Otto Zemzans

EXPLORING NOSQL DATABASES

Comparison of Databases

Bachelor's Thesis
Information Technology

May 2016

MAMK
University of Applied Sciences

DESCRIPTION

MAMK University of Applied Sciences	**Date of the bachelor's thesis** 30.05.2016
Author(s) Otto Zemzans	**Degree programme and option** Information Technology

Name of the bachelor's thesis

EXPLORING NOSQL DATABASES

Comparison of Databases

Abstract

SQL databases have been the mainstream choice for almost any project. They store data in a normalized, tabular structure and do many fancy things for you, such as providing very strong consistency across different database instances. This kind of feature is very useful, but sometimes developers want more freedom, even at the cost of consistency, for example. NoSQL databases can offer more flexibility and extra special features, which wouldn't work very well in SQL ideology.

The IT industry has been showing lots of interest in NoSQL solutions. The amount of data and special requirements have lead many companies to try and find something that SQL cannot solve. NoSQL can sometimes offer more flexible and convenient solutions The purpose of this study was to investigate the world of NoSQL and see how NoSQL databases can be compared between each other and SQL.

Three very different databases were chosen and installed in the practical part. Their basic usage was introduced and very representative example use cases were demonstrated to understand what kind of NoSQL solutions are available. General properties of the databases were researched to understand how one should objectively choose a database to work with.

As a result, a table depicting a way to compare the three NoSQL databases: Redis, Mongo DB and Neo4j. They have different features and distinct data paradigms. Redis is a key-value store. Mongo DB is a document store. Neo4j represents the data as a directed graph.

It was concluded that SQL is not going anywhere, but NoSQL enrich the possibilities of solving problems that SQL cannot handle at all or cannot handle in an elegant or optimal way.

Subject headings, (keywords)

database, nosql, nosql vs sql, redis, mongo db, neo4j, acid

Pages	**Language**	**URN**
39	English	Leave blank

Remarks, notes on appendices

Tutor	**Employer of the bachelor's thesis**
Matti Juutilainen	Mikkeli University of Applied Sciences

CONTENTS

1 INTRODUCTION

There will always be a need for more data. Every year the amount of digital data in the world increases in an almost exponential fashion. And, the trend is going to continue, because there will never be enough data. As humanity gets more absorbed into the digital world, the data produced per person has grown. Instead of being very private and isolated from the rest of the world as in the past, nowadays people have more than 20 online accounts on average and tend to spend a great deal of their life producing data. To push science forward, scientists model all kinds of processes which can always be more precise, which requires a lot of data. The needs for high quality video and sound have imposed huge requirements for storage capacity. And, the standards keep increasing. These and many more applications will never be perfect, and thus the data that is generated will grow a lot.

To quench this thirst for more data, hardware development has been pushed very hard to rapidly improve. Every year better devices emerge to process and store even more data. Even so, at any given point in time, there is a limit of how powerful a system can be built. It is important to also have adequate software, to further optimize the system. Traditionally, to structure the data, one would use a relation database like MySQL. They have very rich capabilities and many features. But, what if you want to squeeze every last bit of performance even at the cost of these capabilities. That is why, NoSQL has emerged.

The goal of this thesis is to explore the possibilities of NoSQL and how they can be compared, what criteria should be used when evaluating the type of a database. In the practical part three representatives of different NoSQL databases will be implemented.

2 SQL DATABASES

Knowing what SQL databases have to offer is an important step to understanding what NoSQL databases are about. This chapter will explain the basics of SQL.

2.1 SQL Introduction

SQL, Structured Query Language is just a highly abstracted programming language for data manipulation. Originally developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s, it nowadays has many different versions that are built upon it, such as MySQL, MS Access, Sybase, Informix, Postgres SQL and Oracle. Depending on the platform and requirements, one of them would be chosen. They all use the SQL which is more or less similar, but have different dialects of SQL. MS SQL Server has T-SQL dialect, Oracle has PL/SQL and MS Access has JET SQL. [1]

2.2 Features

SQL is highly optimized for providing relational data management. It requires you to define all of the data structure beforehand (schema). So, relationships between tables and column structure must be defined in the schema. Everything is stored in tables. One data entity is a row of a table (record). In general, data would be added by finding the right table and making a new row.

Relationships are defined as primary keys and foreign keys as shown in Figure 1. This allows for more separate tables, all data that is connected is not necessarily in a single table, but can be partitioned which increases extensibility and performance.

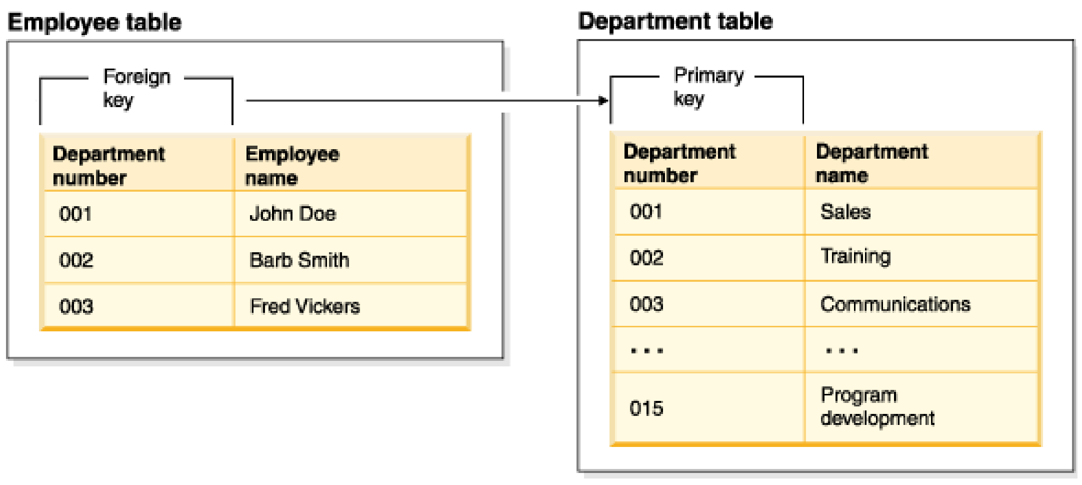


Figure 1. SQL relation example

SQL is quite intuitive to use. It is almost like plain English, and the commands can be separated into three categories as following:

- Data Definition

CREATE, ALTER, DROP

With these commands, tables and/or databases can be created, edited or destroyed. For example, the following query would create a table with id, name, address and age columns, and the primary key would be id.

```
CREATE TABLE USERS(
   ID INT NOT NULL,
   NAME VARCHAR(32),
   ADDRESS VARCHAR(64),
   AGE INT,
   PRIMARY KEY (ID)
);
```

- Data Manipulation

SELECT, INSERT, UPDATE, DELETE

With these commands row data can be retrieve, changed or deleted. For Example, the following query would select all users, with the age value of 18 and more. Displaying only id, age and address columns.

```
SELECT ID, AGE, ADDRESS
FROM USERS
WHERE AGE > 17;
```

- Data Control

GRANT, REVOKE

With these commands it is possible to manage user access.

SQL is very simple to use, but it has quite complex implementations behind the abstracted language. SQL promises high reliability with the atomicity, consistency, isolation and durability properties (ACID).

SQL has been around for a while, so it has had time to mature and build a huge community as well as a lot of highly sophisticated graphical tools to help with development, which is always a problem when moving to newer, less mature alternatives.

3 NOSQL DATABASES

This chapter will focus on showing what NOSQL databases have to offer and how they can compete with SQL.

3.1 NoSQL introduction

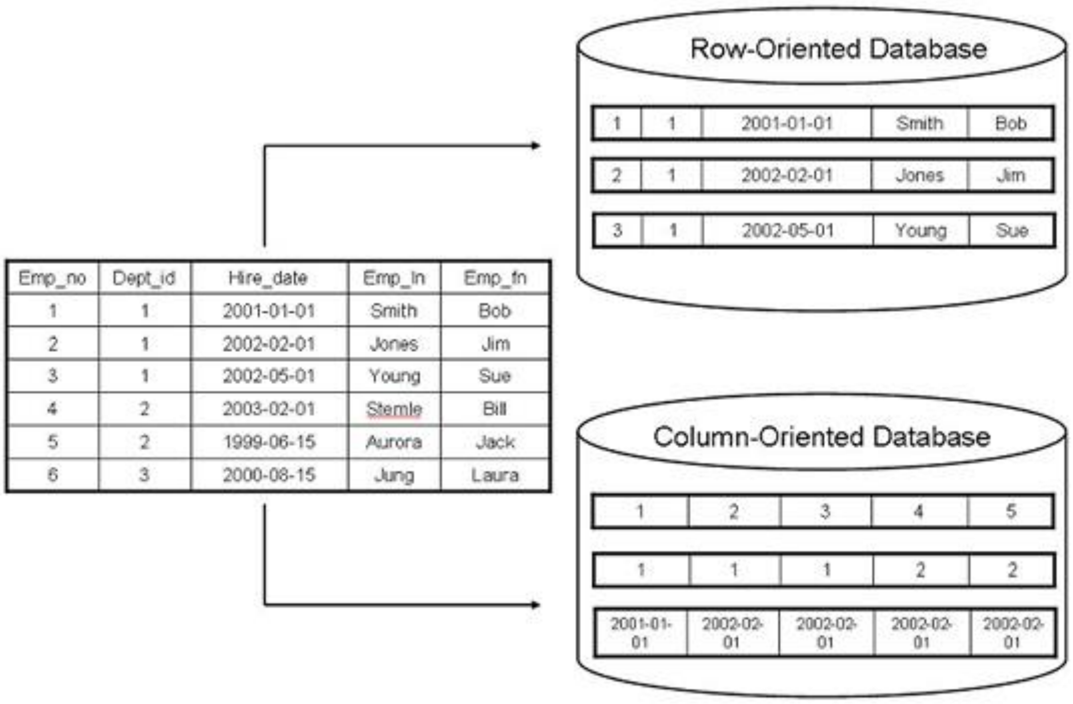
The term NoSQL was used first by Carlo Strozzi in 1998 for his Strozzi NoSQL open-source relational database. This database was relational, but the term has evolved and since 2009 it is more known for non-relational or non-ACID databases. The NoSQL databases vary from simple key-value stores to complex SQL like systems. That is how vague the term is. [2]

3.2 Types of NoSQL Databases

Since there is more freedom of implementation, there are also many more types of NoSQL databases which might vary greatly in features. A way to categorize NoSQL databases is related to their data model.

3.2.1 Column Databases

In SQL a single record or the smallest entity is a row of a table. But the column data model sees this as a limitation. The smallest entity in databases which utilize column data model, is a column. Instead of storing a single object’s properties in one whole entity, it stores the properties of many objects that belong to a column separately, as shown in Figure 2. This ensures that not all properties of an object need to be queried, but only those that you need. Column databases are generally suited for queries that don’t involve all columns, just some and when dealing with not just one row. Also, since the data in columns is very similar, each column can have its own compression algorithm, most suitable for the kind of data it stores. This data model is mostly based on schema, and therefore the data types are predefined.



Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura

1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue

1	2	3	4	5
1	1	1	2	2
2001-01-01	2002-02-01	2002-02-01	2002-02-01	2002-02-01

Figure 2. Row-oriented versus column-oriented data

Examples of such databases: Accumulo, Cassandra, Druid, HBase, Vertica.

3.2.2 Document Databases

In document databases the smallest entity is a document. It is a self-explaining representation of an object, therefore a document is like an SQL record. It is stored in some variation of JSON or XML, as shown in Figure 3. Each document has a unique key associated with it, so you could store a key reference to another document, establishing a kind of primary key and foreign key relation. As opposed to the column database, the document data type can be anything, it is schema-less, and any data type will be encoded in JSON or XML. Having this flexibility of not caring about the data structure is helpful for rapid development and changing data structures (fast application development). Not having to serialize data, but putting any kind of programmatical object in the document and retrieving it can be done much faster, and no serializing means a little bit less calculations. The store often indexes the data within a document, and it understands when only certain fields are needed, not everything.

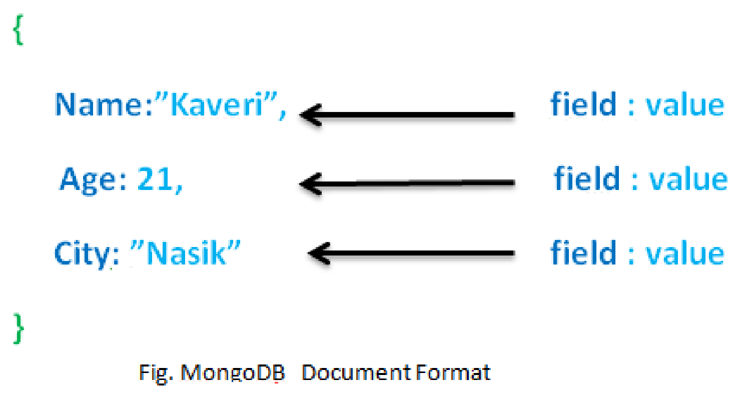


Figure 3. Document database object structure

Examples of such databases: Apache CouchDB, Clusterpoint, Couchbase, DocumentDB, HyperDex, Lotus Notes, MarkLogic, MongoDB, OrientDB, Qizx, RethinkDB.

3.2.3 Key-value Databases

Key-value databases are a superset of document databases, they are more basic. There are just key and value pairs, as shown in Figure 4, similar to document id and document data. But, the document databases have some relation capabilities and they may have some more advanced logic that understands what data is in a document. In key-value databases, this is not a priority. The database doesn’t care how data is related to each other. Parts of the value cannot be queried, just simply what was put inside, is returned. Not being very smart is both a disadvantage and an advantage. It improves the performance and flexibility of building a more highly abstracted logic without help. They are very often used as a caching layer. However, it will take more time to develop complicated data structures and relations.

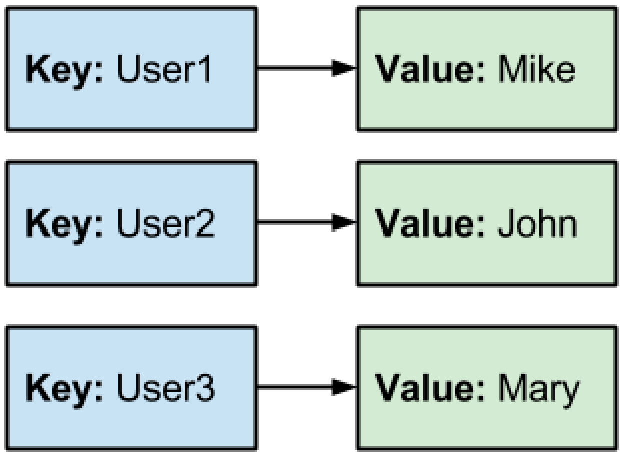


Figure 4. Key-value store data structure

3.2.4 Graph databases

Graph databases are relation driven. Looking at the data structure, it would be a directed graph in a mathematical sense. Compared to SQL, they are a little different though. By comparing a row of SQL data and an entity of a graph database, it can be seen that graph databases have a distinct difference. The data would consist of nodes and vertices, as shown in Figure 5. The nodes would have all the information about some object. And the vertices would represent the relations between objects. Vertices would have the verb name. For example, subject id: “1” knows an object with id: “2”. The point of this data model is to address the problem of having too many relations in the tabular data. People would end up writing very complex SQL queries to get data with many relation levels. Also analyzing such data as graphs is very natural for people to see relation patterns, which would be hard to notice in tabular data.

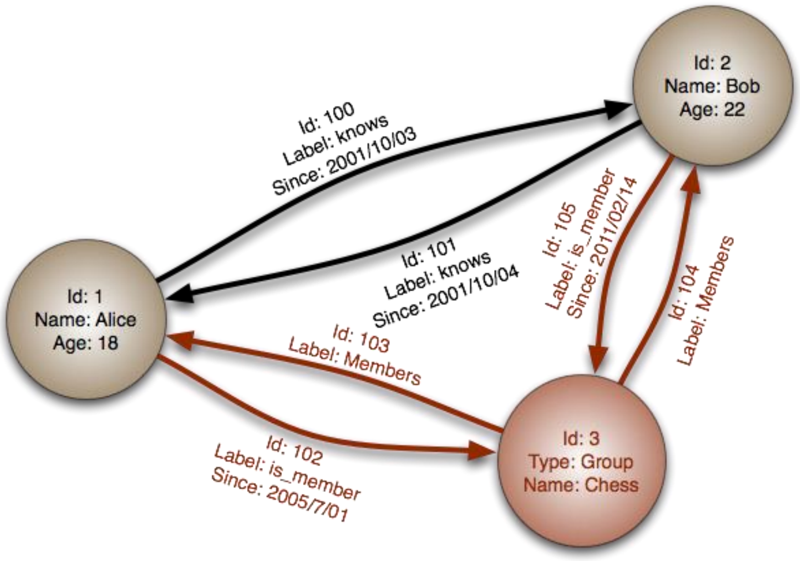


Figure 5. Graph database data structure

Example databases: AllegroGraph, InfiniteGraph, MarkLogic, Neo4J, OrientDB, Virtuoso, Stardog.

3.2.5 Multi-model databases

These databases can have the support for many data models. Sometimes one would want the benefits of different data models for the data, therefore integrating them all in one store is a possibility. The logic for each data model is working separately, while all of the components are connected to a single management system. Some of them also support custom, user defined data models.

Examples of such databases: Alchemy Database, ArangoDB, CortexDB, FoundationDB, MarkLogic, OrientDB.

3.3 Other Ways of Comparison

Comparing databases is not as simple as comparing their data model. It is important to also consider how ACID or BASE they are. Also CAP needs to be considered.

3.3.1 ACID

ACID stands for the following:

- Atomicity – guarantees that a given transaction is completed fully. In case of partial execution of a query, the query is considered a failure and it should not have any effect on the data. Often, due to system failure or power outage, database transactions can be interrupted. This guarantees that the data is not corrupted, even if it means that the good parts of the transaction are lost, too. Also, when multiple transactions try to write to the same place, it is only possible after completing or rolling back a transaction. This ensures that there are no conflicts. A way to do this is locking the data to be read-only until the transaction is finished. Keeping a log file for all of the transactions helps to know how to roll back, in case of failure. [3]

- Consistency – guarantees a consistent state of data. When data is being manipulated, it is either moved to a new valid state, or if an error occurs, it will roll back to the previous valid state. Therefore, no data can be lost. Every time the data is accessed, it is consistent and the same for all parties. In distributed database systems this would mean that all replicas of the data have the same data at any given point. [3]

- Isolation – guarantees that a transaction in process that is not yet finished is completely isolated from other running transactions. Therefore, the transactions should not influence each other while they are being processed. Only after a transaction is committed, another one is allowed to know of the previous transaction's results. [3]

- Durability – guarantees that even in the event of power loss, errors or crashes the data will still be in its last known valid state. Everything that is in the transaction log should be represented as it is in the data. Therefore, if the server crashes after a committed transaction, the transaction is guaranteed to be permanently written to the disk. [3]

It is important to know how ACID a database is. Leveraging these properties to find a perfect match is a challenge.

3.3.2 BASE

BASE stands for the following:

Basically available – This promises availability most of the time. Therefore, there is no 100% guarantee that a request to the database will have a response to whether a transaction succeeded or failed. [4]

Soft State – This means that the state of the system may change without any input. It is mostly due to eventual consistency. [4]

Eventual consistency – This indicates that the system might not be consistent at any given point in time. But it will try to be consistent eventually, when the data doesn't have to be precisely represented across the whole system. [4]

3.3.3 CAP theorem

The CAP theorem states that it is not possible to simultaneously provide all of the three following properties in a distributed database system:

Consistency – all of the database entities that have a copy of the same data are guaranteed to have the same data at any given point in time.

Availability – every transaction request has to get a response informing about the result of the transaction.

Partition tolerance – the system still works even if a part of it is isolated from other parts because of network failures.

At most, only two of these attributes can be true. Usually partition tolerance is a must in distributed databases. Network problems occur once in a while and relying on there not being any is too risky. Therefore, databases tend to choose either availability-partition tolerance or consistency-partition tolerance. In the case of availability the database system would respond to requests even in the situation of network failures, risking having old or bad data. But if the consistency route is chosen, the system would not respond at all in the case of network failure. Leaving the client to wait until everything is resolved. But, there would be no risk of inconsistent data. [5]

3.3.4 Comparison

Generally SQL tends to be extremely ACID, but NoSQL tends to follow BASE more. But, it is possible that NoSQL is completely ACID, just the data model is different. Therefore, NoSQL databases can be very different from SQL. They are a much larger group, as a consequence, there is more variety in ACIDity and BASEness.

4 SCALING

Different kinds of servers have different amount of possible simultaneous connections and how much data they can process. When a server reaches the limit to the maximum connections, or how much data it can deal with, it may be extremely slow and unresponsive. The worst case could even lead to crashes which might not be fixable remotely. Losing uptime is very expensive for businesses, in some cases even disastrous. To deal with this issue one can either scale up or scale out.

4.1 Scaling up (vertical scaling)

It is quite straightforward, to perform better server hardware is required. Scaling up is upgrading the hardware, adding more components, and increasing the amount of traffic your server can support.

4.2 Scaling out (horizontal scaling)

Scaling out involves adding more servers to your system. They usually share the load or handle a specific part of a system. It might be a bit difficult to set up a cluster initially. But adding more and more devices should be trivial afterwards.

4.3 Vertical versus horizontal scaling

Both approaches have a place in certain situations, the following table presents the differences (Table 1). SQL is mostly focused on scaling up, while NoSQL has more focus and built-in support for scaling out more easily. [6]

Table 1. Scaling vertically versus horizontally

	Vertical Scaling	Horizontal Scaling
Hardware costs	The best equipment can be very expensive	Hardware can be less powerful, but it is cheap to add new mid-tier servers
Software costs	No additional cost	License fees for a new machine

Space	No extra space needed	more space per server needed
Power consumption	Very little extra power used	More power used due to having more servers
Ease of implementation	Easy	Can be complicated
Capacity increases	Has a limit	Theoretically could have no limits

5 PRACTICAL PART

In this part I will explore three different kinds of NoSQL databases. They include the following:

- Graph database – Neo4j
- Key-value in-memory database – Redis
- Document database – MongoDB

5.1 Server set up

I chose Up Cloud to manage the server. It is an IaaS that is straightforward to use. After choosing a name, location, one of the hardware packages, storage devices and OS template. I ended up with the following configuration, introduced in Table 2.

Table 2. Cloud server configuration

FQDN server name	Database server
Availability Zone	London #1 (uk-lon1)
Disk image	Ubuntu Server 14.04 LTS (Trusty Tahr)
CPU cores	1
RAM	1024 MB
Disks	Drive A (30 GB)

After a few seconds, my server was deployed. I was given an IP address and root password for SSH access. As I was working from a Windows machine, I used Putty to SSH into the server.

5.2 Software Environment

I chose to use Node to test the databases. It is a JavaScript based cross platform framework for server-side scripting. It is heavily focused on the web, as are many NoSQL databases. It is very convenient for a developer to work with them with Node. For example, document driven databases often use JSON (JavaScript Object Notation) to receive data or even store it. There is no need to change the format. Also JavaScript is very nice to work with the HTTP protocol, and some databases might use that. To setup the environment, I installed Samba, Git and exposed some inbound TCP ports to receive incoming requests, which may not be necessary, but it is convenient.

I am working with SSH and Samba to use the SMB protocol to access Ubuntu file system from Windows. But, it is not wise to expose Samba shares to the public. Samba is not recommended for that because of possible security concerns. But for this experiment, I will still expose it.

5.3 Installing Node

To install Node, we first download a bash script and run it, which installs the required repositories for apt-get:

```
curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash –
```

Install Node with apt-get with

```
sudo apt-get install -y nodejs
```

Then, we have got Node which comes with NPM (Node Package Manager), which can be used to manage all dependencies for Node.

I will use Express framework for abstracting the mundane tasks like instantiating an HTTP server, serving files and organizing routes.

6 NOSQL DATABASE COMPARISON

This part will introduce some more practical aspects of NoSQL databases and try to compare, and draw conclusions.

6.1 Graph Database Neo4j

Neo4j is made by Neo Technology. It is an ACID compliant graph database. This is the most popular graph database as of March 2016. It is written in Java. Interestingly, it uses an HTTP endpoint to receive transaction data. This is a delight for developers who are already familiar with HTTP. [7]

Neo4j uses a query language called Cypher. It is somewhat similar to SQL, but the statement structure is much different to accommodate graph data queries. In addition, Neo4j comes with a web interface that can be accessed from the browser. It can display the graph data without a hassle. [7]

6.1.1 Installation

Neo4j 3.0.0 requires Java 8, install it as follows:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

This also includes downloading the database binaries:

```
wget -O ./neo4j.tar.gz http://neo4j.com/artifact.php?name=neo4j-community-
3.0.0-M05-unix.tar.gz
```

After installing Neo4j I had to change the password in the extracted directory `conf/neo4j.conf.` To run the database process, I had to run `./bin/neo4j con-sole.`

6.1.2 Usage

In Neo4j queries can be made very naturally, like matching a node Car that belongs to Bob. Then return something out of the data that was matched.

The basics of making nodes and relationships are shown in the following:

For example, creating a Node with a “Person” label, and properties name and title.

```
CREATE (n:Person { name: ‘John’, title: ‘Student’ })
```

To create a relationship between nodes, the nodes must be selected and then a directed relationship must be defined, as described in the following:

Matches any persons and assigns a variable a and b to matched persons.

```
MATCH (a:Person),(b:Person)
```

Targets specific persons with given properties name that is ‘Node A’ and ‘Node B’

```
WHERE a.name = 'Node A' AND b.name = 'Node B'
```

Creates a relationship with the given verb, in the ‘RELTYPE’. For example, LIKES, WORKS_FOR, FRIENDS_WITH. This could be any kind of word, but verbs make sense.

```
CREATE (a)-[r:RELTYPE]->(b)
```

And finally, the following returns the output result of the relation.

```
RETURN r
```

The basics of retrieving nodes and relationships, as introduced in the following:

The following matches all nodes with the label “Movie” and returns them as output:

```
MATCH (movie:Movie)
RETURN movie
```

And, the one below returns the title of the movie node that has any kind of directed relationship with the node with the name ‘Oliver Stone’:

```
MATCH (director { name:'Oliver Stone' })--(movie)
RETURN movie.title
```

Also a direction for the relationship could be used like `-->` or `<--`.

As a result, we can use the following:

```
MATCH (charlie { name:'Charlie Sheen' })-[:ACTED_IN]->
(movie)
<-[:DIRECTED]-(director)
RETURN charlie,movie,director
```

The previous snippet would find a node with the name ‘Charlie Sheen’ that has an **outgoing** relationship to another node where the relationships label is ‘ACTED_IN’. At this point we have a bunch of nodes which have an **incoming** relationship named ‘ACTED_IN’ directed from Nodes with the property name ‘Charlie Sheen’. Then, we find all the nodes which have an **outgoing** relationship with the name ‘DIRECTED’ to movie nodes.

6.1.3 Example Data Structure

A good use of graph databases might be relational product recommendations. Currently, Walmart is using Neo4j for product recommendations, because they have so much relational data, that they can optimize their data structure to be stored as a graph. There might be data between users, so some relationships could be mapped between the users and the products they are showing interest in.

A simple scenario could be as follows:

First, we create the user nodes:

```
CREATE (a:Person { name: 'John' }), (b:Person { name: 'Mary' } ), (c:Person
{ name: 'Reese' } )
```

Neo4j works with a RESTful api, therefore debugging the transactions is really easy. This simple query made two network requests:

- POST request to /db/data/transaction, to notify the start of a transaction.
- POST request to /db/data/transaction/6/commit to send the query and to confirm the end of transaction.

Now we create the product nodes:

```
CREATE
(a:Product { name: 'TV', price: '1000€' }),
(b:Product { name: 'Guitar', price: '100€' }),
(c:Product { name: 'A Book', price: '40€' }),
(d:Product { name: 'Laptop', price: '600€' }),
(e:Product { name: 'A Dress', price: '100€' }),
(f:Product { name: 'Power Tools', price: '70€' })
```

To specify the relationships, or past purchases in this case we need:

```
MATCH (a:Person { name: 'John' }), (b:Product { name: 'TV' }), (c:Product {
name: 'Laptop' }), (d:Product { name: 'Guitar' }), (e:Product { name: 'A
Book' })
CREATE (a)-[r:BOUGHT]->(b),
(a)-[r1:BOUGHT]->(c),
(a)-[r2:BOUGHT]->(d),
(a)-[r3:BOUGHT]->(e)
```

So we made the Person node with name "John" associate with some products via "BOUGHT" edges. As shown in the Figure 6.

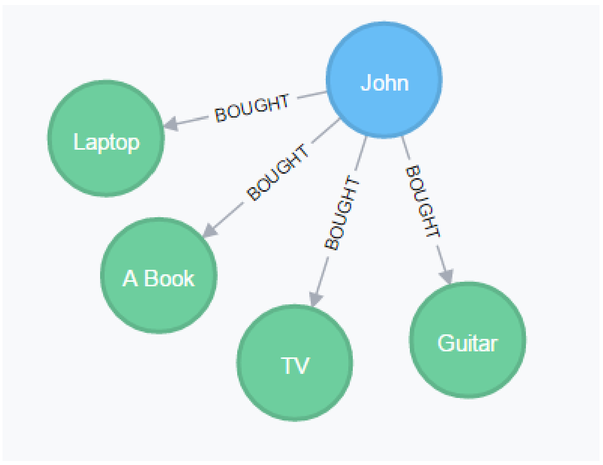


Figure 6. Simple BOUGHT relationship

Let us make some more relationships as follows:

Mary bought the following items (described as nodes):

```
MATCH (a:Person { name: 'Mary' }), (b:Product { name: 'A Dress' }),
(c:Product { name: 'A Book' }), (d:Product { name: 'Guitar' })
CREATE (a)-[r:BOUGHT]->(b),
(a)-[r1:BOUGHT]->(c),
(a)-[r2:BOUGHT]->(d)
```

And Reese bought the following:

```
MATCH (a:Person { name: 'Reese' }), (b:Product { name: 'Power Tools' }),
(c:Product { name: 'A Book' }), (d:Product { name: 'TV' })
CREATE (a)-[r:BOUGHT]->(b),
(a)-[r1:BOUGHT]->(c),
(a)-[r2:BOUGHT]->(d)
```

Now that we have some data, it is possible to see certain patterns, as shown in Figure 7.

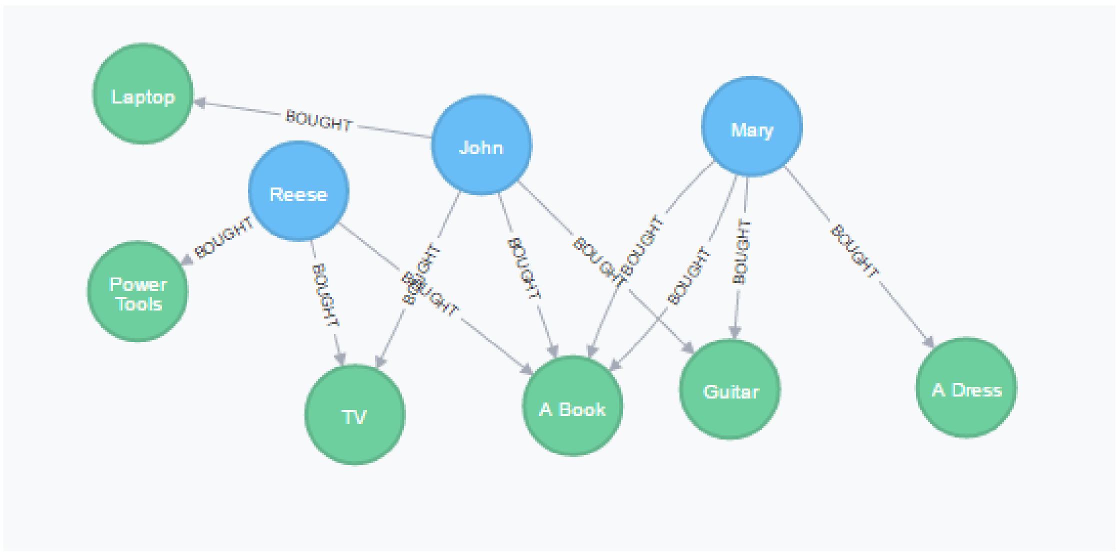


Figure 7. More Person nodes

Graph databases shine when there are lots of relationships, so let us add more as follows:

```
MATCH (a:Person { name: 'Reese' }), (b:Person { name: 'John' }),
(c:Person { name: 'Mary' })
CREATE
(a)-[r1:KNOWS]->(b),
(a)-[r2:KNOWS]->(c)
```

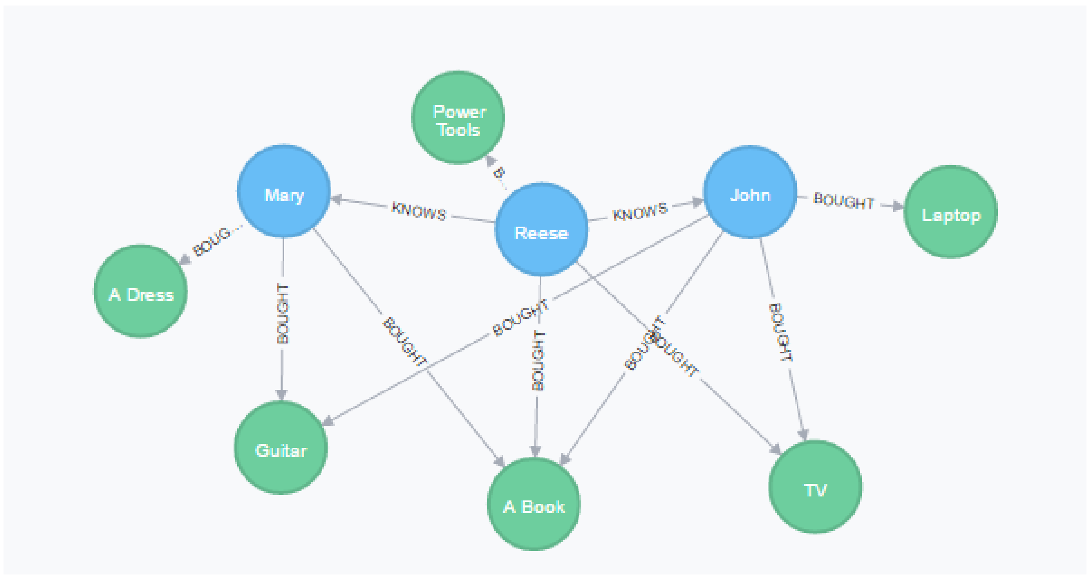


Figure 8. Relationships between Person nodes

Now, what if we wanted to get the products that Mary and the friends of her friends bought in the current schema, as shown in Figure 8. This is not a problem in Neo4j:

```
MATCH (mary:Person {name: "Mary"}),
(mary)-[:KNOWS]-(friend),
(friend)-[:KNOWS]-(friendOfFriend),
(friendOfFriend)-[:BOUGHT]->(items)<-[:BOUGHT]-(mary)
RETURN items
```

To do the same in MySQL, for example, something similar would be created in SQL, by first defining the schema:

```
CREATE TABLE USERS(
  ID INT AUTO_INCREMENT,
  NAME VARCHAR(20),
  PRIMARY KEY(ID)
);

CREATE TABLE PRODUCTS(
  ID INT AUTO_INCREMENT,
  NAME VARCHAR(20),
  PRIMARY KEY(ID)
);

CREATE TABLE FRIENDSHIPS(
  ID INT AUTO_INCREMENT,
  FRIENDEE INT,
  FRIEND INT,
  PRIMARY KEY(ID)
);

CREATE TABLE PURCHASES(
  ID INT AUTO_INCREMENT,
  USER INT,
  PRODUCT INT,
  PRIMARY KEY(ID)
);
```

and then inserting some data as follows:

```
INSERT INTO USERS (name) VALUES ("Person A");
INSERT INTO USERS (name) VALUES ("Person B");
INSERT INTO USERS (name) VALUES ("Person C");

INSERT INTO FRIENDSHIPS (friendee, friend) VALUES (1,2);
INSERT INTO FRIENDSHIPS (friendee, friend) VALUES (2,1);
INSERT INTO FRIENDSHIPS (friendee, friend) VALUES (3,1);
INSERT INTO FRIENDSHIPS (friendee, friend) VALUES (1,3);

INSERT INTO PRODUCTS (name) VALUE ("Product A");
INSERT INTO PRODUCTS (name) VALUE ("Product B");
INSERT INTO PRODUCTS (name) VALUE ("Product C");
INSERT INTO PRODUCTS (name) VALUE ("Product D");

INSERT INTO PURCHASES (USER, PRODUCT) VALUES (1,2);
INSERT INTO PURCHASES (USER, PRODUCT) VALUES (1,3);
```

Now, if we wanted to do the same kind of evaluation of the commonly purchased products between some user and the friends of his friends, it would get quite complex and slow. One example of this could be:

```
select * from PRODUCTS as p1 where p1.id in
  (select p.product from PURCHASES as p join
    (select q.name, f2.friend from FRIENDSHIPS as f2 join
      (select u1.name, u1.id from USERS as u1 join
        (select f.friend from USERS as u join FRIENDSHIPS as f
on u.id=f.friendee where u.name="Person A")
   as f1 on u1.id=f1.friend)
     as q on f2.friendee=q.id)
       as q2 on p.user=q2.friend);
```

This kind of join pattern is very non-intuitive to think about. It took me a long time to even make it work. And it is definitely much less performant.

When the dataset is large enough, it could take several seconds to query a nested join, but it might be 1,000 times faster for a graph database, because it is designed to work with connected data.

With more complex data, there could be more complex relationship queries and it would not matter. It is very easy and intuitive to add nested relation lines in Neo4j.

Of course, for simple queries that are not relational graph databases do not make any sense. It would be much slower to query rows. But where data is relationship centric, Neo4j is the way to go. Many enterprises try to integrate some part of their services with Neo4j, such as the social part of their websites.

6.2 Key-Value Database Redis

Redis is one of the most popular NoSQL databases. And most likely the most popular of its kind. It is an in-memory key-store database. It runs in volatile RAM storage, making data dumps once in a while in some persistent storage. Most of the time Redis is used as a caching layer, but sometimes it could even be used as a persistent database.

Of course, you may lose some data, but it might make sense in cases where it is not so critical to lose a few hours of data. [8]

Another competitor for memory-based caching key-store might be Memcached. But as of now, Redis has grown so much and offers many more features.

6.2.1 Installation

To install Redis, build-essentials and Tool Command Language are needed.

```
sudo apt-get install build-essential
sudo apt-get install tcl8.5
```

Let us download compressed Redis source code

```
wget http://download.redis.io/redis-stable.tar.gz
```

and uncompress the file as follows:

```
tar xvzf redis-stable.tar.gz
```

We now need to go inside the folder, compile and check the dependencies in the following way:

```
cd redis-stable
make
make test
```

After that, we install Redis system-wide

```
sudo make install
```

and run a script that lets Redis run as a daemon.

```
sudo ./utils/install_server.sh
```

Next, start the service with my designed port, 2911

```
sudo service redis_2911 start
```

and now Redis is accessible via

```
redis-cli
```

6.2.2 Usage

Redis might seem very simple, just a key-value store. But even as a key-value store, Redis has a lot to offer. As of April 2016, Redis offers nearly 200 commands, starting from straightforward database cluster support to fancy publisher/subscriber features. Also, it includes a built-in Lua interpreter, which can programmatically use Redis API. Redis supports the following data structures:

Redis Keys – This is an arbitrary string, even an empty string, which is used to map a key to a particular data structure.

Redis Strings – This is a string value.

Redis List – This is a linked lists, fast to add or get on/from head or tail, slow to read an arbitrary position.

Capped List – This is linked lists with a limit, for most recent additions. In other words, everything beyond a certain size would be gone. For example, seeing only the last 10 elements put inside the list.

Blocking List – This is a list that can also be used as a process-shared blocking queue. Where requesting values from an empty object will not fail, but will block the request and wait for data to arrive or timeout, such that polling is not necessary.

Redis Hash – This is an object with an arbitrary number of field-value pairs.

Redis Set – This is an unorganized set of strings, providing unique values and supporting set intersection, union and other set operations.

Redis Sorted Set – This is similar to sets. It provides uniqueness to the values, but also, the values are associated with floating point weights. These weights are used as a measure to sort the values. If the weights are the same, Redis uses lexicographical comparison.

Bitmap – It stores values as key-value pairs. But the value is described by an arbitrary binary number which can potentially be very memory efficient, when storing limited option values like 1 and 0 (the best case).

HyperLogLog – This is similar to a set, but it has the ability to, with an error margin of 1%, calculate the cardinality of the set, with a nearly constant time complexity.

As all keys in Redis are just plain strings, there is a convention to namespace them manually, such as “category:name”, for example. The most straightforward way of using Redis is just using plain strings. Normally, there would not be a need for anything more. Besides, it is possible to encode any kind of data structure in some format like JSON or XML as a string.

When writing a string, the following:

```
set person:name John
```

would simply set a key named “person:name” with the “John” value.

When making a list:

```
LPUSH lists:listname stringvalue
```

will find or create a list with the key “lists:listname” and push a value in front of the list.

To make it a capped list, we can apply:

```
LTRIM lists:listname 0 10
```

where “lists:listname” is the key for a list that exists. 0 is the start of the index and 10 is the end of the index that shows which part of the list should be preserved. This example will have 11 maximum elements at one time.

To make a hash we can use the following:

```
HSET hashkey name Bill age 25 gender male
```

where “hashkey” is the key and the value can be represented, as shown in Table 3.

Table 3. Redis Hash tabular data

name	Bill
age	25
gender	male

To make a set we can use the following:

```
SADD setkey setvalue
```

where “setkey” is the key and “setvalue” is the value.

When making a sorted set the following:

```
ZADD setkey 1.532 value1 2 value2
```

would make a sorted set with in the key “setkey”, with two weighted values which can be floats or integers. Querying this would not require the iteration of all of the set keys in the worst case at the expense of the extra memory used to store the weights.

It is important to know that, because Redis limits bitmap size to 512MB, the offset values can be from 0 to $2^{32} - 1$. To make a bitmap the following is needed:

```
SETBIT keyname 15 1
```

where “keyname” is the key for the bitmap, 15 is the offset and 1 is the value. Value can only be 1 or 0. After creating a bitmap, flipping bits is very quick, and counting the flipped bits as well, since many CPUs have hardware level support for operations with raw bits.

Redis is optimized for reading fast, with implicit indexes. Querying an arbitrary key will have a constant time complexity. However, there are no relations in Redis, and making them is up to the developer. Reading different data is similar to writing.

For example, when reading a string:

```
GET person:name
```

would simply get “person:name” key’s value.

And, reading from a list:

```
LRANGE lists:listname 0 -1
```

will return all the values from the list “lists:listname” starting from 0 to the last index. It is much more efficient to read from either the end or the beginning of the list.

For reading from a hash we need:

```
HGETALL hashkey
```

where “hashkey” is the key, of which all of the fields and values will be returned.

Reading from a set takes place with the following:

```
SMEMBERS setkey
```

where “setkey” is the key. It will return all of the values in the set.

From a sorted set the following:

```
ZRANGE setkey 0 -1
```

would find the elements from the index 0 to the last index in the sorted set.

To read a bit from a bitmap the following:

```
GETBIT keyname 15
```

can query the 15th bit from the bitmap that is associated with the “keyname” key.

6.2.3 Example usage

Even though Redis might be seen as a caching solution mostly. It is possible to make Redis persistent as well. It will still utilize memory, but it will also have mechanisms to reload data. As of now there are two of them:

- RDB – makes database dumps at certain intervals.

- AOF – records all transactions, optimizes them in the background and replays them upon starting the database.

It is possible to use none of them or both of them. But each of them will add some extra memory costs and a little amount of processing costs.

But the most common way of using Redis is indeed a caching layer, if we wanted to make a proxy service, for example, as shown in Figure 9.

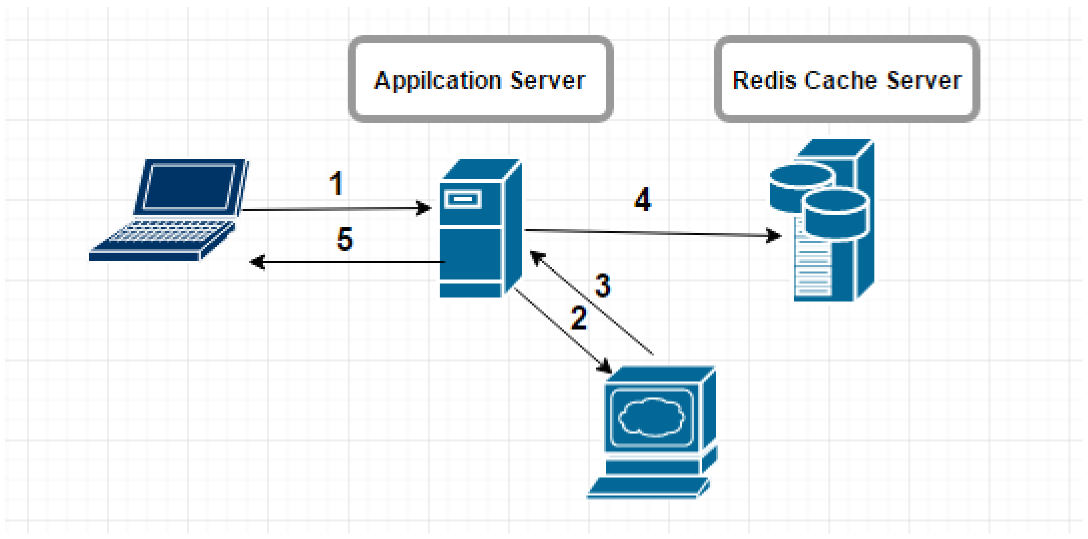


Figure 9. Without hitting the cache

The following describes the proxying process:

1. The client requests a website via the “Application Server”.
2. “Application Server” requests the contents of the requested website.
3. The resource responds with the HTML content.
4. The Application Server saves it in Redis.
5. The Application Server returns the requested content.

After caching the site, the request could be as shown in Figure 10.

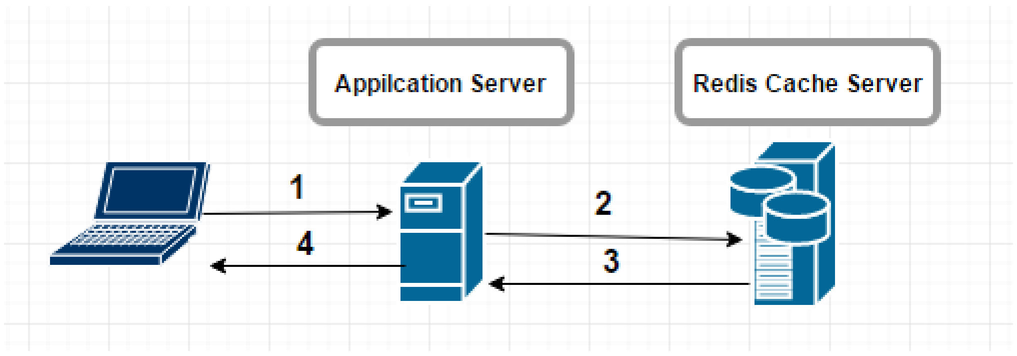


Figure 10. Cache hit case

1. The client requests a website via the “Application Server”.

2. Application Server asks Redis Cache Server for given site contents.

3. Redis Cache Server responds with a match and the cached contents.

4. The Application Server returns the cached contents.

This way, if the data is more or less static, the Application Server returns the proxied site with just one request, if we do not consider the requests to Redis, because that should be very optimized and fast for the connection with the Application Server. Even if the data is not that static, the cache time to live could be adjusted to account for it.

The pseudo code of the application server could be the following:

```
// simple http server
listen for requests on some port

// run this when somebody requests this
on request (request, response) {
    // send request parameter ‘getUrl’ to redis,
    // check if a key with that name exists
    if ( redis(“EXISTS”, request.getUrl) ) {
      // return the cached response
      response.return( redis(“GET”, request.getUrl) )
    } else {
      // retrieve the content from given url
      site content = curl ( request.getUrl )
      // set the record with url as key and content as value in redis
      redis(“SET”, request.getUrl, content )
      // return content
      request.return( content )
    }
  }
```

The Redis part of the pseudo code is very simple. It would save an additional roundtrip to some web server, if the cache gets hit.

6.3 Document database MongoDB

MongoDB (from humongous) is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of the JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster. MongoDB is developed by MongoDB Inc, and is published as free and open-source software under a combination of the GNU Affero General Public License and the Apache License. As of July 2015, MongoDB is the fourth most popular type of database management system, and the most popular for document stores. [9]

6.3.1 Installation

To install Mongo DB, we need to import the MongoDB GPG public key, so we can use the package manager securely:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
```

Then, we ad sources list file for MongoDB dependencies in “/etc/apt/sources.list.d” file

```
echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multi-
verse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
```

Finally, we install the latest version of MongoDB from the newly added repository:

```
sudo apt-get install -y mongodb-org
```

and start the MongoDB service with:

```
sudo service mongod start
```

6.3.2 Usage

MongoDB has a way to group documents, called collections. They are separately indexed, and could be thought of like of an SQL table. The query target is a collection. To interact with MongoDB, it is possible to use Javascript syntax.

In Mongo, writing operations are “insert”, “update” and “delete”. Inserting a new document is as simple as follows:

```
db.mycollection.insert (
  {
    name: “John”,
    age:  25,
    companyId: 231
  }
)
```

where “db.mycollection” targets a collection called “mycollection” in the currently selected database. Updating a document could be done by updating or replacing a document as follows:

```
db.mycollection.updateMany (
  { age: { $gt: 18 } },
  { $set: { companyId: 0 } }
)
```

where the first JSON object passed in is the filter criteria. In this case, all objects in the collection with the age field greater than 18 would be selected. The second object is the modification object. It is declared with $set, and will replace all fields inside the following object to the given values.

Deleting a document is very similar. Either we use “deleteOne” or “deleteMany” with a given filter criteria:

```
db.mycollection.deleteOne (
  { name: “John” }
)
```

which would delete the first document that has a name field with the value “John”.

6.3.3 Reading

One very important difference from the way MongoDB queries data is that there are no primary/foreign keys, therefore transactions work for a single collection at a time. Of course, it is possible to try to replicate a similar relationship pattern by embedding references to documents from a different collection within a given document. But, it might be quite difficult to make a robust relationship logic from scratch.

To read some data, a client must pass the query criteria within a given scope of a collection. And on top of that, a projection could be passed. A projection is a limited representation of a document, according to the requested fields. It is similar to selecting only the required columns in SQL.

To execute a query and to get the results the “find” method is used as follows:

```
db.mycollection.find (
  { name: “John” }
)
```

This would return all the documents in “mycollection” that have the name field with the value “John”. Additionally, a projection could be added in the following way:

```
db.mycollection.find (
  { name: “John” },
  { name: 1}
)
```

where the second parameter is the projection, which tells which fields should be included in the response. Here, the name field only would be included. In addition, projections can be exclusive, as follows:

```
db.mycollection.find (
  { name: “John” },
  { name: 0}
)
```

Here, the name value in the projection parameter is 0. This means that every field should be returned except for “name”.

6.3.4 Example usage

MongoDB could be used to replace SQL, which might be a good idea in some cases. But, how would relational data be served without joins? This example will show an implementation that is almost equivalent to SQL joins.

Let us have a very common scenario with a simple JOIN of a customer and products. There is a user who has bought some products. In SQL we would typically have a user database which stores all of the user information, as shown in Table 4. And there would be a product database which stores all of the product information, as shown in Table 5.

To join these two unrelated databases, there might be a purchase database, much like in Table 6. It would map the id of the user to the product of a user. It might also be a transaction and store some additional information.

Table 4. User table

id	name
…	…
23	John

Table 5. Product table

id	name
…	…
45	Laptop
46	Tablet
47	Smart Phone

Table 6. Transaction table

id	user_id	product_id
…	…	…
4	23	45
5	23	46
6	23	47

In SQL, we would just use the JOIN operation with the user_id and product_id from the transaction table to connect user and product table data.

To implement something equivalent in MongoDB, the Mongo client user interface will be used. To run it, we first execute:

```
mongo
```

Let us make a collection for each of the tables:

```
db.createCollection("users");
db.createCollection("products");
db.createCollection("transactions");
```

MongoDB generates an identifier for each document automatically, but we may also enforce our own id. To get a reference to your object easily, we can use a helper function ObjectId. It generates an identifier like MongoDB would automatically do, but we can get it before making the document.

Let us next assign a variable to the identifier that we will use to create a user document as follows:

```
john_id = ObjectId()
```

Next, we create a user in the “users” collection with the generated identifier to enforce our own id as follows:

```
db.users.insert(
  {
    _id: john_id,
    name: "John"
  }
)
```

Now we can create the product identifiers in a similar fashion:

```
laptop_id = ObjectId();
tablet_id = ObjectId();
smart_phone_id = ObjectId();
```

and insert the products:

```
db.products.insert(
   [
     { _id: laptop_id, name: "Laptop" },
     { _id: tablet_id, name: "Tablet" },
     { _id: smart_phone_id, name: "Smart Phone" }
   ]
)
```

And now we put those references to use for creating transactions in the following way:

```
db.transactions.insert(
   [
     { user_id: john_id, product_id: laptop_id },
     { user_id: john_id, product_id: tablet_id },
     { user_id: john_id, product_id: smart_phone_id }
   ]
)
```

Since MongoDB only supports queries with one collection at a time, there will be no joins, just sequential queries. For example, if we want a typical relationship with a given user name (“John”) to what he has purchased, the following is one approach:

We first find John’s id:

```
johns_id = db.users.find({name: "John"}, { _id: 1 })[0]._id
```

We then use the name as the filter and id as the inclusive projection. Since find returns an array-like object, we can get the first element. Then accessing the "_id" value from the object is trivial.

Now that the id is known, it can be used in "transactions" collection to find all of the relevant transactions with the following:

```
var johns_transactions = db.transactions.find({ user_id: johns_id }, {
product_id: 1 }).toArray()
```

which will also return an array of transactions. Therefore, for each of them, there has to be a query to get the referenced product names.

As a result, we can get the product_ids for each of the found transactions as follows:

```
product_id_1 = johns_transactions[0].product_id
product_id_2 = johns_transactions[1].product_id
product_id_3 = johns_transactions[2].product_id
```

This should be done in a more programmatically way, but this works for the experiment.

After that we can get the product names from the product collection:

```
db.products.find(
  { _id:
    { $in: [product_id_1, product_id_2, product_id_3] }
  }
)
```

Finally, we find all of the products which have an id that is either one of the products in johns_transactions. This may not be the optimal solution to try to do SQL-like JOINS independently in MongoDB, but it is possible, if there really is a need for it. However, the flexible, schema-less document structure allows for more performant strategies, such as simply embedding dependent relationship data. This might lead to more duplication in data, but to faster reads. Either way, MongoDB is a flexible solution to take on even the heavyweight SQL in many scenarios.

6.4 Comparison

Researching the properties and utilizing the NoSQL database management systems has enabled a more objective approach evaluating the databases. As seen in Table 7, most of the relevant properties that can be used to evaluate these databases are listed there.

Table 7. Comparison Table

Name	Redis	Neo4j	Mongo DB
Description	In-memory key-value store	Graph database	Document store
Developed by	Salvatore Sanfilippo	Neo Technology	MongoDB, Inc
Release year	2009	2007	2009
Implemented in	C	Java	C++
Supporting OS	Windows, BSD, Linux, OS X	Windows, Linux, OS X	Windows, Linux, OS X, Solaris
Schema	Schema-less	Schema-less	Schema can be optionally added
Query language	Low-level commands	Cypher – powerful queries like SQL	Powerful JS based queries, no joins or multi-collection queries
Number of client languages	33+	11+	27+
ACIDity	Atomic transitions, and manual (optimistic locking) locking in addition. Eventual consistency. Not very durable by default. But can be configured to be via logs/snapshots.	Totally ACID. Immediate consistency by default, but eventual consistency can be configured in HA cluster. However, this requires Enterprise edition.	Atomic operations available only on any single document. Eventual consistency model by default, but this can be changed to immediate consistency.
Built-in Replication	Master-slave	Master-slave (only in enterprise)	Master-slave (can chain slaves)
Data	Key-value mappings, supporting strings, lists, capped lists, blocking lists, hashes, sets, sorted sets, bitmaps, hyperloglogs. Doesn't have key grouping.	Data is stored as nodes with a label. And vertices. Internal data for nodes and vertices can be stored as a boolean, byte, short, int, long, float, double, char or a string.	Data represented in JSON documents. Therefore, supporting data types that JSON format supports. Data grouped in collections of documents.

7 CONCLUSIONS

As experienced firsthand, NoSQL databases can be very different from each other, despite being bundled together. Researching about basic database properties and requirements has helped me understand the rising popularity of NoSQL.

Before writing this thesis, I was clueless about NoSQL. I was stuck in the old SQL paradigm of “one size fits all”. Of course, SQL is not going anywhere. But, as a developer I have gained a broader view of the database world. When designing my next project, I will surely have more flexibility in choosing the best option and better judgment to evaluate the best tool for the job. Looking back to my experience with SQL, I can see many places where it would have been wiser to choose a different approach than SQL, because of ease of development, for example.

Results revealed that NoSQL databases can be quite different and target specific problems in the IT industry as opposed to SQL. The fact that specific tools perform better in their own specialty is a given. Therefore, integrating many different databases with separate roles is optimal in order to solve problems in the best way.

In reality, there is no concern about performance of SQL for most of the world’s projects. Deciding for or against SQL should not be based on that, but rather convenience, knowledge about the database, platform support, language preference and sentiment. NoSQL starts making a big difference when there is really a lot of data involved or the data has to be served in real-time.

BIBLIOGRAPHY

1. Chapple, M. (2015). SQL Fundamentals. Databases. Available at: http://databases.about.com/od/sql/a/sqlfundamentals.htm Retrieved. 28.02.2016.

2. Rouse, M. (2011). NoSQL (Not Only SQL database). Available at: http://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL Retrieved. 28.02.2016.

3. Barry & Associates, Inc. (2016). Service Architecture. ACID Properties. Available at: http://www.service-architecture.com/articles/database/acid_properties.html Retrieved. 10.03.2016.

4. Pritchett, D. (2008). File Systems and Storage. Base: An Acid Alternative. Available at: http://queue.acm.org/detail.cfm?id=1394128 Retrieved. 20.03.2016.

5. Browne, J. (2009). Brewer's CAP Theorem. The kool aid Amazon and Ebay have been drinking. Available at: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem Retrieved. 25.03.2016.

6. Shalom, N. (2010). Scale-out vs Scale-up. Available at: http://ht.ly/cAhPe Retrieved. 27.03.2016.

7. Neo Technology, Inc. (2016). Neo4j Documentation. Available at: http://neo4j.com/docs/ Retrieved. 03.04.2016.

8. Sanfilippo S. (2016). The Redis Documentation. Available at: http://redis.io/documentation Retrieved. 10.04.2016.

9. MongoDB, Inc (2008-2016). The MongoDB 3.2 Manual. Available at: https://docs.mongodb.com/manual/ Retrieved. 16.04.2016.