applied in the values of any attribute in a relation. Referential integrity constraints are applied when foreign keys are used. Foreign keys are the attributes that connect relations with each other by being referred in some other relation. Referential integrity constraint ensures the presence of the relation whose key value is being referred.

2.2.1.3 MySQL - An Example

MySQL is an open-source relational database management system. It is very popular owing to its cost-effective delivery of high performance and secure (ACID compliant) applications.

Commands

Here is the the list of some of the basic operations provided by SQL.

SELECT - Used to select and retrieve data from a database.
UPDATE - Used to update data in a database.
DELETE - Used to delete data from a database.
INSERT INTO - Used to insert data into a database.
CREATE DATABASE - Used to create a new database.
ALTER DATABASE - Used to modify a database.
CREATE TABLE - Used to create a new table.

2.2.2 The Rise of NoSQL Databases

With the passage of time, there have been considerable changes in data language, processing, architecture and platforms. Applications and enterprises started using multiple technologies for management of their data. Development of versatile applications also led to handling of various kinds of data. This versatility of data and its handling gave rise to various new data models. It has been, thus, made clear in recent years that although relational databases are still a useful and powerful tool that we expect to be using for many more decades [6], but a profound change in types of data shows that relational databases won't be the only databases in use. The relational model followed in RDBMSs ensures ACID (Atomicity, Consistency, Isolation, Durability) guarantees but does not provide the horizontal scalability and the availability required by applications and enterprises dealing with large volumes of versatile data. To address these requirements, a NoSQL movement started in mid 2000's. The term NoSQL refers

to various databases that do not follow the fixed relational model but offer different models according to the type of data. These databases fore-go the strong transactional guarantees provided by the relational model in favor of eventual consistency and improved horizontal scalability.

NoSQL databases can be divided into four major categories based on their data model: key-value, column family, document and graph databases.

2.2.2.1 Key-value Stores

A key-value store is a system that stores data in the form of key-value pairs. Data is stored as values with specific keys to find and access them. A key-value pair is considered a single record in the database. Various NoSQL systems follow the structure of key-value stores but they all have different methods to define keys and values and also differ greatly in their allowed operations and ways to access the key-value pairs. In a key-value storage model, arbitrary key-value pairs can be stored without any restriction or regulation pertaining to the format of the value. The main feature of the key-value store is that it is simple and quick. A main benefit of the simple format of such data storage model is it makes easier for data to be distributed among different database servers. So, key-value stores are beneficial for applications that are "data-intensive". Some examples of systems following this data model are Apache Cassandra, Redis, Oracle NoSQL and Aerospike.

2.2.2.1.1 Data Layout

In a key-value store, a key-value pair refers to a tuple of two strings (key, value). This pair is stored as a hash table with two columns, key and value. The key may be represented by an arbitrary string such as a filename, a URL or a hash. The value can be any kind of data such as a document, an image or a string.

2.2.2.1.2 Integrity Constraints

In general, key-value stores maintain integrity constraints on the stored data by applying a hash technique. This technique makes sure that all the keys stored in the database are unique and no two keys are similar.

2.2.2.1.3 Redis - An Example

Redis is a key-value in-memory database. The name Redis stands for REmote Dictionary Server. It was initially released in 2009 by Salvatore Sanfilippo [14]. Data structures such as strings, lists, sets, hashes, sorted sets and indexes are supported by Redis. Almost all popular languages provide Redis bindings. Some of them are C, C++, Common Lisp, Dart, PHP, Erlang, Java and JavaScript (Node.js). Other features provided by Redis are in-built replication, Lua scripting and transactions. Redis keys are binary

safe, which means that any binary sequence can be used as a key in Redis. A string, an image, even an empty key can act as Redis key. Redis string is the simplest data type that can be linked as a value with a Redis key. Unlike general key-value stores, Redis supports different data types as values, not just strings. Due to its in-memory nature, it speeds up web applications and is a preferred choice when instant results are required.

Operations:

Various commands are used to perform operations on Redis server. The command line interface of Redis, Redis-cli, is used to send commands to Redis and receive replies sent by server. There are various commands in Redis to manage keys and values. Some basic commands are SET (to set a key with a value), GET (to return the value associated with a given key), APPEND (to append a value at the end of the string) and EXISTS (to determine if a key exists). Redis commands will be explained in detail in Chapter 3. An example of SET and GET command using Redis-cli is given in Figure 2.1 .

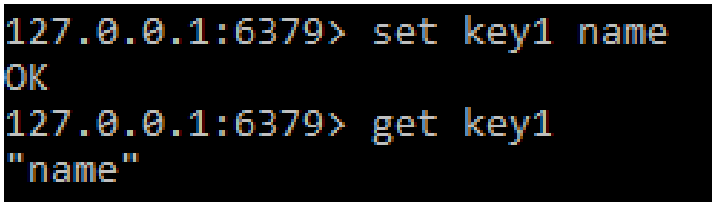


Figure 2.1: Example of SET and GET command in Redis-cli

2.2.2.2 Document Stores

Document stores allow data or documents to be stored in a semi-structured manner. A document is mainly a complex value which can further nest other documents, lists or scalar values. Document stores are similar to key-value stores in the fact that they also store data with a unique key. But unlike in key-value stores, in document stores the *value* is not just a random string but a structured document following a certain text format.

2.2.2.2.1 Data Layout

As the name suggests, document-oriented data stores store data as semi-structured documents, although different databases follow somewhat different structure from each other. Data encodings generally used by document stores include XML, JSON and BSON. A document store also supports multiple types of documents. Examples of documents encoded in JSON and XML formats respectively are shown in Figure 2.2 and Figure 2.3.

```
{
    "FirstName": "Bob",
    "Address": "5 Oak St.",
    "Hobby": "sailing"
}
```

Figure 2.2: Example of JSON encoding

```
<contact>
  <firstname>Bob</firstname>
  <lastname>Smith</lastname>
  <phone type="Cell">(123) 555-0178</phone>
  <phone type="Work">(890) 555-0133</phone>
  <address>
    <type>Home</type>
    <street1>123 Back St.</street1>
    <city>Boys</city>
    <state>AR</state>
    <zip>32225</zip>
    <country>US</country>
  </address>
</contact>
```

Figure 2.3: Example of XML encoding

2.2.2.2.2 Integrity Constraints

Integrity constraints are generally included in the schema specification. As document-oriented databases are schema less to a great extent, integrity constraints are not enforced directly. Thus, integrity constraints in a document store may be applied at application level or through the usage of indexes (a unique index enforces a unique constraint) [15]. In XML documents, Document Type Definitions (DTDs) offer some attributes that are used to apply integrity constraints to some extent but they do not cover all semantic constraints.

2.2.2.2.3 MongoDB - An Example

To explore the various operations allowed by document stores, an example of a database that belongs to such a category of storage models, MongoDB, is discussed. MongoDB [1]

[1] https://www.mongodb.com/

is a data storage system designed to store documents. In this data store, a database is made up of collections. Documents are grouped into these collections. A document stored in MongoDB is not arbitrary but is structured and can further comprise of other documents or lists. The documents stored in MongoDB follow BSON format and are basically a collection of key-value pairs.

Operations:

MongoDB provides a command line interface named as mongo shell. The main operations that this data store provides are stated as follows:

db.collection.insert(): To add a new document into a collection
insert(coll,doc): To add a main document in a collection
find(): This method returns all documents in a given collection when parameter list is kept empty. A "selector" can also be given in the parenthesis to find particular documents
remove(coll, selector): To remove from a given collection, all documents that match the given selector
update(): To modify the existing data in a document or to replace a given document with a new one.
Furthermore, MongoDB also provides many options to aggregate existing documents. Also, it allows for references to be made between documents.

2.2.2.3 Extensible Record Stores

Extensible record stores allow data to be stored in the form of rows and columns. Although the basic data structure of extensible record stores are tables but they allow a high degree of flexibility in column management. This type of data storage systems are also known as column-family stores, as the concept of column families is implemented in them, which refers to containers comprising of subsets of columns. Other names given to extensible record stores are tabular data stores, columnar data stores or wide column stores. Some popular NoSQL databases following the column-store model are Google's BigTable, Facebook's Cassandra and Amazon DynamoDB.

2.2.2.3.1 Data Layout

Extensible record stores take the concept of storing data in tables just like in relational data model and are similar to key-value stores as well in a way because they also map an id or a key to each value. A column store consists of a collection of rows, where each row can contain any number of columns. An example showing how a row is constructed in a column store is given in Figure 2.4

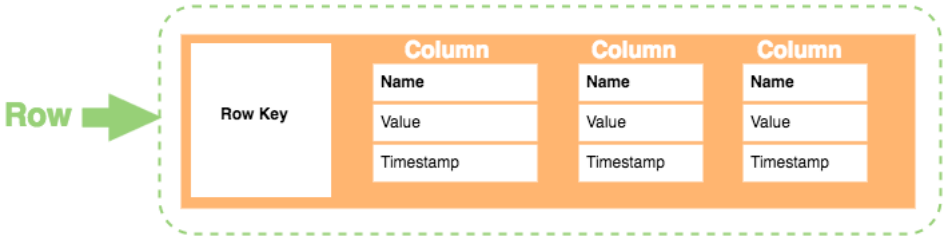


Figure 2.4: Column store

2.2.2.3.2 Integrity Constraints

Integrity of data is ensured in column family stores by assignment of a unique key to each table. Other integrity constraints provided by relational database managements systems are not ensured in column family stores. [15]

2.2.2.3.3 Google's BigTable - An Example

BigTable is a distributed storage system, following the column store data model, created by Google.Inc. BigTable is used to manage structured data and is aimed to provide high scalability, availability and high performance [16]. A BigTable basically represents a sparse, multidimensional, sorted map which is indexed by a row key, a column key and a timestamp. The information contained in this map is thus presented in three dimensions; row, column, timestamp. Timestamps maintain versions of the stored data.

2.2.2.4 Graph Databases

Graph databases present data in the form of entities and their relationships. These databases prove to be significantly helpful when relationships between different entities needs to be highlighted. An example of their application can be found in a problem domain similar to a social network. The focus of graph databases on the visual representation of data items and their relationships makes them more human-friendly compared to other NoSQL databases [17].

2.2.2.4.1 Data Layout

Graph databases represent data in form of entities and their relationships. Entities are represented by nodes and the relationships between entities are shown by edges. This model contains vertices (to represent entities) and edges (to represent relationships between entities).

```
START person=node:people(id = {id})
MATCH person-[:FRIEND_OF]->friend-[:FRIEND_OF]
    ->friend_of_friend
WHERE not (friend_of_friend<-[:FRIEND_OF]-person)
RETURN friend_of_friend, COUNT(*)
ORDER BY COUNT(*) DESC
```

Figure 2.5: Example of a Neo4j query to suggest friends for a person

2.2.2.4.2 Integrity Constraints

To ensure consistency of data, some integrity constraints are applied in graph databases. These constraints include labels having unique names, various typing constraints on nodes [18], functional dependencies [19], domain and range restrictions on properties of nodes and edges [20].

2.2.2.4.3 Neo4j - An Example

Neo4j is one of the most popular NoSQL graph databases. It is written in Java and Scala and became available for public use in 2007.

Operations:

Neo4j's query language is Cypher. Cypher queries, in ideal cases, are just like constant strings and hence databases cache them as compiled strings [21]. Parameters in these queries can be numbers, arrays or strings depending on what type of query is being run. Cypher, just like SQL, is not only a querying language but also manipulative i.e. it also provides update and delete options. An example of a query run in Cypher for retrieving friend suggestions for a person is shown in Figure 2.5.

A summary of relational and NoSQL databases is given in Table 2.1.

We have, so far, discussed different data models, example databases and the type of data they store. However, in the world of today, applications and businesses have become so complex that the usage of one data model to cater for all the diverse data requirements has become nearly impossible. This has led to the concept of multiple data models in a single application, each catering to a specific kind of data. Owing to this need, the emergence of the polyglot persistence approach, discussed in Section 2.2, took place.

2.3 The Polyglot Persistence Concept

Each NoSQL database is suited for a certain kind of data model, making them a good solution for some problems and bad for the others. For instance, key-value stores are preferred to be used when storing data about frequently updated page visits, document stores are suitable for storage of unstructured data e.g. user data and graph databases are a suitable option to store relationships between different users on a social media

Database	Data Layout	Integrity Constraints	Example Commands	Example database
Relational database	Rows and columns, fixed schema	Unique key attributes to avoid duplication. Referential integrity constraints	Select, Update, Delete, Insert into	MySQL, Oracle and IBM DB2
Key-value store	Tuple of two strings, key and value	A hash technique makes sure that all keys are unique	store.put(key, value) value=store.get(key) store.delete(key)	Redis, Oracle NoSQL Database and Aerospike
Extensible Record store	Rows and columns. Collection of rows called containers Containers comprise of subset of columns	Unique keys and identifiers to each table	Examples from DynamoDB: create-table, list-tables, describe-table, get-item	DynamoDB, Google's BigTable and HBase
Document Store	Semi-structured documents; formats include XML, JSON	Applied at application level and by use of indexes. DTD's offer attributes to apply integrity constraints	Examples from MongoDB: db.collection.insert(), update(), find()	MongoDB, CouchDB and OrientDB
Graph Store	Entities and relationships (vertices and edges)	Domain and range restrictions on nodes and edges' properties, labels with unique names, functional dependencies	Match-Where-Return, Order by	Neo4j, JanusGraph and AgensGraph

Table 2.1: A summary of relational and NoSQL databases.

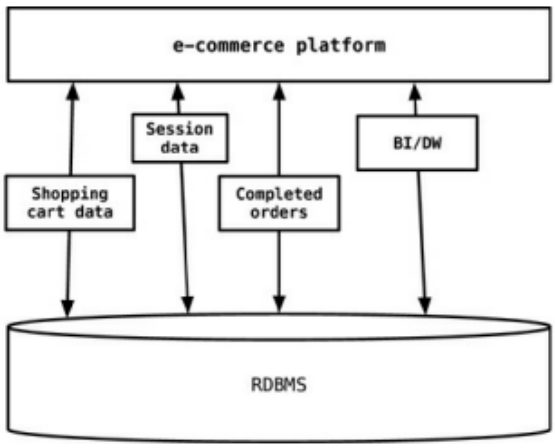


Figure 2.6: Use of RDBMS for every aspect of storage in an application

website (Transformations on Graph Databases for Polyglot Persistence with NotaQL Johannes Schildgen1 , Yannick Krück2 , Stefan Deßloch3) Besides having different data models, all NoSQL databases have a different support for transactions and distribution, and different benefits and drawbacks. (ref) Using a single database for varying problems and requirements leads to poor, inefficient solutions [6]. For example, in Figure 2.6, an e-commerce platform is using a single database, RDBMS, for handling all different kinds of issues.

Though session data, orders data and shopping cart data require different scales of consistency and availability requirements and orders data needs more strong backup/recovery strategy compared to session data, choosing a single database to handle storage of all these issues leads to wastage of resources and time.

Large systems and enterprises of today are usually concurrently working on various problems and have huge volumes of versatile data. For such systems, it becomes difficult to choose one database as this does not suit the different types of data being tossed around in the system. This difficulty of choosing one database for different problems led to the emergence of polyglot persistence concept [6]. This concept suggests usage of more than one databases against a single backend. Every part of the system can use the database that suits its requirements. So, where structured data is being worked on, relational database can be used there and the part of the system that deals with unstructured document-like data can use a document store. Taking further the example [6] shown in Figure 2.6, a better solution would be to choose key-value store for shopping cart and session data. As shopping cart data is accessed by user ID of the customer, usage of key-value store is a better option here. Once the order is confirmed by the customer, that data can be stored in RDBMS, thus avoiding storage of transient data in RDBMS by using key-value store for that purpose. Also, a graph store handles recommendations of products to other customers. This solution is shown in Figure 2.7.

In order to implement an application following polyglot persistence approach, along with frameworks and APIs, languages and platforms are also required that are able

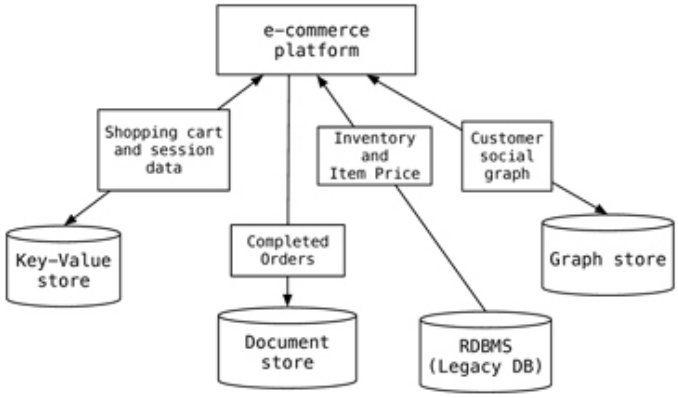


Figure 2.7: Example implementation of polyglot persistence

to transform and move data from one data model to another (Schildgen2017). This interoperability between data stores is carried through using a language that supports a variety of data stores and their distinct data models (Schildgen2017).

2.3.1 Limitations

Realization of such a system using more than one NoSQL database leads to a number of deployment and management issues. An abstraction and integration layer is required to be formed between all databases that are part of the application and this imposes operational and technical issues [15]. To make such a system fault-tolerant is an extremely challenging task [6] and many factors have to be considered for its successful implementation. Maintaining consistency in such a system is a main challenge as there are various databases working together and hence referential integrity must be maintained to ensure data synchronization (when records from different databases refer each other [6]). To access such a system, one must be familiar with access methods of all the required databases as there is no unique query language or interface to access polyglot systems. If the system allows intersecting subsets of data in different databases, problems like logical redundancy may occur [6]. To avoid such a problem, restrictions have to be applied on duplicate data and this may contradict user requirements in some cases. Polyglot database systems are not able to make efficient optimizations and are also unable to route queries as each database system is working independently and does not have insights into working of other systems [15]. Furthermore, maintenance and administration of such a complex system requires a team consisting of experts in various databases. Reliability of such a system can be questioned because this system is as fault-tolerant as its least fault-tolerant database [22] [23] [24]. Thus, polyglot persistence concept proposes an effective architecture but it has its limitations. Researches made in recent years on polyglot persistence approach include Musketeer [25] which provides an intermediate representation between various platforms for data processing and applications using them [5], which allows users to run queries over more

than one integrated databases and RHEEM [26], which allows storage and processing of multi-model data and interoperability between multiple platforms.

2.4 Multi-model Databases

Multi-model database systems provide an alternate platform to support the storage and processing of multi-model data. They tend to minimize the limitations (complexity of development, operations and deployment) posed by polyglot and similar approaches and enhance their good features. In polyglot and other similar approaches [4], a multiple number of different databases coexist, each handling its own data model; whereas, in multi-model databases, multiple models are managed and handled in a single database system. A multi-model database tries to incorporate more than one data store in a single engine using one query language and an API that covers all the databases involved [6]. Multi-model databases provide improved consistency compared to polyglot systems [15] as there are lesser issues related to data duplication and data synchronization. Also, fault tolerance is higher as now it is dependent on the ability of one database engine to handle faults. The administration and management also becomes more efficient as the experts have to focus their efforts to benefit from one single database system instead of multiple independent databases [24].

2.4.1 Categorization

According to their underlying physical data model, multi-model databases can be classified into two main categories [15]:

2.4.1.1 Single Layout Multi-Model Database

A multi-model database that stores data in a single common physical layout is known as a single layout database. In this type, a single pool of low level primitive operators process the stored data. Each high level logical data model in the multi-model database is provided with high level operators that internally make use of the primitive operators in order to process the stored data for the data model they belong to. The single storage engine in this layout helps in enhancing the integration and optimization features for fetching and processing data from various logical data models. Figure 2.8 shows a framework (GRAPHITE) [3] presented by Marcus et al. in which a set of high level query operators is used to query graph data which is stored in a relational database. Thus, in this framework, functionality of a graph data model is added to a relational database using a single-layout multi-model approach [3]

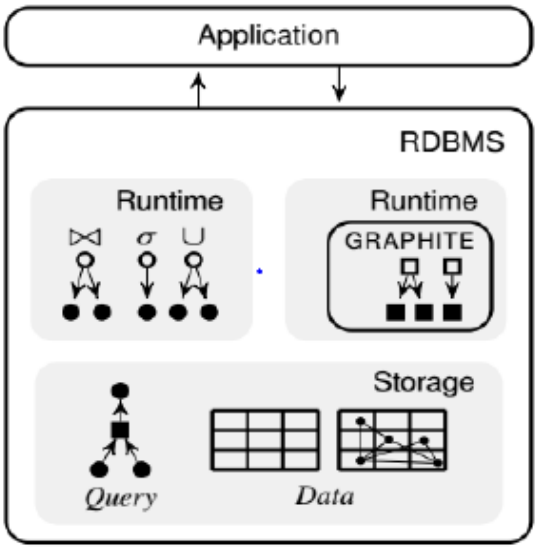


Figure 2.8: Single Layout Multi-Model Database

2.4.1.2 Multi Layout Multi-Model Database

A multi-model database that stores data in more than one physical storage layouts or models is known as a multi layout database. Each high level logical data model is provided a set of high level operators that are directly mapped to the set of primitive operators used to process the data stored in the specific data storage layout according to the specific data model. The specialized multiple storage layouts help make multi layout databases outperform single layout databases [15]. Figure 2.9 shows an example of the multi layout multi-model database based on the GRAPHITE framework.

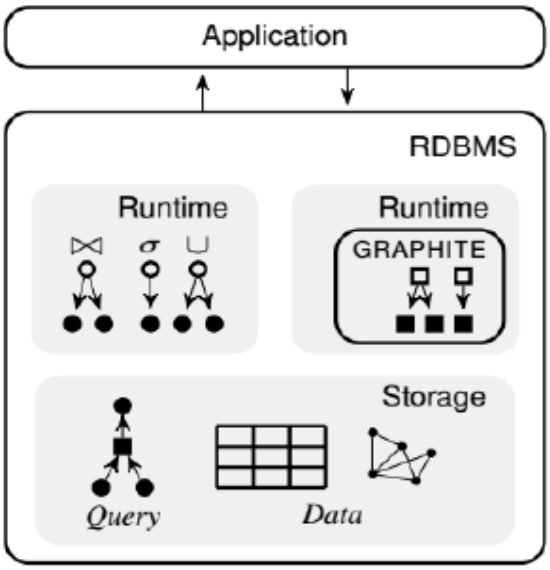


Figure 2.9: Multi Layout Multi-Model Database

Figure 2.10 shows an example of a federated polyglot database. As in the federated system , in polyglot database as well, the query executer lacks internal information about selectivity statistics [4] and though it fetches data from internal databases, it does not have an information that would enable it to optimize the execution. So, in case where data is duplicated in multiple databases, the query optimizer is unaware of the fact that which database would be fastest to answer a query [15]. In case of multi-model databases, the query optimizer is able to access the internal information about the selectivity statistics and hence is capable of optimizing a query using this information.

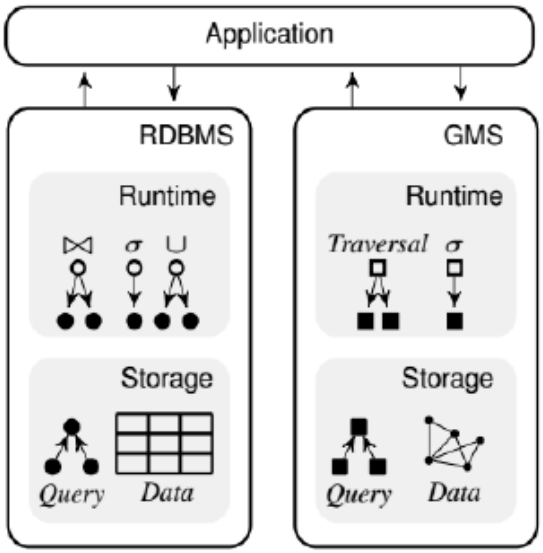


Figure 2.10: The Federated Polyglot Database

2.4.2 An Example - ArangoDB

ArangoDB [2] is an open-source NoSQL multiple-model database [27] developed by ArangoDB GmbH. It supports key-value, document and graph data models. ArangoDB is a suitable choice for applications requiring efficient memory utilization, high performance and convenient methods for querying data [27]. ArangoDB provides an SQL-like data querying language called ArangoDB Query Language (AQL) which fulfills the purpose of efficient data querying. AQL is used to query all different kinds of models available in ArangoDB database. This querying language is independent of the underlying client programming language. It can be run on different platforms like Linux, Windows and OSX.

2.4.3 Challenges

The concept of multi-model databases is comparatively new and due to lack of enough research on how to use or design a multi-model database, the development of such a database is quite a challenging task. Although a number of multi-model databases are available in market, there is still not much information or guidance on how to fully exploit their multi-model features. Some of the crucial points that must be considered during the development of a multi-model database are stated below:

- The data model of the multi-model database should be flexible enough to accommodate data from a number of different data models

[2] https://www.arangodb.com/

- One query language has to be designed that can unitedly query different kinds of data
- Devising ways to perform mapping of data from one model to another has to be done
- Special methods have to be developed to keep the multi-model data consistent while transferring from one model to another
- Design of complex joins to be made in order to perform cross-model querying of data

2.5 Summary

We can summarize this chapter as follows:

- In this Chapter, we have discussed how the evolution of data storage systems has been made from mere manual record storage to complex computerized database systems.
- This Chapter summarizes the history of databases, the ever-increasing storage requirements of rapidly changing data and the evolution of systems to address those requirements.
- Relational and NoSQL models along with their data layouts, integrity constraints and basic operations have been discussed with examples.
- Due to the increasing versatility of modern applications, one data model can not handle the different types of application data. To address this scenario, Polyglot Persistence concept came into being. This concept along with examples has been discussed in detail. The limitations of polyglot approach and the problems developers have to face while implementing a polyglot systems have been highlighted.
- Finally, the need to have a multi-model database, its categories and the challenges one may face in implementing such a database have been explained.

In Chapter 3, we will discuss the evaluation questions that this work attempts to address. Also, the background (experimental setup, tools and software used, benchmarks applied) behind implementation of our prototype, Multi-model Redis Client (MMRC), are explained in Chapter 3.

3. Design - Multi-model Redis Client (MMRC)

In this chapter we discuss the design details of our prototype. Since we selected Redis as a database that enables multi-model functionality, we devote a part of the chapter to describing the modules and features we used.

- Section 3.2 recapitulates the evaluation questions that this work attempts to answer.
- Section 3.3 explains our reasons for chosing Redis and discusses the multi-model features of this system, and its related modules.
- Section 3.4 presents the the experimental setup used.
- Section 3.5, Section 3.6.1 and Section 3.6.2 explain the datasets and benchmarks (YCSB and ArangoDB benchmark) used in this work.
- Section 3.7 concludes this chapter.

3.1 Design - The Second Step

The second step of the our research methodology as proposed by the waterfall model is the design of the experiment to be conducted. This chapter presents the design details including the experimental setup and the selected tools and benchmarks.

3.2 Evaluation Questions

This work attempts to evaluate an off-the-shelf multi-model database, assessing its features and how data could be managed in it. Specifically we aim to understand data management in such a system. We strive to understand: does data load better into one model (over others), or does one model stand out in terms of efficient query processing.

It is our motivation that the comparison on these aspects within a single multi-model database could be informative, helping us to understand how data should be managed in such systems, and how the lifecycle of data could take place; for example by starting from a model, and moving to others, according to certain requirements.

To this aim we implement a prototype of a multi-model database, with respect to benchmark systems, and we employ this prototype to answer the following evaluation questions (which are a more specific formulation from our research questions):

1. How well do different data models perform in terms of data ingestion?
2. How well do different data models perform in regards to basic OLTP operations, with an emphasis on point queries?
3. How costly is the process of migrating data between a central model and other? Can improvements from the database, such as the use of stored procedures and scripting features contribute to the efficiency of this mapping process?
4. How good is each data model for the core task of data querying, when considering more OLAP-like operations?

From these questions we seek to understand if there should be a single central model adopted for data loading and basic OLTP processing; how well is the mapping between models supported; and how well do models perform in analytical tasks. These considerations could be essential to understand the lifecycle of data in a multi-model database.

3.3 Multi-Model Redis

For our work we have selected Redis as a multi-model database.

As explained in previously, a multi-model database has a central engine that must be able to load data in a certain data model, transfer data between different data models and query data in any given data model. Redis fulfills this criteria. ArangoDB, OrientDB, MongoDB and AsterixDB are some comparable alternatives. We selected Redis over MongoDB due to its main-memory design. Ease of adoption was the second criteria that, after an early evaluation, lead us to choose Redis. Furthermore, work comparing the performance of these systems reports that Redis can outperform most of the alternatives evaluated[28].

Redis [1], is a non-relational, in-memory, key-value data store. It is open source, written in ANSI C and works on most POSIX systems. It provides fast KVS implementations for various basic operations [29]. The popularity of this simple key-value database is owed to its speed, ease of use and high performance [30].

Redis-cli is the command line interface of Redis and is used to send commands to Redis and read replies from the server. Furthermore, a large number of programming languages have Redis libraries and language clients, making Redis an easy option for using on applications dealing with different programming languages (e.g Jedis for Java).

With a growing number of NoSQL databases in market, it becomes difficult for users to choose the solution most suitable for their problem. Owing to its fast speed, Redis is considered a good option to be used in write-heavy applications where data is changing quite frequently [31] and is not a suitable choice in cases where the dataset is very large and a chunk of data is accessed a lot more than the rest of the data. Also, Redis is not selected as the best choice to use when dataset does not fit in memory [31].

3.3.1 Data Structures

One of the main characteristics that separate Redis from other key-value databases is that Redis can store and handle high-level data types. These data structures are:

- **Strings** - The simplest type of values associated with Redis keys. Redis keys also belong to String data type.
- **Sets** - Redis Sets are unordered collections of strings
- **Sorted Sets** - Ordered, unique, non-repeating string elements
- **Hashes** - Maps containing fields associated with values
- **Bitmaps** - Bitmap is not exactly a data type but a set of operations defined on the String type, in order to treat it as an array of bits
- **Geospatial Indexes** - A data structure used to deal with geospatial data (latitude, longitude, etc.)

[1] https://redis.io/

```
> MULTI
OK
> INCR foo
QUEUED
> INCR bar
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
```

Figure 3.1: An example showing transactions in Redis

3.3.2 Features

Some of the primary features and common use cases of Redis are described below:

Transactions:

Redis supports the concept of transactions. A transaction allows multiple commands that are executed sequentially. In the middle of this execution, no other request is served. A command *MULTI* is called in order to enter into Redis transaction. The reply *Ok* confirms that a transaction has been started. All the commands after that are queued until execute command (*EXEC*) is given. Similarly scripts are executed in Redis as transactions, granting them complete control of the database.

An example is shown in Figure 3.1.

Lua-Scripting:

The Lua interpreter was introduced in Redis 2.6. EVAL and EVALSHA are used to evaluate scripts. Redis commands can be called from a Lua script by using these Lua functions:

redis.call()
redis.pcall()

Through these two functions Lua scripts are able to call all the functions supported by Redis, including the use of additional modules.

The deployment of Lua scripts can be useful to reduce client-server communication, through shipping to the server a certain amount of processing that would otherwise be managed on client-side.

Persistence:

As a server shutdown or failure proves to be fatal for data stored in-memory, it is very important for databases to provide a back-up method to avoid data loss. Although Redis is an in-memory database, it also allows durability by persisting data to disk. The Redis architecture is structured in such a way that it can provide cache-like behavior while allowing persistence similar to other traditional disk-centric databases [30] . Two methods are provided by Redis to provide data durability; one uses snapshots and the other uses append-only files (AOF) [30], every change made to the in-memory data is saved to the disk, thus allowing more reliability of data. In our work we employ Redis' persistence features.

Sharding:

Redis provides efficient partitioning features, so data in Redis can be split or distributed across multiple instances of Redis, thus utilizing memory efficiently. This way, more data can be stored than possible on one server. Partitioning of data is enabled in Redis owing to its key-value data model.

Some general scnearios where Redis simple data structures can be used are session stores, counters, queues and full-page caches. Some popular companies using Redis are Twitter, Pinterest and Github.

3.3.3 Redis Modules

Redis version 3.2 has introduced externally pluggable functionality or features, called modules, into the Redis database. Using these modules, Redis can be enhanced, and has the potential to be turned into a multi-model database as well. Some of the Redis modules are RediSearch (providing full-text search over Redis), RediSQL (providing full capabilities of SQL-embedding, emulating SQLite), Neural-redis (which allows the online training of neural networks as Redis data types), ReJSON (a document data type for Redis), Redis Graph (a graph data type provided by Redis) and Redis-ML (Machine learning model server). The modules that we have chosen in this work to investigate Redis as a multi-model database are ReJSON and RedisGraph, hence these two will be discussed in detail in the sections below.

3.3.4 ReJSON

ReJSON [2] is a module for Redis that allows Redis to represent itself as a native JSON store. It implements JSON documents as a native Redis data type. Storing, fetching and update of JSON documents using Redis keys can be carried out through ReJSON. ReJSON is developed at Redis Labs [3], it is written in C++, and the code base is open source manner[4].

[2] https://oss.redislabs.com/rejson/
[3] https://redislabs.com/
[4] https://github.com/RedisLabsModules/rejson

```
127.0.0.1:12543> JSON.SET foo . '{"foo" : "bar"}'
OK
```

Figure 3.2: An example of JSON.SET command on Redis-cli

```
127.0.0.1:12543> JSON.GET foo
{"foo":"bar"}
```

Figure 3.3: An example of JSON.GET command on Redis-cli

3.3.4.1 Using ReJSON

Before using ReJSON, a Redis server and any Redis client (redis-cli or any other client) has to be set. The module is built separately and loaded on the Redis server while it is starting up. Redis gives a notification when ReJSON is successfully loaded and is ready to be used. Like all other Redis data types, ReJSON values are stored in keys and access to them is provided by a specialized set of commands. This set of commands is designed specifically such that it is easy for users of Redis and JSON both to comprehend them easily. Some basic commands will be discussed here.

Commands

One of the most basic ReJSON commands that is used to create a document is JSON.SET. This command sets a value to a ReJSON key. This value is retrieved by JSON.GET command. Examples of these simple commands using Redis-cli as the client are given in Figure 3.2 and Figure 3.3. The usage of paths and JSON formats are key characteristics of working with ReJSON.

Other common commands used in ReJSON are shown in Table 3.1 .

Storage and Representation

The storage of ReJSON documents is done as binary data, which each document represented in a tree structure, which allows fast access to nested documents.

In ReJSON, whenever JSON.SET command is called, a streaming lexer parses the input JSON and builds a tree data structure from it. ReJSON stores this binary data in the tree's nodes.

Figure 3.4 andFigure 3.5 show simple examples of how data is stored in a tree structure in ReJSON.

```
127.0.0.1:6379> JSON.SET object . '{"foo": "bar",
"ans": 42}'
OK
```

Figure 3.4: An ReJSON query example

Command	Action
JSON.DEL	Deletes a value
JSON.MGET	Retrieves values associated with multiple keys
JSON.TYPE	Returns the type of a value
JSON.STRLEN	Returns the length of the value string
JSON.ARRAPPEND	Appends a value into the array of values associated with a key

Table 3.1: Some common ReJSON commands.

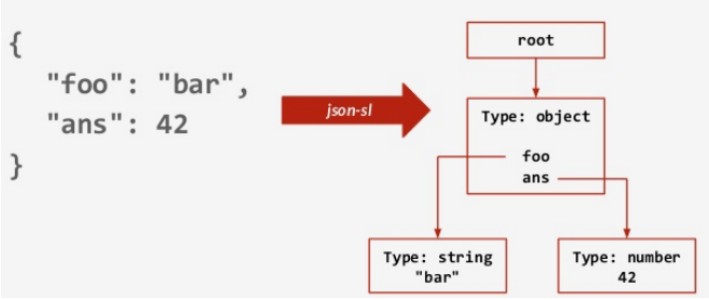


Figure 3.5: Example of a tree structure where ReJSON data is stored

Primary Features

Some of the main features of ReJSON modules are stated below:

- ReJSON supports the JSON standard completely[5].
- Keys in ReJSON may contain any valid JSON value; e.g. scalar values, arrays and objects.
- Nested documents are also supported.
- ReJSON can be used with many Redis and ReJSON clients. Some of the languages that have a supporting library and client implementation for ReJSON are Java (JReJSON) and and Python (rejson-pi)
- ReJSON supports atomic operations for all JSON values types.

3.3.5 Redis Graph

Redis Graph [6] is a Redis module that represents a graph database. Using Redis Graph, a user can implement the functionalities of a graph store in Redis. Just like in any other graph store, nodes in Redis Graph represent entities and edges represent relationships or connections. This module is implemented using C++ and is also available on Github [7] as an open-source project.

3.3.5.1 Using Redis Graph

Just like other Redis modules, in order to use Redis Graph, the user has to start the Redis server and select a suitable Redis client. The module, Redis Graph, has to be loaded on to the server during start-up time. After receiving the notification of successful loading, Redis Graph is ready to be used.

Redis version 4.0 or above is required to run this module.

Instead of inventing their own query language, the creators of Redis Labs have implemented a subset of the Cypher query language [21] in this module. Cypher is one of the most popular graph database query languages. The openCypher project [8] makes Cypher free to be adopted for graph processing purposes by any product or application. Redis Graph is one of the many databases that have implemented Cypher, others being SAP HANA Graph, AgensGraph and Neo4j. Some of the basic commands that are available in Redis Graph to access and query graphs in Redis are discussed here.

[5] https://tools.ietf.org/html/rfc7159
[6] https://oss.redislabs.com/redisgraph/
[7] https://github.com/swilly22/redis-graph
[8] https://www.opencypher.org/

```
graph.QUERY <graph_id> 'CREATE (:<label>
{<attribute_name>:<attribute_value>,...})'
```

Figure 3.6: Structure of a Redis Graph query to create a graph

```
GRAPH.CREATENODE <Graph_Name> <Label>
<Attributes>
```

Figure 3.7: Structure of a Redis Graph query to create a node in a graph

Commands

To work with a graph, the major components are nodes and their relationships. The GRAPH.QUERY command serves as a starting point to let Redis not that the following command is a Redis Graph command. Next the user needs to determine the name of the graph to be accessed. After giving such information the user can write any command. Generally all commands start with either MATCH or CREATE.

The structure of CREATE commands is shown in Figure 3.6. Commands used to add nodes to a graph are shown in Figure 3.7.

Although, only a subset of Cypher query language is available in Redis Graph, it is still enough to extract useful insights from the graphs and analyze their contents. GRAPH.QUERY command is used to query the given graphs in various ways. Some of the common queries are Match-Where-Return (matches a given clause and returns the required graphs), Aggregations (sum, min, max etc.), Limit (to limit the number of records returned by a query), Delete (to delete a node and its relationships) and Set (to update the properties of nodes and their relationships). Relationships can be queried by using the path matching (s:person)-[:FRIENDS_OF]->(p:person) syntax popularized by Cypher. Nonetheless, in our work with Redis Graph we found that the libary does not support regular path queries yet (i.e., queries where regular expressions, for example signaling the number of hops to be considered, are passed to the operands, (s:person)-[:FRIENDS_OF*1..2]->(p:person)).

Storage and Representation

Different graph databases rely on different data structures for representation and storage of graphs. For Redis Graph, the main focus while choosing a data structure for graph representation was fast querying of the graph. For this purpose, the developers have chosen a structure called Hexastore. A hexastore is used to store a list of RDF triplets (three elements). Each triplet is composed of a subject, a predicate and an object. In Redis Graph, these are equivalent to two nodes and one relationship between them (subject is the source node, object is the destination node and predicate is the relationship). A hexastore in Redis Graph contains all six permutations of these three, such that queries can be performed in every direction, starting from every point of a relationship.

When a query is run, the specific hexastore is searched to find the strings containing the prefix given in the query. It is true that hexastore is a memory consuming data structure as it stores six triplets for one relation, but Redis Graph has implemented for this structure a trie data structure which utilizes memory efficiently and also provides fast searches[9]. The reason is that it doesn't create duplicates of the string prefixes that it has already seen before [32].

Limitations

Redis Graph is a module which still needs a lot of work. There are many areas in which the research is still going on and an improved version may appear anytime soon. Some of these areas are given below:

- Currently, there is no support for patterns in which a node has more than one outgoing edges. In this case, only one edge is considered and the rest are ignored .
- A number of aggregation functions (stdev, percentileCount etc.) are still not fully implemented
- There is no implementation for finding the shortest path between two nodes
- Queries on a single node (MATCH (A) RETURN A) cannot be made

As of now, Redis Graph is an experimental module and has not had any release version yet.

Primary Features

Some of the distinguishing features of Redis Graph are:

- Redis Graph provides fast indexing and querying of graphs on top or Redis.
- In Redis Graph, nodes and relationships can also have attributes, and thus can be labeled.
- Storage of graph index and entities are done in-memory to allow better performance.
- Redis Graph, like other modules, also supports the persistent features of Redis, according to configuration. Hence, storage on disk is also possible.
- The data structures implemented in Redis Graph for storage are memory-efficient.
- Simple and very popular query language.

[9] https://oss.redislabs.com/redisgraph/

3.4 Experimental Setup

Here, we will describe the machines, tools and datasets used for our implementation. The operating systems used in this work are Windows 10, Linux Ubuntu 17.04 and Linux bash shell on Windows 10. The machine mostly used in this work is a multi-core commodity machine (HP ProBook 4530s) with a 64-bit operating system. The specifications of the machine used are:

- **Processor:** Intel Core i3 (2nd Gen) 2310M / 2.1 GHz
- **Number of cores:** Dual-core
- **Cache:** L3 - 3MB
- **RAM:** 4GB

Furthermore, the tools and benchmarks that have been used in this work are:

- Redis 4.0.2
- Redis Graph (Redis Module)
- ReJSON (Redis Module)
- Yahoo! Cloud Serving Benchmark (YCSB)
- ArangoDB Benchmark

3.5 Datasets used

Apart from the YCSB generated dataset (explained in the next section), the dataset that has been used in this work is the Pokec [10] dataset, which serves as a backbone for the ArangoDB multi-model benchmark[11]. Pokec is the most popular social networking platform in Slovakia [33]. It is almost 10 years old and connects around 1.6 million users. The Pokec datasets contains anonymous data of the Pokec network. This data includes gender, age, interests, professions etc. of Pokec users.

[10] https://snap.stanford.edu/data/soc-pokec.html
[11] https://github.com/weinberger/nosql-tests/

3.6 Benchmarks used

In our work we used two benchmarks, we discuss them next.

3.6.1 Yahoo! Cloud Serving Benchmark

In order to evaluate the first of our evaluation questions, addressing the loading of data and simple operations, when using the different models, we consider the Yahoo! Cloud Serving Benchmark.

With the emergence of various new databases, it becomes a difficult decision for developers to choose the right option for their application. Firstly because all databases provide different benefits and are suitable for specific problem domains and secondly because there is no simple and straightforward way of comparing the performance of on database with another. One way to analyze this performance difference is through benchmarks. Benchmarking is used to evaluate different parts and levels of systems, from CPU, database software, to complete enterprise systems [34]). Yahoo presents Yahoo! Cloud Serving Benchmark (YCSB) [12] that facilitates performance comparisons of various databases on basic CRUD operations.

YCSB is composed of two major parts: The YCSB Client as the workload generator and the second part is the YCSB core package that contains the set of standard workloads. The workloads describe the data that will be loaded into the database during loading phase [1] and the queries that can be performed on the data set during querying phase. Although the workloads present in the standard package (read-heavy workloads, write-heavy workloads, scan workloads, etc.) adequately assess different aspects of a system's performance, YCSB also provides options for extensibility, thus, user-defined workloads can also be added to it. The YCSB framework along with the standard set of workloads is open-source and is publicly available to be used to evaluate performance of database systems. The architecture of the YCSB framework is shown in Figure 3.8:

There are six core workloads present in the core package. All of these six workloads have similar dataset. These workloads are:

- **Workload A (Update heavy workload):**
  This workload contains 50% reads and 50% writes.
- **Workload B (Read mostly workload):**
  This workload contains 95% reads and 5% writes.
- **Workload C (Read only):**
  This workload has 100% reads.
- **Workload D (Read latest workload):**
  In this workload, new data is inserted and the most popular records are the ones that are most recently added.

[12] https://github.com/brianfrankcooper/YCSB

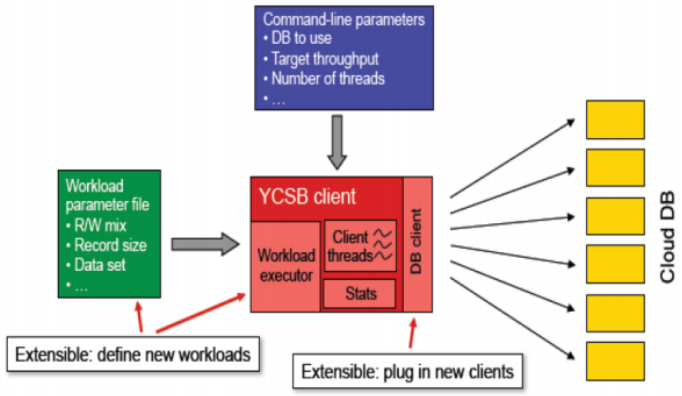


Figure 3.8: The YCSB Architecture [1]

- **Workload E (Short ranges):**
  In this workload, instead of querying the records individually, short ranges of the records are queried.

- **Workload F (Read-modify-write):**
  In this workload, the client writes back the changes into the record after reading and modifying it.

3.6.1.1 Running a YCSB Workload

There are six major steps [13] to successfully run a workload in order to test a system's performance:

1. The first and foremost step before testing a system's performance with YCSB is to set up the system to be tested. Tables or other storage elements according to the database being used have to be created where the YCSB workload will be stored. Thus, before running the YCSB client, storage of records from YCSB workloads has to be arranged. For the case of Redis and the modules no items need to be created.
2. Creation of a class that handles insert, update, delete and scan calls that are produced by the YCSB client has to be done. The YCSB client will automatically load this class and use it to send different function calls to the database API. For our work we employed the Redis client, and we extended it, creating several versions that represented the access to the different modules.
3. After that being done, the appropriate workload has to be selected. This workload will define the data loaded into the database and the operations performed on this data.
4. YCSB provides additional parameter settings that user may modify according to the requirements. These parameter settings are done after the specific

[13] https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload

workload has been selected. These settings include setting the number of client threads (by default one but more can be added), to set the number of operations performed per second, and to check the status of a long running workload after frequent intervals. In our work we employed the basic modules.

5. At this point, when the workloads along with additional parameters have been selected, data is loaded into the database. The output of this step tells clearly how long it took for the data to load and its overall performance.
6. Once the data is loaded, the YCSB client is told to perform operations on this data. The result of this step shows statistics of read, write and other operations (associated with a particular workload) and performance of execution of these operations including execution time, throughput, number of operations, etc.

Some of the properties that the Workload generator considers while generating workloads are fieldcount ((no. of fields in a record, by default 10), fieldlength (the size of a field, by default 100), readproportion (proportion of read operations), updateproportion (proportion of update operations), insertproportion (proportion of insert operations), requestdistribution (choosing a distribution used to select records for operation; uniform, zipfian or latest).

3.6.2 ArangoDB Benchmark

To date there is no single standard multi-model benchmark to test, beyond the basic CRUD operations, some simple analytical operations on different models. The closest benchmark that we were able to encounter, following the mention from Lu and Holubova[4] was the ArangoDB benchmark.

The ArangoDB benchmark is basically created to test the performance of ArangoDB itself and many other databases in different scenarios. It is completely open-source and all the scripts and details of the tests are available online [14]. Before any of the test included in the benchmark is performed, there is a warm-up phase during which data is loaded in the database. The dataset used for the tests in this benchmark is the Pokec [15] dataset. This dataset not only allows simple read/write operations but also graph queries (e.g. to calculate shortest path between two nodes).

Some of the tests included in this benchmark are given below:

- **Single-read:** read queries on documents
- **Single-write:** write queries on documents
- **Aggregations:** group users by age and count the total per group

[14] https://github.com/weinberger/nosql-tests
[15] https://snap.stanford.edu/data/soc-pokec.html

- **Neighbors:** searching direct neighbors and returning their ids
- **Neighbors second:** searching direct neighbors and neighbors of neighbors, returning only the ids
- **Neighbors second with data:** searching direct neighbors, neighbors of neighbors and returning their profiles
- **Shortest path:** To find how close two nodes are to each other
- **Memory:** To find average of total main memory consumed during the test runs

For our evaluations we have used the aggregations, neighbors, neighbors second and neighbors second with data use cases. However we modified the neighbors queries such that they worked on the complete dataset, instead than on pre-configured nodes.

3.7 Summary

This chapter can be summarized as follows:

- This chapter implements the second step (design) of our research.
- In this chapter, we have explained the experimental setup used in this work and the evaluation questions that this work attempts to target.
- Furthermore, Redis, along with its modules that have been made use of in this work are explained in detail.
- The benchmarks, YCSB and the ArangoDB benchmark, used for the tests are also explained.

In the next chapter, we will discuss the implementation of our first major test, data loading into different data models, describe the results, provide their summary and discuss them in detail.

4. Data Ingestion and Basic CRUD Queries in Different Data Models

In this Chapter, we aim to address the first evaluation question. Namely, we study the time for loading data into the different models, and for performing very local read operations, by using the YCSB benchmark.

This chapter and the next cover the implementation and evaluation steps of the waterfall model, with each chapter corresponding to a different iteration.

- We start by restating the evaluation questions that we address in this chapter (Section 4.1).
- In Section 4.2 we present the details of the implementation required for this test.
- Next in Section 4.3 we collect the results of our evaluation to answer the research questions defined for this chapter.
- Section 4.4 concludes the chapter.

4.1 Evaluation Question

The first part of testing a multi-model database is to evaluate how good each supported model performs for the core tasks of data loading and basic CRUD operations. This is our first test which is also stated in the following evaluation questions.

1. How well do different data models perform in terms of data ingestion?
2. How well do different data models perform in regards to basic OLTP operations, with an emphasis on point queries?

4.2 Implementation - The Third Step (1)

This section corresponds to the implementation step for the first test (the third step of our research methodology). Here, we describe the details of the test performed.

4.2.1 Loading of Data

As stated earlier, the core tasks that mark the performance of a multi-model database are data ingestion into models, basic CRUD operations, transformation of data from one data model to the others and performance of data querying over any of the data models. The data models presented in this work are:

- Key-value store (Redis)
- Document store (ReJSON)
- Graph (RedisGraph)

In the first set of tests, collected in this chapter, we evaluate the performance of each model in terms of data ingestion. For this purpose, we have implemented 2 Yahoo! Cloud Serving Benchmark (YCSB) clients for each of the novel data models, based on the existing Redis client.

The data to be ingested in the modules was also taken from the six core workloads provided by YCSB. This benchmark is discussed in detail in Section 3.6.1.

We have used the six core workloads and loaded them into Redis, ReJSON and RedisGraph, evaluating the resulting performance.

The workloads are, as listed below:

1. **Workload A (Update heavy workload):**
   This workload contains 50% reads and 50% writes.
2. **Workload B (Read mostly workload):**
   This workload contains 95% reads and 5% writes.
3. **Workload C (Read only):**
   This workload has 100% reads.
4. **Workload D (Read latest workload):**
   In this workload, new data is inserted and the most popular records are the ones that are most recently added.
5. **Workload E (Short ranges):**
   In this workload, instead of querying the records individually, short ranges of the records are queried.
6. **Workload F (Read-modify-write):**
   In this workload, the client writes back the changes into the record after reading and modifying it.

```
Jedis jedis = new Jedis("localhost");

jedis.set("foo", "bar");

String value = jedis.get("foo");
```

Figure 4.1: An example of using Jedis to create a key-value pair

4.2.2 Setting up YCSB client for Redis

The YCSB framework provides the opportunity to interface with different database systems. We have written a YCSB client to run against Redis, ReJSON and Redis Graph. This client is written using Java. The code is provided in an appendix (Section 8.1).

To interface Redis with YCSB, we have made use of Jedis[1], the Java client for Redis. Jedis is compatible with Redis version 2.8 onwards. It supports many Redis operations both administrative and query-related, including sorting, implementing transactions, sharding, pipelining and performing different operations on all data types of Redis. A simple example of using Jedis is shown in Figure 4.1:

The data loading into Redis, ReJSON and Redis Graph has been evaluated in terms of throughput (operations/sec). Our results portray the average across the loading for all workloads. At first, throughput for the data ingestion into a data model is measured and then the performance of Redis and the modules is evaluated for different OLTP operations provided by the workloads like reads, updates, inserts, scans and read-modify-writes.

4.3 Results and Discussion

This section corresponds to the last steps (evaluation and verification) of our research methodology for the first test. In this section, We present the results of our data loading evaluation.

Figure 4.2 shows the performance of Redis (throughput achieved for the workload) and the two modules for data load and other operations provided by the YCSB workloads.

The first test of data ingestion into Redis and its modules has provided very interesting results. It can be seen clearly that in most of the cases the basic Redis performs better than the two modules, and RedisGraph performs the worst in all cases except one.

[1] https://github.com/xetorthio/jedis

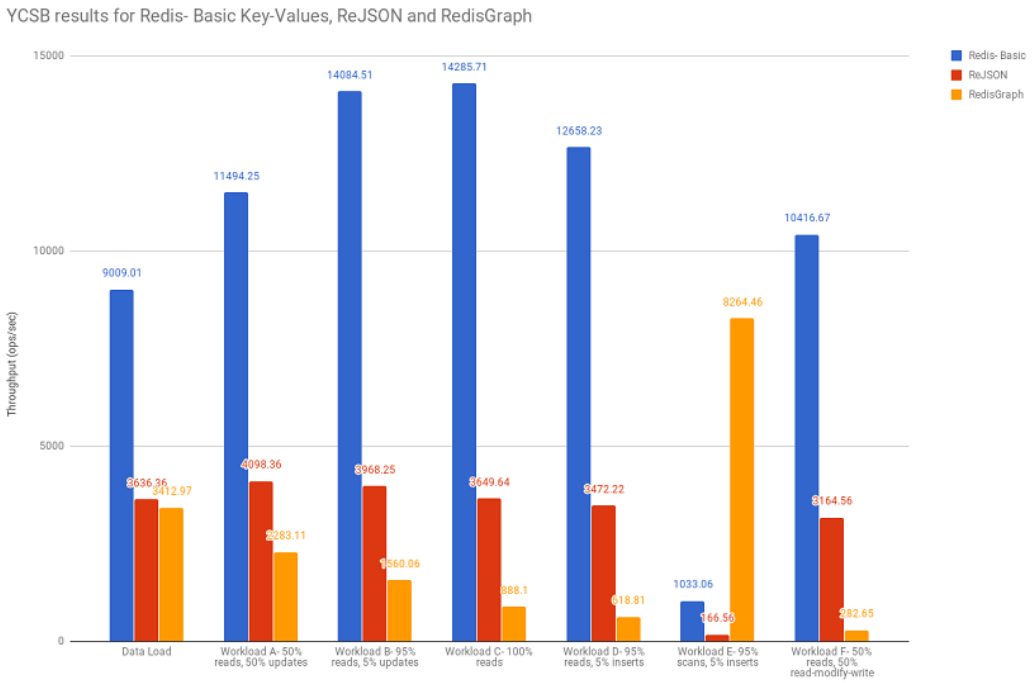


Figure 4.2: Performance of Redis, ReJSON and RedisGraph in terms of data loading

The reason for ReJSON to perform inefficiently, when compared to Redis could be due to the overheads caused by serialization and de-serialization to the JSON format which is accomplished using gson (a library from Google). Furthermore there is no specific aspect in the workloads that could highlight ReJSONs expected application, consisting of nested document operations.

In workloads involving updates it can be understood that Redis Graph might suffer from a performance deterioration, due to the maintenance cost of how it stores data in an index structure. Similarly, when workloads are read intense (which are random-access patterns), the hexastores do not bring any specific benefit. Furthermore, since in hexastores each property is indexes separately, there is a necessary effort for reconstructing the tuples, which could introduce overheads.

It can be seen in Figure 4.2 that when the Workload E is considered, in which scan operations are handled, the performance of RedisGraph is notably better than Redis basic and ReJSON. The reason for this may be due to the type of index being used in RedisGraph and how it handles requests. It would be worthwhile to study the exact performance profiles that underly this observation, because sequential access patterns and an efficient management for those could be the cause of the good performance.

Overall, Redis achieves a 3 fold better throughput than the one of the other models. ReJson follows in second place with 2-4 times better performance than the final alternative, RedisGraph.

Overall, the results show that Redis basic is only inefficient at scans. The performance of ReJSON lies somewhat between Redis basic and RedisGraph except for scans where RedisGraph is the best, as mentioned above.

In the important task of data load Redis basic performs close to 3 times better than the alternatives. It furthermore performs better for random access patterns (or those random but with skew towards popular keys), as seen in read intense workloads. Hence our results suggest that Redis basic should be chosen for data loading and for workloads where there is a large component of random reads. Whereas only for the case of scans, RedisGraph should be selected.

4.4 Summary

We can summarize this chapter as follows:

- In this chapter we provided the results for the first tests performed over the Redis modules. These tests involved data ingestion and workloads mostly consistent of basic point queries into Redis and the two modules, evaluated in this work.
- The most important observation is that data load happens better in basic Redis, which motivates us to consider that this should be the default model for loading data, when there are time constraints for data ingestion. We also observed that Redis Graph is the most inefficient of models, we judge that this is due to overheads from updating the index structure that this module keeps.
- Regarding the workloads themselves, we found that basic Redis overall performs better for these workloads, consisting of random access patterns (though with a skew, defined by default, where data the selection of tuples per query is done following a Zipf distribution, with 20% of the data accessed 80% of the time).
- We also noted that sequential access patterns, as those present in the short scans perform better for Redis Graph, leading this model to outperform all other alternatives. This observation suggests that there are indeed cases where it is reasonable to design data in applications to have a lifecycle in multi-model databases, where loading happens to one model, and data is then moved to models that match the workloads.

In the next chapter we present the results of our second major tests, i.e. evaluating the transformation of data across models, and the querying of data in Redis, ReJSON and RedisGraph.

5. Moving Data Across Models and Querying of Data in Different Models

This chapter attempts to address the second set of evaluation questions in this work, i.e the cost of moving data across models, and the performance of data querying in different data models.

- In Section 5.1 we restate the evaluation question that we address in this chapter.
- Section 5.3 presents the details of the implementation setup required for this test.
- Section 5.6 gives the evaluation of the results.
- Section 5.7 concludes the chapter.

5.1 Evaluation Question

The second main tasks that a multi-model database must cater for along with data loading, is the transformation of data between models, and the querying of data per model. To evaluate these aspects we establish the following evaluation questions:

3. How costly is the process of migrating data between a central model and other? Can improvements from the database, such as the use of stored procedures and scripting features contribute to the efficiency of this mapping process?
4. How good is each data model for the core task of data querying, when considering more OLAP-like operations?

5.2 Implementation - The Third Step (2)

This section corresponds to the implementation (the third step of our research methodology) details for the second test.

5.3 Setting up

In order to address the questions stated above, we select a benchmark that contains queries in different models, the ArangoDB benchmark. Such benchmark uses as a dataset the Pokec social network graph, an open source dataset.

Our first task was to load this data into basic Redis, with vertexes and edges mapped into a same key-value store but prefixing their keys such that the entities could be distinguished (we start the vertexes with the letter v and the edges with the letter e).

For our evaluation we observed that the high number of edges (30 million) slowed down the loading process and lead to frequent crashes in the transformation process. In order to reduce this risk we made the decision to only load a small proportion of the edges, according to how they are sorted in the official version of the dataset. Hence, after evaluations, we decided that, for repeatability, the use of 100k made the tests more manageable. Accordingly we tested on all the vertexes, but only that small number of edges.

After considering these choices we had to consider possible ways for efficiently transforming the data from one model to another; and for supporting the different queries selected. We discuss such choices in the subsequent sections.

5.4 Moving Data Across Models

After considering the design of Redis, we determined that in order to copy data from one model to another it would be necessary first to select the keys of the items to move, and then to query Redis for each each key, to fetch the corresponding data. Next, the data would have to be created into the new module using the corresponding commands.

Since the redis-cli does not support looping statements, other than through Lua scripts, we considered two alternatives to iterate through each key, retrieve the data and send to the new module. The first choice was by using a client in a programming language, the second was by using the scripting features (or stored procedures).

For ReJSON the creation of data into the corresponding module was simple: documents did not involve nesting, and vertexes and edges were mapped as separate documents.

For Redis Graph we stumbled on a limitation of the current interface: we were not able to call the MATCH statement over two different alias. What this means as that, for the cases where an edge was using 2 already seen vertexes, then one of them would have to be created. In our work we took advantage of our selection of edges to enable this repetition. However, due to loss of generality, we do not perform the transformation test across models for edge data.

5.5 Querying of Data

As mentioned previously, to perform more analytical queries we have used the queries from ArangoDB benchmark to assess the performance of Redis, ReJSON and Redis graph. ArangoDB benchmark is basically created to test the performance of ArangoDB itself and many other multi-model databases in different scenarios. It is completely open-source and all the scripts and details of the tests are available online [1]. Before any of the test included in the benchmark is performed, there is a warm-up phase during which data is loaded in the database. The dataset used for the tests in this benchmark is the Pokec [2] dataset. This dataset not only allows simple read/write operations but also graph queries (e.g. to calculate shortest path between two nodes).

From the tests in this benchmark we selected the following:

- **Aggregations:** group users by age and count the total per group
- **Neighbors:** searching direct neighbors and returning their ids
- **Neighbors second:** searching direct neighbors and neighbors of neighbors, returning only the ids
- **Neighbors second with data:** searching direct neighbors, neighbors of neighbors and returning their profiles

We did not include shortest path queries, which would introduce loss of generality, due to our choice of loading only a portion of the indexes. We also omitted the single queries, as they are comparable to the YCSB CRUD operations evaluated previously.

From the queries we found that they could be expressed in OpenCypher as single (or at most two) queries. To implement for the other models we used the Java language clients, and our most reasonable implementation.

5.6 Results and Discussion

This section corresponds to the fourth and last steps (evaluation and verification) of our research methodology for the second set of tests.

Figure 5.1 collects our results across 10 runs of moving data between models. We report exclusively the loading of profile data (i.e., vertexes). The results show that, when compared to loading on basic Redis, the cost of moving the data to a different module/model, is around 2.5x-12x more than that of the loading into Redis. Here too we observe that the slowest model to load data into is Redis Graph, a fact that we originally attributed to the cost of updating data in the hexastore. However our observations

[1] https://github.com/weinberger/nosql-tests
[2] https://snap.stanford.edu/data/soc-pokec.html

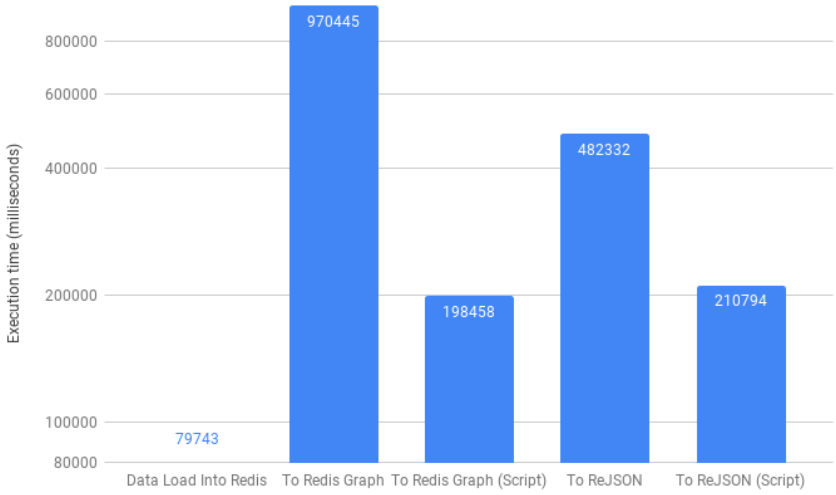


Figure 5.1: Average execution time for moving Pokec data from Redis basic to other modules

when using scripting (instead of the Java language client, Jedis) lead us to question such asssumption, as we will explain next.

In our study we also evaluated the usefulness of scripting. We found out that for both Redis Graph and ReJSON there are performance gains stemming from this approach, leading to 5x and 2.2x performance gains, respectively. When using scripting we note that the gap between original data load and loading into Redis Graph is reduced.Furthermore we note that in average Redis Graph does load faster than ReJSON, though the difference is proportionally small, being only 1.06x. However this is an important observation since it points out that the inefficiency of loading data into Redis Graph could have other sources apart from the loading into the hexastore (which remains the same for both alternatives, with and without scripting).

Scripting helps in keeping the data on the server side, which is pertinent when moving data from one storage to another. Scripting also helps to remove the number of calls that the client needs to send to the backend. Furthermore scripting could have an impact on access patterns, according to how scripts are executed, and it can improve concurrency control, since a script runs a single transaction in Redis. Hence, we believe that one of these factors requires research for improving the loading of data into Redis Graph and ReJSON through the language clients.

We should also note that more careful tuning of the scripts could lead to more improvements.

Moving on to the querying, Figure 5.2 displays our results for performing the aggregation query. From this plot it is evident that language support for the operation (i.e., in the case of Redis Graph) and the use of the hexastore leads to better performance than implementing the features from a language client (i.e., in the other cases). We observe

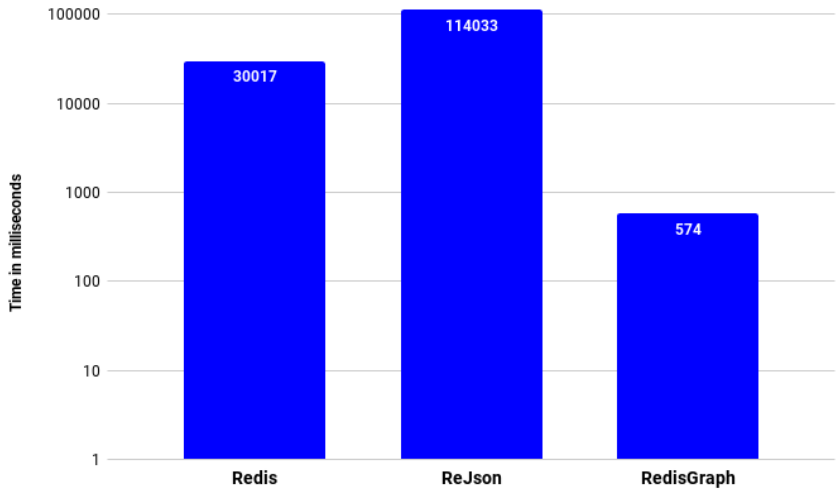


Figure 5.2: Average execution time for the aggregation query of the ArangoDB Benchmark across Redis modules

around 200x performance differences between the cases. On the other hand Redis is close to 4 times better than ReJSON, which is not specially suited for this kind of queries, as there are no nested documents involved.

A key differentiator in these results is the data model employed. By representing data in the form of triples, Redis Graph creates an index entry for each property of vertexes and edges (since a property is just a triple connecting a vertex to a property with a given assignation relationship). As a result, there is an index which can be immediately used to answer aggregation queries. Nonetheless, the gains at query time are tied to the high comparative cost of inserting data into this model, as seen during the load process in Section 4.3, where Redis Graph ranked consistently as the less performing alternative.

This trend is further continued in the Neighbors query of the benchmark, as shown in Figure 5.3. In this query we ask for the identifiers of the immediate (1-hop) neighbors for all vertices in the Pokec dataset[3]. For this case Redis Graph is 67 time faster than Redis and 220 times faster than ReJSON. Once again for this case the results can be attributed to both the language support for the query, and the specialized indexing that Redis Graph includes for managing relationships, which other models do not posses.

Similar results are shown in Figure 5.4. Here we see that by evaluating the condition of whether it is possible to hop a second time, the performance for all cases suffers a deterioration, which is within 1.2-2x for all modules. We should note that the results with respect to the first hop do not change (i.e., stemming from our non-inclusion of a large amount of edges).

In this evaluation, given that currently Redis Graph does not support regular path queries (which would enable to collect in a single query all results from one or two hops),

[3]It should be noted that we only included the first 100k edges.

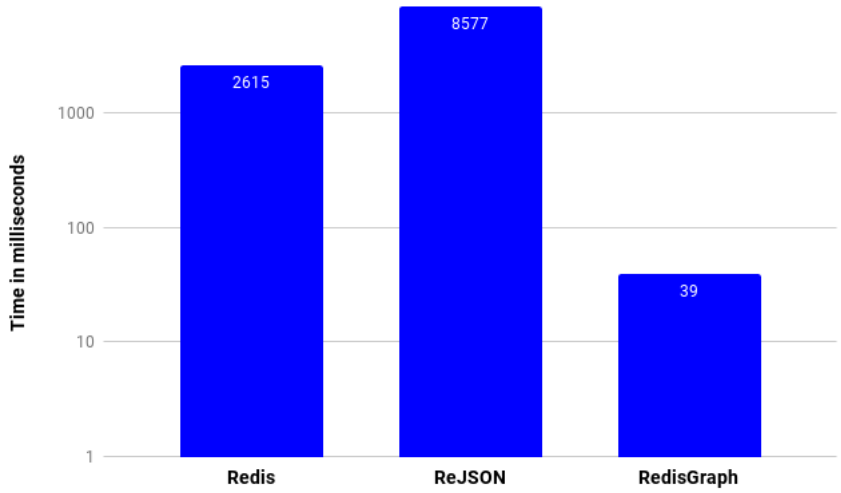


Figure 5.3: Average execution time for the neighbors query of the ArangoDB Benchmark across Redis modules

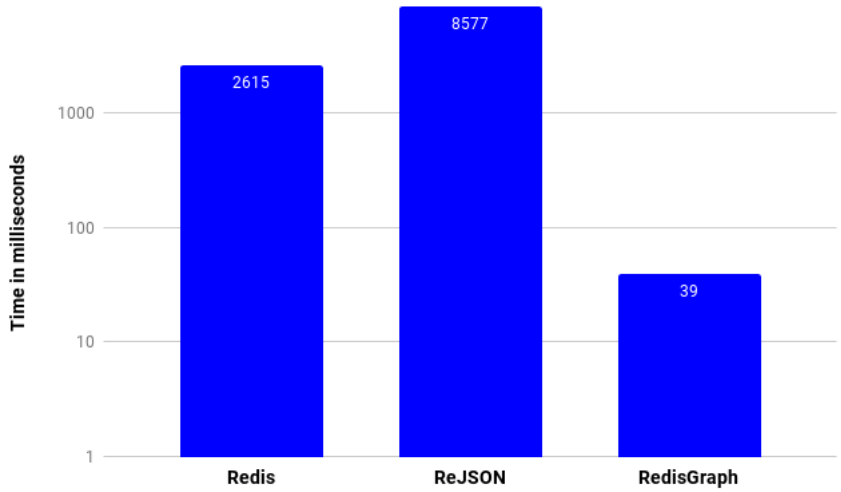


Figure 5.4: Average execution time for the neighbors2 query (with 2 hops) of the ArangoDB Benchmark across Redis modules

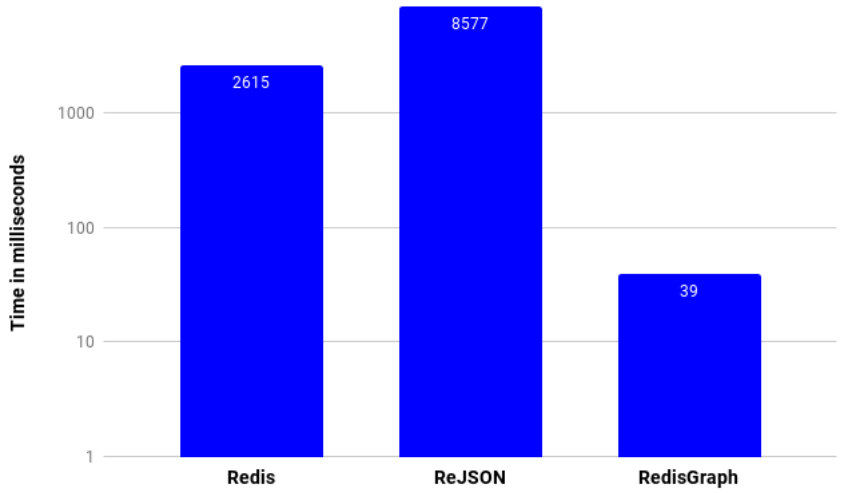


Figure 5.5: Average execution time for the neighbors2data query (with 2 hops and returning all attributes from the resultant vertexes) of the ArangoDB Benchmark across Redis modules

we had to break down the query into two successive queries, one with one hop, and one for two hops. The total column shows the total of adding both these subqueries. Results show once again a large gap in the performance of the modules, with Redis Graph being 48 times faster than Redis basic, and 142 times faster than ReJSON.

Figure 5.5 collects the results of the final evaluation in our study. Here, unlike the previous query, all the data of the matching vertexes is returned. This introduces a small overhead for tuple reconstruction.

As more data is being read, following a random access pattern, ReJSON shows the largest absolute performance deterioration, when comparing Figure 5.4 and Figure 5.5. These observations are consistent with our findings for the YCSB benchmark regarding this access pattern for the module. Following the same observations for the YCSB benchmark, we could have expected a similar behavior in Redis Graph. Though this is not particularly notable when comparing the execution times across the modules, the performance impact is high (when using a relative measure it is a difference of 1.5x, whereas ReJSON scores a difference of 1.68x and basic Redis of 1.44x). In fact, the efficiency from how Redis Graph handles the query when comapred to other modules, hides the performance deterioration of the additional data fetch.

With this we conclude our study on the goodness of models for querying, and in the costs pertaining to moving data across modules. In the next section we summarize our observations.

5.7 Summary

In this chapter we sought to answer to these questions: How costly is the process of migrating data between a central model and other? Can improvements from the database, such as the use of stored procedures and scripting features contribute to the efficiency of this mapping process? How good is each data model for the core task of data querying, when considering more OLAP-like operations?

We can summarize our findings as follows:

- Regarding the migration we observed that by coupling data access in one module with insertion in another module, the cost can be higher than that of data loading into a single module (nonetheless it is higher within the same order of magnitude, for ReJSON, which suggests that with careful engineering and adding efficiency to the process this gap could be improved). However, we should note that further studies are needed in this regard, since the data movement could be studied while deleting the data in one module, thus not increasing the amount of data stored.

- We observed that moving data to ReJSON is more efficient than moving data to Redis Graph. These results are consistent with our findings when evaluating these models for the data load from the YCSB benchmarks, and for the different workloads with updates.

  In our study we also evaluated the usefulness of scripting. We found out that for both Redis Graph and ReJSON there are performance gains stemming from this approach, leading to 5x and 2.2x performance gains, respectively. When using scripting we note that the gap between original data load and loading into Redis Graph is reduced.Furthermore we note that in average Redis Graph does load faster than ReJSON, though the difference is proportionally small, being only 1.06x. However this is an important observation since it points out that the inefficiency of loading data into Redis Graph could have other sources apart from the loading into the hexastore (which remains the same for both alternatives, with and without scripting).

- We can also report that the use of scripting can improve the process. However we should note that further studies are necessary, since more optimal scripting could be adopted.

- We show interesting results when comparing the use of scripts for Redis Graph and ReJSON. The performance difference between them decreases, and Redis Graph outperforms ReJSON by a limited factor of 1.06x. Such observation suggests that the inefficient performance of Redis Graph on data loading as observed in YCSB could be improved in some way through the use of scripting. It also suggests that there is room for improvement by studying why scripting helps (e.g. by reducing the number of calls between client and storage, or improvements in concurrency control) and how this can be replicated in the support provided by the module, when making a similar task through the language client.

- Regarding queries we studied a selection of queries from the ArangoDB benchmark. We found that Redis Graph, by offering index support, and through the specialized query language it offers for these operations, leads to the best results across all cases. Redis performs second best, but with a performance difference of at least 2 orders of magnitude, since it is not specialized for more analytical queries. Finally ReJSON shows the less efficient performance.
- From our results we should also note that we were not able to test a workload specially fitted to the characteristics of ReJSON.

In the next chapter we discuss related work to our study in the areas of benchmarking multi-model databases, some aspects of multi-model databases, and the related topic of relational databases supporting graph models and workloads.

6. Related Work

In this Chapter, we present an overview of the research done in the field of multi-model databases related to our work.

Literature shows that there has been some research done on various aspects of multi-model databases. Here, we consider three major domains and discuss the related work done in those fields. The fields that we consider are:

1. Relational databases supporting graphs
2. Multi-model databases
3. Benchmarking multi-model databases

6.1 Relational Databases Supporting Graphs

There are a number of application domains in which graphs play a very significant role. These include social and communication networks. Relational databases are leveraged in some applications to allow graph processing. The data of such applications can be considered as graphs with relational attributes on vertices and edges [35] or it can be viewed as relational data with a graph-like structure [36].Such applications tend to make use of both relational and graph queries at the same time. These graph-relational queries have two main parts; graph operations and relational predicates. An example of a graph-relational query would be the selection of some specific users from a relational table to find hospitals nearest to them using shortest path query over a road graph network [2].

The two main approaches, present in literature, that are used to build graphs-supporting applications on top of relational databases are:

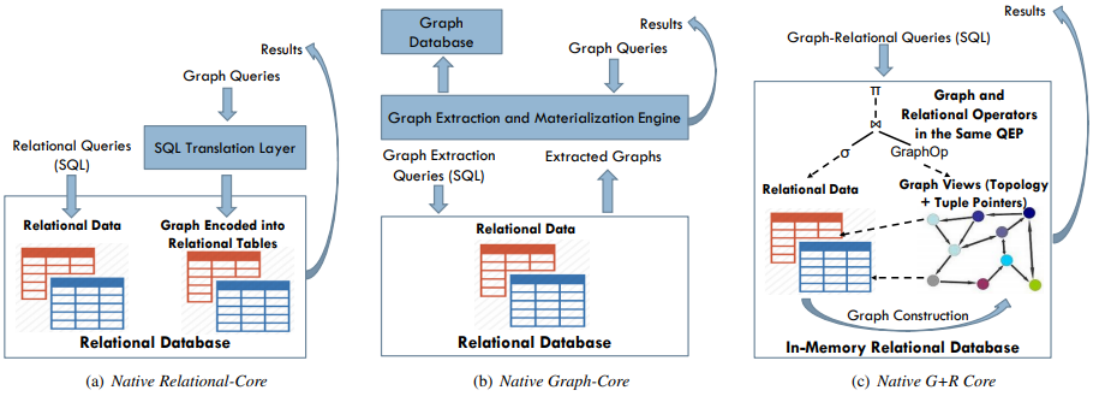


Figure 6.1: Different approaches of leveraging relational databases to support processing of graphs [2]

- **Native-Relational Core:** In the native-relational core approach (SQLGraph [35] and Grail [37]), a graph is embedded inside a relational database. An application built on top of the relational database, then, translates the graph queries into relational queries in order for them to be executed by the relational database. Although it is not easy to translate all graph queries into relational queries or SQL statements, tools are developed to automate this translation [2].
- **Native Graph-Core:** In native-graph core approach (Ringo [38] and GraphGen [36]), it is assumed that the relational database systems stores the graphs already. An application built on top of RDBMS retrieves these graphs from the relational database and assess them outside the scope of the relational database. This approach is more related to graph databases as in this case, RDBMS has nothing to do with graph analysis and execution of graph queries.

Hassan et al. [2] have proposed a hybrid approach and named it Native G+R Core that makes use of the strengths of the two standard approaches mentioned above. The relational database that they have used in their research in order to implement this hybrid approach is VoltDB [1]. By doing this, Hassan et al. [2] suggest a union of graph and relational databases in their research prototype, GRFusion. The three approaches allowing graph processing on top of an RDBMS are shown in Figure 6.1

In GRFusion, the relational database stores the graph. For example, rows of a table in an RDBMS contain vertexes and another table has edges stored in its rows. GRFusion allows users to describe and query graphs using a declarative language. These queries can be purely graph queries, purely relational queries or a mix of both i.e. graph-relational queries. The query engine then analyzes the queries and executes them. The architecture of their proposed query engine is shown in Figure 6.2.

Another research by Paradies et al. [3], a graph traversal framework, Graphite, is proposed for processing of graphs inside a relational database system. Figure shows

[1] https://www.voltdb.com/

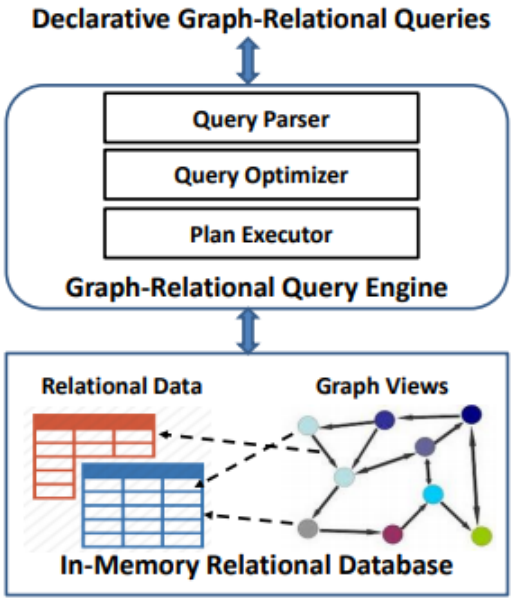
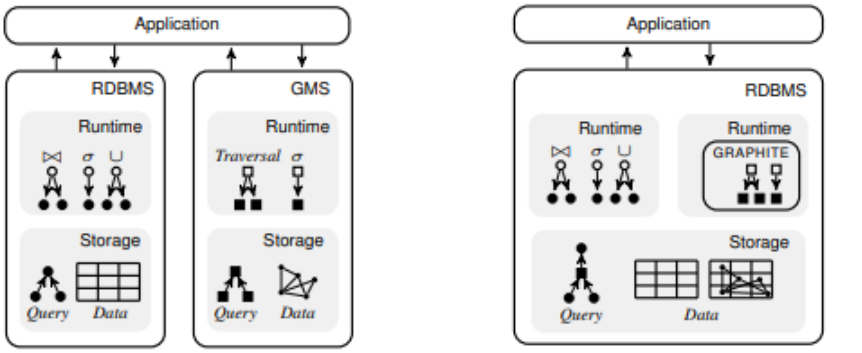


Figure 6.2: GRFusion's query engine architecture (This query engine allows processing of data in both relational and graph data models) [2]

relational database management system and graph management system with their specialized storage engines and processing units and also the integration of graph processing into relational database system as proposed in [3].

In this work [3], along with the graph-relational integrated framework, two different implementations of the traversal operator are also described. This work shows that implementation of graph processing and traversal over a relational database management system can be done efficiently on a common storage engine within the RDBMS. It also states that these integrated systems are competitive with specialized graph management systems [3]. The different architectures of graph processing with relational databases systems are shown in .

Jindal et al. [39] present a graph analytics tool built on top of a relational database management system. They name this tool Vertexica. Using this tool, along with SQL for purely relational systems, users can use a vertex-centric query language to execute purely graph or graph-relational queries. The basic idea behind this research is to inculcate user-friendly and good performance graph processing abilities in a relational database system.



(a) Separated RDBMS and GMS with specialized storage engines and processing via application.

(b) RDBMS with integrated graph processing capabilities and common storage engine.

Figure 6.3: Dierent architectures for processing of graphs [3]

6.2 Multi-Model Databases

The growing versatility of everyday data and its usage in different applications and enterprises has made the use of one database supporting a single data model difficult for users. A platform that lets users use multiple kinds of data at the same time in a single system is, hence, a better choice. Lu et al. [4]) give an overview of multi-model database management systems, study the past research on multi-model databases and highlight directions for future work in this field. They also discuss the challenges and opportunities in development of multi-model databases. According to this research [4], multi-model databases can be categorized into single-database and multi-database. In the latter one, multiple models are managed in a single engine and in the former one, more than one database system are working together, each handling its own data model. This categorization is shown in Figure 6.4.

According to this work, one of the main challenges in implementation of a multi-model database is selecting the method to store multiple distinct models. The second major challenge is the design or selection of a query language that is capable of accessing data from different data models. The third challenge is related to query optimization and evaluation.

In another research, Lu et al. [5] present the concept of UDBMS, a untied database management system, for managing multiple data models in one platform. This model,as stated in this work, aims to provide a unified data model and flexible schema for different kinds of data, a unified method for query processing, a unified index structure and also guarantees cross-model transactions. Furthermore, the challenges that can be faced while developing such a unified model are also discussed in this work. Two major challenges stated are:

- **Data Querying Challenge:** In a multiple model database, users have handle data of different kinds coming from different sources. As opposed to a single

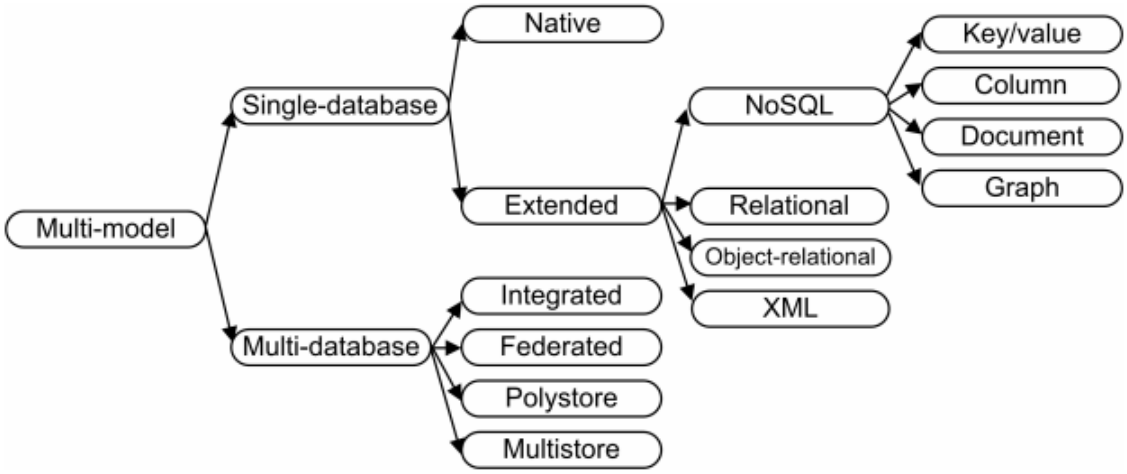


Figure 6.4: Classification of multi-model database systems [4]

database handling one data model with a specifed query language, this becomes a challenge as the multi-model database is not able to perform the complete set of queries available in specialized databases. Hence, a novel approach for querying of data in a multi-model database has to be established.

- **Data Consistency Challenge:** In a multi-model database, updating data can cause consistency issues. If the update has been made in one model and not the other, this will lead to consistency problems. Therefore, cross-model transaction guarantees have to be implemented in order to face this issue.

6.3 Benchmarking Multi-Model Databases

Benchmarking is an important technique to assess the performance of a system. With the growing number of databases proposing to handle multiple data models, it becomes very necessary to benchmark multi-model databases in order to choose the right one in terms of feasibility and usability. There are a number of benchmarks that are used to evaluate big data systems (e.g. YCSB, BigBench, TPCx-BB, Bigframe). But these are not suitable to benchmark multi-model database management systems. Jiaheng Lu has proposed a Unified Database Management System Benchmark (UDBMS-benchmark) [2] for multi-model databases. This benchmark consists of the following features:

- **Multi-Model Data:** Multiple data models are handled in this benchmark. Figure 6.5 shows an overview of the included data models.
- **Multi-model schema evolution:** The automation of schema evolution for multiple model data is also done by UDBMS benchmark.

[2] http://udbms.cs.helsinki./?projects/ubench

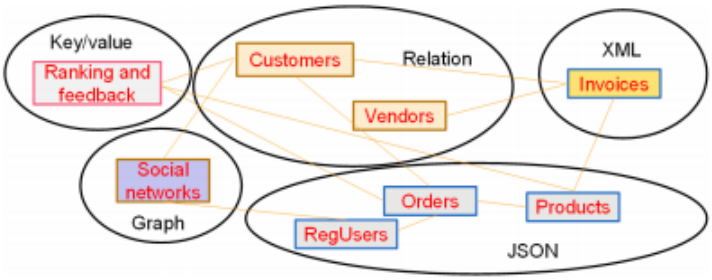


Figure 6.5: Data models in UDBMS benchmark [5]

- **Multi-model transaction and consistency:** Transactions in a multi-model database involve multiple data models. In order to ensure consistency, ACID rules and eventual consistency methods are employed.
- **Multi-model data conversion:** Ideally, a multi-model database should be able to convert data from one data model to another. Data generators, thus, should be able to support accurate outputs for data transformation tasks.

With these features included, UDBMS-benchmark allows a comprehensive evaluation of multi-model databases.

In this Chapter, we presented an overview of the related work. In Chapter 7, we conclude our work by providing a summary, threats to validity and future directions in which work could be done.

6.4 Summary

In this Chapter:

- We have discussed different works in literature related to this research
- The works considered in this section are relevant to tree main domains (Relational databases supporting graphs, multi-model databases and benchmarking multimodel databases) of this work

In Chapter 7, we will be presenting a conclusion of this work, highlight threats to validity and mention directions in which work can be done in future.

7. Conclusion

In this Chapter:

- We conclude our work in Section 7.1 by highlighting significant aspects of this research, discussing our findings and summarizing our approach.
- We present the shortcomings of this research and the threats to validity in Section 7.2
- Finally we highlight some areas from this domain, where work can be done in the future in Section 7.3

7.1 Summary

The growing volume and versatility of today's data has led to emergence of innovative and efficient ways of handling it. The choice of a single database catering to a single specific data model has become very difficult because applications and enterprises are making use of different kinds of data at the same time. This need to inculcate multiple data models in a single database engine has led to the rise of multi-model databases. A summary of our work done in this research is provided below:

- We follow the steps provided by the waterfall model to make our research reproducible and produce valuable artifacts.
- The purpose of this research is to analyze the steps needed to create and maintain a multi-model database system, evaluating the core tasks of data ingestion, data transformation and data querying that a multi-model database should be able to perform and providing insights on how a multi-model database could be made to work efficiently, helping to understand how the lifecycle of data could be improved.

- To carry out this research, we make use of Redis, a popular key-value store. Redis has extensible modules that enhance its capabilities to more than just key-value storage. We make use of two of the Redis modules, ReJSON for document data and Redis Graph for graph data, to explore its multi-model features.

- Three main evaluation areas are specified for which we carry out the three core tasks of data ingestion, data transformation across models, and data querying.

- Firstly, we assess the performance of Redis, ReJSON and Redis Graph, for **data ingestion** as to how each one of them performs when data is loaded into them. We have used OLTP workloads provided by YCSB for loading. It has been seen here that Redis performs much better than its modules for the specifics of loading.

- YCSB provides different workloads, each having different proportions of scan, read, write and update operations. Most of these workloads correspond to random access patterns, for which basic Redis is noted to perform better across most cases. Redis Graph's performance is the best when the operations are mainly scans. Other than this highly differentiated case, Redis performance is overall better than the modules.

- In the test of **data transformation**, we observe that coupling data access in one module with insertion in another module, can result in higher costs as compared to just loading data into the basic Redis module. To make data storage efficient, deletion of data in one of the modules can be done after the transformation to the other module, hence decreasing the amount of data stored. This was beyond the scope of our work.

- It is also observed that movement of data to ReJSON is more efficient as compared to moving it to Redis Graph. It is an observation that the use of scripting does indeed provide more efficient results. We show speedups 5.5x and 2.2.x. We consider that there might be room to improve the approach to scripting.

- Generally, it is expected that loading into Redis Graph would be more costly than loading into the alternative models, since in the former there is a need of maintaining an index. In our experiments with scripting for moving data across formats we noted that the performance of Redis Graph was able to surpass, by 1.06x that of ReJSON. These results suggest that by studying why scripting performs well (e.g. by reducing the calls between client and backend, or by being having complete control of the backend, in a single operation) it could be possible to determine the core cause for the improvement, and apply such techniques to improve the data load into Redis Graph.

- In the task of **data querying**, queries from ArangoDB benchmark are used. Overall, the performance of Redis Graph was shown to be the best among all modules in terms of data querying. It might be due to the index support it offers and the specialized query language it provides to carry out these operations.

Redis is the second best here, with a difference of 2 orders magnitude, as it is not specialized for more analytical queries, like Redis Graph. ReJSON performs least efficiently in this case. A reason could be that the workloads used did not fit specifically to the characteristics of ReJSON.

7.2 Threats to Validity

This section presents the threats to validity of our results and the shortcomings faced in this work.

7.2.1 Internal Threats

Some of the internal threats that may have affected our measurements are stated as follows:

- We applied the default properties to the workloads and did not explore and modify internal features of YCSB workloads while data loading. By making adjustments to the core workload properties, the results of data loading could be made better.
- Redis Graph still has a lot of areas where work needs to be done. By eliminating the limitations of the current version of Redis Graph, the research can be improved.

7.2.2 External Threats

Some of the external threats to validity that may have influenced the results in this research are mentioned as follows:

- As this research is only based on basic features provided by Redis and its modules, there can possibly be a lack of generalization faced in this work, pertaining to other databases and different implementations for the logical data models.
- Limited representativeness of the dataset used may have influenced our overall results. This may be improved by using different kinds of standard datasets.

7.3 Future Work

In this work, we have explored the multi-model features of Redis and have provided some insights on how an ideal multi-model database should work efficiently. However, naturally, there are many avenues left open for future work in this field and many areas where improved research could achieve better results. Using latest, more improved, versions of Redis and Redis modules will definitively enhance the performance of multi-model Redis. Furthermore, we have not explored all the features that Redis database provides (e.g. there are other modules like RedisSearch, which could allow to explore

more features of Redis' multi-model offerings). By adding more features of Redis in this work, this research can be made more comprehensive. Also, Redis provides other modules as well that we did not include in our work due to time limitations. More modules can be added to increase the diversity of the multi-model Redis client that we propose in this work.

We suggest that some future work should evaluate and understand the causes for data load into Redis Graph to improve with scripting, such that this configuration could be added to the normal load process.

More evaluation of ReJSON seems important, specially adopting workloads that match the features of such module.

We assert that work is needed in developing multi-model benchmarks such that the features of multi-model databases can be compared. We believe that work in this area could help in standardizing how a basic set of operations should be executed in different models (e.g. aggregation).

Finally, we consider that future work can build on how we evaluated the multi-model features of Redis, to propose techniques to manage the lifecycle of data, by loading it into a model, and transforming it (possibly in an incremental, rather than all-at-a-time manner) to other models, according to the workload to be served. This would enable developers of these databases to truly gain from the protean features in the multi-model databases.

8. Appendix

8.1 Appendix A: Code

In this section we include some code of our implementation, for reference:

8.2 YCSB Reads and Scans using Redis Graph

The following code corresponds to the reads and scans implemented in the YCSB client that we designed.

```
@Override
 public Status read(String table, String key, Set<String>
    fields,
     Map<String, ByteIterator> result) {
   Double zscore= jedis.zscore(INDEX_KEY, key);
   String script = "local ex = {redis.call('GRAPH.QUERY',
      KEYS[1], ARGV[1])} ";
   script+="local head=table.remove(ex, 1) return head";
   List<String> keys = new ArrayList<String>();
   List<String> args = new ArrayList<String>();
   keys.add("user");
   String query= "MATCH (n:User) WHERE n.key='"+key+"'
      RETURN ";
   if (fields == null) {
     query+="n.key, n.field0, n.field1, n.field2, n.field3,
        n.field4, n.field5, n.field6, n.field7, ";
     query+="n.field8, n.field9";
   } else{
     query+="n.key, "+fields.stream().map(entry->"
        n."+entry).collect(Collectors.joining(", "));
```

```
    }
    args.add(query);
   List<ArrayList<String>> response =
      (List<ArrayList<String>>) jedis.eval(script, keys,
      args);
   List<String> selectedFields = new
      ArrayList<String>(Arrays.asList(response.get(0).get(0).split(",")));
    if (response.get(0).size()>1){
      String valuesToParse= response.get(0).get(1);
      valuesToParse= valuesToParse.substring(1,
         valuesToParse.length()-1);
      List<String> values = new
         ArrayList<String>(Arrays.asList(valuesToParse.split("\",\"")));
      Iterator<String> fieldIterator =
         selectedFields.iterator();
      Iterator<String> valueIterator = values.iterator();
      while (fieldIterator.hasNext() &&
         valueIterator.hasNext()) {
        result.put(fieldIterator.next(),
             new StringByteIterator(valueIterator.next()));
      }
      assert !fieldIterator.hasNext() &&
         !valueIterator.hasNext();
      return Status.OK;
    }
    return Status.ERROR;

  }

  @Override
  public Status scan(String table, String startkey, int
     recordcount,
      Set<String> fields, Vector<HashMap<String,
         ByteIterator>> result) {
    Set<String> keys = jedis.zrangeByScore(INDEX_KEY,
       hash(startkey),
        Double.POSITIVE_INFINITY, 0, recordcount);
    HashMap<String, ByteIterator> values;
    for (String key : keys) {
      values = new HashMap<String, ByteIterator>();
      read(table, key, fields, values);
      result.add(values);
    }
```

```
    assert result.size()==recordcount;//We double-check this
       since we were uncertain on why the scan performed so
       well.
    return Status.OK;
  }
```

8.3 Moving Data into ReJSON

```
public static void convertToDocStoredProcedure(String
   graphName, String vertexTypeName, String
   relationshipName) {
                List<String> keys2Ship = new
                   ArrayList<String>();
        keys2Ship.add(graphName);
        List<String> args = new ArrayList<String>();
        args.add("0");
        for (int i=100; i<1000; i++){
                String script = "local hg = function (key)
                   local bulk = redis.call(\"hgetall\", key)
                   local result = {} local nextkey for i, v
                   in ipairs(bulk) do if i % 2 == 1 then
                   nextkey = v else result[nextkey] = v end
                   end return result end ";
                script+="local val=redis.call(\"keys\", \"v";
                script+=i;
                script+="*\") for i,v in ipairs(val) do
                   local bulk = hg(v)
                   redis.call('JSON.SET','doc'..v..'','.','"
                + "{\"id\":\"doc'..v..'\", "
                + "
                   \"description\":\"'..bulk[\"desc\"]..'\",
                   "
                + " \"region\":\"'..bulk[\"region\"]..'\", "
                + "
                   \"completion\":\"'..bulk[\"completion\"]..'\",
                   "
                + " \"age\":\"'..bulk[\"age\"]..'\", "
                + " \"gender\":\"'..bulk[\"gender\"]..'\", "
                + " \"public\":\"'..bulk[\"public\"]..'\", "
                + "
                   \"registration\":\"'..bulk[\"registration\"]..'\",
                   "
```

```
                + "
                   \"lastLogin\":\"'..bulk[\"lastLogin\"]..'\"}')
                   end return";
                jedis.eval(script, keys2Ship, args);
            }
        }
```

8.4 Queries for ArangoDB Benchmark with Redis Graph

```
GRAPH.QUERY pokec "MATCH (n:profile) RETURN n.age, COUNT(n);"
GRAPH.QUERY pokec "MATCH
   (s:profile)-[FRIENDS_OF]->(n:profile) RETURN n.id"
GRAPH.QUERY pokec "MATCH
   (s:profile)-[:FRIENDS_OF*1..2]->(n:profile) RETURN n.id"
GRAPH.QUERY pokec "MATCH
   (s:profile)-[:FRIENDS_OF*1..2]->(n:profile) RETURN n"
```

Bibliography

[1] Y. Abubakar, T. Adeyi, and I. Auta, "Performance evaluation of nosql systems using ycsb in a resource austere environment," *International Journal of Applied Information Systems (IJAIS)*, vol. 7, no. 8, 2014. (cited on Page xiii, 36, and 37)

[2] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi, "Extending in-memory relational database engines with native graph support," 2018. (cited on Page xiv, 57, 58, and 59)

[3] M. Paradies, W. Lehner, and C. Bornhövd, "Graphite: An extensible graph traversal framework for relational database management systems," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, 2015. (cited on Page xiv, 20, 58, 59, and 60)

[4] J. Lu and I. Holubova, "Multi-model data management: What's new and what's next?," in *EDBT*, 2017. (cited on Page xiv, 2, 3, 20, 22, 38, 60, and 61)

[5] J. Lu, Z. H. Liu, P. Xu, and C. Zhang, "Udbms: Road to unification for multi-model data management," *CoRR*, 2016. (cited on Page xiv, 19, 60, and 62)

[6] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013. (cited on Page 2, 10, 18, 19, and 20)

[7] A. Pavlo and M. Aslett, "What's really new with newsql?," *ACM Sigmod Record*, 2016. (cited on Page 2)

[8] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The bigdawg polystore system," *ACM Sigmod Record*, vol. 44, no. 2, pp. 11–16, 2015. (cited on Page 2)

[9] R. Lu, H. Zhu, X. Liu, J. K. Liu, and J. Shao, "Toward efficient and privacy-preserving computing in big data era," *IEEE Network*, 2014. (cited on Page 8)

[10] S. M. Wille and S. Scharf, *Indexing Nature: Carl Linnaeus (1707-1778) and His Fact-Gathering Strategies*. Department of Economic History, London School of Economics, 2009. (cited on Page 9)

[11] S. Rule, "A history of information technology and systems," *Information Technology*, 1917. (cited on Page 9)

[12] R. W. Taylor and R. L. Frank, "Codasyl data-base management systems," *ACM Comput. Surv.*, vol. 8, Mar. 1976. (cited on Page 9)

[13] D. D. McCracken, *A Simplified Guide to Structured COBOL Programming*. John Wiley & Sons, Inc., 1976. (cited on Page 9)

[14] I. M. Lewis, "Key-value storage systems (and beyond) with python," *The Python Papers*, 2010. (cited on Page 11)

[15] S. Manaa, "A survey on multimodel databases.". (cited on Page 13, 15, 19, 20, and 22)

[16] R. D. Padhy, M. R. Patra, and S. C. Satapathy, "Rdbms to nosql: Reviewing some next-generation non-relational databases," *INTERNATIONAL JOURNAL OF ADVANCED ENGINEERING SCIENCES AND TECHNOLOGIES*, 2011. (cited on Page 15)

[17] A. B. M. Moniruzzaman and S. A. Hossain, "Nosql database: New era of databases for big data analytics - classification, characteristics and comparison," *CoRR*, 2013. (cited on Page 15)

[18] G. M. Kuper and M. Y. Vardi, "The logical data model," *ACM Trans. Database Syst.*, 1993. (cited on Page 16)

[19] M. Levene and A. Poulovassilis, "An object-oriented data model formalised through hypergraphs," *Data Knowl. Eng.*, 1991. (cited on Page 16)

[20] J. J. Carroll and G. Klyne, "Resource description framework ({RDF}): Concepts and abstract syntax," 2004. (cited on Page 16)

[21] F. Holzschuher and R. Peinl, "Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 2013. (cited on Page 16 and 32)

[22] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The bigdawg polystore system," *SIGMOD Rec.*, 2015. (cited on Page 19)

[23] S. PIMENTEL, "Polyglot persistence or multiple data models?.". (cited on Page 19)

[24] L. Wiese, "Polyglot database architectures = polyglot challenges," in *LWA*, 2015. (cited on Page 19 and 20)

[25] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: All for one, one for all in data processing systems," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015. (cited on Page 19)

[26] D. Agrawal, C. Chawla, E. Ahmed, K. Zoi, O. Mourad, P. Paolo, Q. Jorge-Arnulfo, T. Nan, and M. J. Zaki, "Road to freedom in big data analytics," in *EDBT*, 2016. (cited on Page 20)

[27] A. Oussous, F. Z. Benjelloun, A. A. Lahcen, and S. Belfkih, "Comparison and classification of nosql databases for big data," in *Proceedings of International Conference on Big Data, Cloud and Applications.*, 2015. (cited on Page 23)

[28] Y. Abubakar, T. S. Adeyi, and I. G. Auta, "Performance evaluation of nosql systems using ycsb in a resource austere environment," *Performance Evaluation*, vol. 7, no. 8, pp. 23–27, 2014. (cited on Page 26)

[29] M. Paksula, "Persisting objects in redis key-value database," *University of Helsinki, Department of Computer Science*, 2010. (cited on Page 27)

[30] C. Branagan and P. Crosby, "Understanding the top 5 redis performance metrics," *Austin: Datadog*, 2013. (cited on Page 27 and 29)

[31] T. Macedo and F. Oliveira, *Redis Cookbook: Practical Techniques for Fast Data Manipulation.* " O'Reilly Media, Inc.", 2011. (cited on Page 27)

[32] I. Haber, "Redis graph, redisconf17.". (cited on Page 34)

[33] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International Scientific Conference & International Workshop, Present Day Trends of Innovations*, 2012. (cited on Page 35)

[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010. (cited on Page 36)

[35] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie, "Sqlgraph: an efficient relational-based property graph store," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015. (cited on Page 57 and 58)

[36] K. Xirogiannopoulos, V. Srinivas, and A. Deshpande, "Graphgen: Adaptive graph processing using relational databases," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, 2017. (cited on Page 57 and 58)

[37] J. Fan, A. G. S. Raj, and J. M. Patel, "The case against specialized graph analytics engines," in *CIDR*, 2015. (cited on Page 58)

[38] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec, "Ringo: Interactive graph analytics on big-memory machines," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015. (cited on Page 58)

[39] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker, "Vertexica: your relational friend for graph analytics!," *Proceedings of the VLDB Endowment*, 2014. (cited on Page 59)