UiO : Department of Informatics
University of Oslo

NoSQL Databases in the Enterprise

An Experience with Tomra's Receipt Validation System

Davlet Dzhakishev

Master's Thesis Autumn 2014

NoSQL Databases in the Enterprise
An Experience with Tomra's Receipt Validation System

Davlet Dzhakishev
Institutt for informatikk, University of Oslo
Ole Johan Dahls Hus
Gaustadalléen 23 B
N-0373 OSLO
Norge
davletd@ifi.uio.no

5th November 2014

Abstract

The information processing demands of many of today's businesses have outgrown the legacy relational database management system (RDBMS) software resulting from both the data explosive growth and the variety of data models. Today, businesses must manage increasingly large volumes of data that must be available across distributed systems and as well as able to evolve and adapt data models according to the changes of business requirements.

Enterprises across all industries are challenged by the task of ensuring scalability of massive quantities of data while at the same time keeping database models simple and flexible at the same time. A new and advanced set of software, "NoSQL" as it is so called, has emerged in response to this challenge and offer new methods for storing data. The NoSQL ecosystem has flourished, with numerous software contributions appearing under the NoSQL umbrella. However, as more enterprises have implemented NoSQL solutions, a distinctive set of criteria has emerged that can help today's IT professionals more easily make use of NoSQL solutions built for enterprise-wide development.

In this thesis we investigate the usage of NoSQL solutions in the enterprise environment, where RDBMS traditionally run the show. Tomra AS provided us with the industrial case, featuring implementation of the distributed system for validation of receipts, using two different NoSQL databases. In this work, we describe implementation of the receipt validation system and evaluate NoSQL solutions, based on variety of criteria, such as performance, scalability and ease of use. We also describe the advantages NoSQL approach has over SQL (RDBMS) approaches in a distributed environment. The goal is to help decision makers in the enterprise to make better informed judgements when choosing a particular set of database software for their data handling strategies.

Acknowledgments

First I would like to thank my supervisor, Sagar Sen, for his excellent guidance and advice. I appreciate his involvment and passion for research. He has been inspiring curator with high expectations, that helped me to push for better results.

I would like to thank my second supervisor, Magne Jørgensen for taking his time and giving a valuable feedback on my thesis.

I would also like to thank Tomra AS, namely Christian Hovde for the great opportunity to use their industrial case as the topic for my theis. I thank my former colleagues Erik Drolshammer and Bård Lind for the opportunity to learn from them and work on the mentioned project together.

Finally I would like to thank my family, friends and especially Morgaine Wood for motivating and helping me. This thesis would have never been finished if it was not for the support from all of you.

Thank you!
Davlet Dzhakishev
November 2014

Contents

I Introduction and Background **1**

1 Introduction **3**
1.1 Time of explosive growth of non-relational data 3
1.2 Not only SQL data storage 4
1.3 NoSQL trends 5
1.4 Motivation 6
1.5 Problem statement 7
1.6 Industrial case with Tomra AS 8
1.7 Structure of the thesis 8

2 Background **11**
2.1 Introducing NoSQL 11
2.1.1 Criticism of NoSQL 13
2.1.2 Types of the NoSQL databases 14
2.2 MongoDB: A Document-oriented database 19
2.2.1 MongoDB competitive features 20
2.3 Neo4j: A graph database 22
2.3.1 Neo4j competitive features 23
2.4 Related work 24
2.4.1 Fundamental research of NoSQL 24
2.4.2 Comparative studies 25
2.5 Case studies 27
2.5.1 Neo4j case studies 27
2.5.2 MongoDB case studies 28
2.5.3 Summary 29

3 Research Method **31**
3.1 Qualitative and quantitative research methods 31
3.2 Data collection in the thesis 32
3.2.1 Comparative research 32
3.2.2 Secondary source research and data analysis 33
3.2.3 Case study research 33
3.2.4 Participant observation and fieldwork 34
3.3 Scientific method 34
3.4 Development research 35

II Industrial Case, Implementation and Evaluation **37**

4 Industrial Case **39**
4.1 Tomra Systems ASA Industrial Case 39
4.1.1 Problem description 39
4.1.2 Solution proposal 41
4.1.3 Disclaimer 41

5 Implementation **43**
5.1 Overall project architecture 43
5.1.1 Receipt Validator 45
5.1.2 Machine Network 47
5.1.3 Performance requirements 47
5.2 Relational Implementation. Legacy approach 48
5.2.1 Monolithic architecture 48
5.2.2 Data storage for Receipt Validator 48
5.2.3 Implementation of Receipt Validator 50
5.2.4 Data storage for Machine Network 52
5.2.5 Implementation of Machine Network 52
5.2.6 Summary 53
5.3 NoSQL Implementation. Separation of concerns 54
5.3.1 Distributed architecture. 54
5.3.2 Implementation of ReceiptDB 55
5.3.3 Implementation of MachineNetworkDB 57

6 Evaluation and experiments **61**
6.1 Evaluation 61
6.1.1 Scalability and Performance 62
6.1.2 Usability and maintainability 63
6.1.3 Cost reduction 65
6.2 Performance experiments 67
6.2.1 Experiment results 67
6.2.2 Indexes 69
6.2.3 Summary 71

7 Summary and Conclusion **75**
7.1 NoSQL databases 75
7.2 Research questions discussion 76
7.3 SQL or NoSQL? 77

A Listings of queries for populating databases with data **79**

List of Figures

1.1 NoSQL momentum, Evans Data 5

2.1 Customer data example of Key-value store 15
2.2 16
2.3 Core graph entities 18
2.4 Data availability mechanism 21
2.5 Large collection with data distributed across 4 shards. 22
2.6 Cypher declaring a relationship between nodes "a" and "b" . 23

4.1 T9, one of the flagships in the reverse vending machines market. 39
4.2 Three step process. 40

5.1 Sample barcode 46
5.2 Monolithic database architecture 48
5.3 MSSQL structure 53
5.4 Distributed project architecture 55
5.5 Two areas of responsibility 56
5.6 Graph Structure 58

6.1 Web interface for Neo4j console 64
6.2 Performance of Neo4J against SQL 68
6.3 Performance of Neo4J using indexes 71
6.4 Performance of SQL using indexes 72
6.5 Performance of Neo4J with indexes against SQL with indexes 72
6.6 Response times on verify request 73

List of Tables

2.1 Table with movie data 17
2.2 Row-oriented Database Layout 17
2.3 Column-oriented Database Layout 17
2.4 Summary information of popular NoSQL databases 19

6.1 Pricing for standard licensing per core 66
6.2 Time in ms, required to perform search query in respective db at a given data size 68
6.3 Time in ms, required to perform search query in Neo4j at a given data size 70
6.4 Time in ms, required to perform search query in SQL at a given data size 70

Listings

5.1 SQL query style search 53
5.2 Normalized structure of database 56
5.3 Embedded document 57
5.4 Structure of the database depicted in Cypher 58
5.5 Cypher Query that returns all nodes and relationships 58
5.6 Cypher query style search 58
6.1 Cypher query style search 64
6.2 Enable indexes in Neo4j 69
6.3 Create indexes in Neo4j 69
6.4 Create indexes in MSSQL 69
A.1 SQL. Creation and populating ′Countries′ table 79
A.2 SQL. Creation and populating ′Chains′ table 79
A.3 SQL. Creation and populating ′Stores′ table 80
A.4 SQL. Creation and populating ′Machines′ table 81
A.5 Cypher. Creation of ′Country′ nodes 82
A.6 Cypher. Creation of ′Chain′ nodes 82
A.7 Cypher. Creation of ′Store′ nodes and relationships 82
A.8 Cypher. Creation of ′Machine′ nodes and relationships . . . 83

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Time of explosive growth of non-relational data

In 1970, Edgar F. Codd published a paper where he revealed his thoughts on how information stored in the large databases could be accessed without knowing the details of how or where mentioned information is stored[6]. His revolutionary ideas spawned a new family of products, known today as relational databases, which began with IBM DB2 in mid-1980s and then continued into the 1990s with Oracle, Sybase, Microsoft SQL Server and MySQL. Relational databases have since become the predominant choice for keeping of financial records, personnel data, manufacturing, logistical and other information.

Today's businesses rely on the collection and storage of increasing amounts of data. Their information processing demands have already outgrew the relational database capabilities. The Web's explosive growth contributed to the need for businesses to manage not only increasingly large volumes of data, but also data that must be made available across distributed (geographically or otherwise) systems and which does not follow a common relational data model.

While Internet giants such as Amazon, Facebook and Google may have been the first to truly struggle with the big data problem, enterprises across industries - and not just Web-based organizations - are now struggling to manage massive quantities of data, data entering systems at a high velocity or more commonly, both. For example, according to a recent report from consulting giant McKinsey & Company, the average investment firm with fewer than 1,000 employees has 3.8 petabytes of data stored, experiences a data growth rate of forty percent per year and stores structured, semi-structured and unstructured data[8].

As a result of the aforementioned demands of large-scale data storage, it was not long before a movement began with an aim to address the main problems data scientists and engineers were facing. With the original intention of building modern, scalable databases, a new and advanced set of software has emerged to meet today’s data handling demands and the term "NoSQL" was introduced to describe these progressive data management engines. Being non-relational, distributed, open-source and horizontally scalable, they contained some RDBMS-like qualities, but went beyond the limits that typically constrained traditional relational databases. There has been a rapid shift to a new method for storing data since then and almost all software development related conferences have NoSQL topics in their agendas. Technology leaders are no longer asked *if* they will have a NoSQL strategy, but rather *when* their NoSQL strategy will roll out - and more importantly, what it will be comprised of.

1.2 Not only SQL data storage

What exactly is NoSQL? Some may think that the "No" part of the NoSQL name is intended to distinguish it as apart from and having nothing in common with SQL solutions. The "No" part of the NoSQL label, however, should rather be thought of as "not only SQL"[1], which communicates the fact that a NoSQL database does not completely discard all features/functions that define a relational database. In fact, a few NoSQL databases provide a SQL-like query language that helps to ease the transition from the world of RDBMS. NoSQL rather completes SQL and provides an alternative when conventional solutions are found not to be as efficient. Today’s NoSQL databases can:

- Serve as an online processing database, so that it becomes the primary datasource/operational datastore for online applications[8].
- Use data stored in primary source systems for real-time, batch analytics and enterprise search operations.
- Handle "big data" use cases that involve data velocity, variety, volume and complexity.
- Excel at distributed database and multi-data center operations.
- Offer a flexible schema design that can be changed without downtime or service disruption.
- Accommodate structured, semi-structured and non-structured data.
- Easily operate in the cloud and exploit the benefits of cloud computing.

Structure-wise, NoSQL databases are organized in a very different way with data in NoSQL databases being greatly denormalized, residing in structures organized in a variety of formats such as columnar, document, key-value store or graph. Most NoSQL databases do not conform to the standard Codd relational model where data is normalized to a third form.

[1] http://nosql-database.org/, Retrieved October, 2014

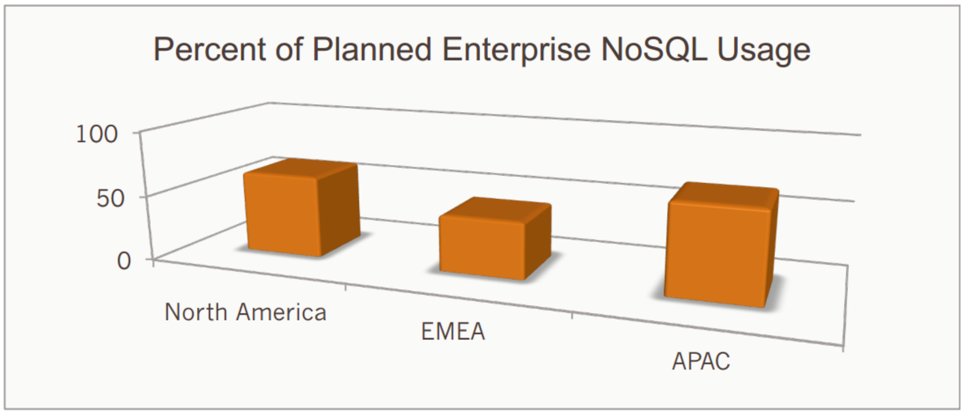


Figure 1.1: NoSQL momentum, Evans Data

That means we can often avoid resource-intensive "join" operations to satisfy end user requests.

NoSQL databases perform best when dealing with data that is either impossible to store properly in an RDBMS or data that performs very poorly when accessed in a relational manner. Let us examine such a problem as traversal in a social network. This problem, like almost any, can be solved in a relational way, yet it becomes unwieldy after a certain point of time as the data continues to increase. The graph database is a convenient way to tackle this kind of problem in a scalable way and Google's Knowledge Graph, Twitter's Interest Graph and Facebook's Social Graph are good examples of this use case.

1.3 NoSQL trends

The capabilities of NoSQL databases are fast becoming well known to IT leaders. For example, an Evans Data survey revealed that corporate enterprise developers in North America are rapidly accepting NoSQL. The study also showed that NoSQL databases are already being used in fifty six percent of organizations surveyed and sixty three percent of respondents said they plan to use NoSQL within the next two years[8] (Figure 1.1).

We can observe that NoSQL adoption is significantly stronger in the Asia-Pacific region as nearly seventy percent of respondents from the region have plans to introduce NoSQL databases in their projects. Yet even in the EMEA (Europe, Middle East and Africa) region, we can expect a growth of forty percent of enterprises which are planning to utilize NoSQL within the next two years.

These numbers imply that NoSQL databases may indeed replace most of the traditional relational databases in the enterprise sector. Netflix, the world's leading Internet subscription service for movies and TV shows, has replaced a number of its existing Oracle systems with Cassandra NoSQL database, which runs in the cloud. One of the main reasons for this shift was due to the centralized nature of the SQL-based database system that they were running before. With one single point of failure, it was impossible to guarantee high levels of up-time world-wide. Another problem was that schema changes required system downtime. "Every two weeks, we'd have at least ten minutes of downtime to put in the new schema", Cockcoft explains. The limitations of a SQL database impacted their availability and scalability, not to mention the reliability and flexibility they needed to create and manage data clusters quickly as the company expanded internationally[9].

While the shift from SQL to NoSQL databases that Netflix and other big companies have made is common, many companies are choosing to leave their existing legacy RDBMS systems in place. However, they are actively introducing NoSQL databases with their new projects, especially when new systems require flexible or non-standard schema designs expected to deal with large-scale data or should be executed in a distributed environment. NoSQL databases allow them to easily integrate new systems with existing ones benefiting from new technology without disrupting their main business.

Technology aside, another reason many new development and/or migration efforts are being directed towards NoSQL databases is the high cost of legacy RDBMS vendors versus NoSQL software. In general, NoSQL software costs are a fraction of what vendors such as IBM and Oracle charge for their databases. This fact is especially important in the age of startups, such as we are experiencing today when small companies who are yet to earn their first revenues need stable databases that are able to hold large volumes of data from very early on in their business endeavours.

1.4 Motivation

Currently there are large-scale changes taking place within many enterprises and many of these said changes are paving the way for great opportunities when it comes to NoSQL databases. Enterprises are internally developing more and more web-based or service-oriented applications with the intention for them to interact with their customers, suppliers, or other stakeholders. They are complex applications that store session information, user generated data, sensor data, telemetric data, etc. and increasingly require the agility and capabilities of the NoSQL.

NoSQL databases have been chosen as the topic of this thesis due to the increasing impact of data-usage across the IT industry and major problems with relational databases that do not fulfil all the needs of the enterprise for scalable, distributable and efficient data-storage. One of the main challenges with NoSQL, however, is the fact that there are relatively few engineers and organizations who are familiar with NoSQL databases, thus making it difficult for them to make a choice in terms of data storage solutions and methods in their favour. The intention of this thesis is to serve as both research insight on NoSQL databases as well as an assessment of them in the situation of a real industrial case which also happens to be a first-time-encounter environment.

1.5 Problem statement

We have already argued that NoSQL data storages may be the optimum choice for solving big data problems or for data distributed across multiple servers in a web environment. However, not all businesses operate with big data, and not all of them require web scalability. Indeed, there are relatively few works highlighting the usage of NoSQL in a medium or small-sized enterprise environment where traditional usage of SQL databases more frequent than in any other environment[31]. This thesis investigates how a medium-sized enterprise company may successfully utilize NoSQL data storage solutions and benefit from them. Thus, the Main Research Question is:

- *MRQ: What are the advantages of using NoSQL data storage in the enterprise environment?*

From this initial question it is possible to derive a number of Secondary Research Questions that aid in expanding the topic.

- *SRQ1: How does the choice of data storage influence the flow of the application development and architecture of the system?*

- *SRQ2: How easy is it to integrate a new solution with the legacy one when using NoSQL databases?*

In order to investigate the MRQ, a real industrial case concerning of a large company will be analysed. This company currently utilizes multiple legacy systems, while their crucial business data is rapidly increasing. It is necessary for them to implement new services and systems using a non-conventional approach in order to meet future business requirements in addition to the requirements of their partners.

In the course of this investigation, thorough analysis of data-behaviour and data usage of the system the aforementioned company intends to build will be carried out. Consequently, it should be possible to identify the most common use cases as well as typical scenarios of data writing and data reading. The acquisition and analysis of such information in conjunction with further research should help conclude whether or not the use of NoSQL data storage in this particular case is in face necessary. In due course a selection of available solutions on the market will be described and discussed in a bid to defend the reasoning behind the author's final determination of the most suitable solution.

In order to investigate SRQ1 and SRQ2, it will be necessary to analyse the process of decision making when modelling data and designing the architecture of the system under the research. Evaluation of the extent to which NoSQL influenced the application development process will likewise be undertaken. After the system has been built, the performance of the solution will then be tested and evaluated based upon specific requirements from the industrial partners with the results being compared with a possible RDBMS solution. Assessment of the executed solution will be based upon multiple criteria, including the ease of building, availability of the common language API and costs related to the development.

1.6 Industrial case with Tomra AS

An industrial case study was used to further investigate the subject of using NoSQL data storage solutions in a business enterprise. The case study explores the question of how, by using different database solutions, we may change the way our data is modelled as well as the way we develop an application. Tomra AS is a Norwegian company based in Asker and is the leader in the Reverse Vending market. Tomra were in need of a new solution that would allow them to validate the receipts coming from their reverse vending machines. As part of this project we, the team of external consultants in collaboration with Tomra specialists and management, have implemented a new solution using NoSQL databases as the method of data storage for the system. During the implementation two popular NoSQL solutions came to be chosen - MongoDB document database and Neo4j graph database. This is an unprecedented case in the Tomra corporation and made for an interesting task to develop a new system using NoSQL databases and then to be able to integrate it with the existing systems the company has. The case study will be discussed in more detail in Chapter 4.

1.7 Structure of the thesis

The structure of this thesis will coincide with the following order to systematically investigate the problem area of developing a NoSQL solution in an enterprise environment. Part 1 contains the chapters Introduction, Background and Research methods. In the course of the introduction, the topic of the topic of the thesis is presented, alongside the defence of and explanation as to why the subject of this thesis is relevant. In addition, the research questions and a hypothesis are also presented. The Background chapter gives a broader explanation of characteristics of NoSQL databases, their appliances and features. Different studies related to this thesis are discussed in the last part of the chapter. The methods used for gathering data and implementing the applications are described in the chapter entitled Research methods.

Part 2, Industrial Case, Implementation and Evaluation, contains the description of the Industrial case used in the course of this research project.

It likewise elaborates upon the process described in the Implementation chapter. Experiments and evaluations have been conducted in order to investigate the problem area in more depth. The final chapter summarizes the thesis and discusses whether the hypothesis has been confirmed or not.

Chapter 2

Background

In this chapter we will look closer at the different types of NoSQL databases, their features, advantages and things they are criticized for. Our aim is to acquire an understanding of the subject to make well informed decisions about incorporating NoSQL in the enterprise. We will describe MongoDB and Neo4j database systems in more detail, since they are featured in the industrial case that we will present in the next Chapters. In the following section we present related articles and studies investigated in the NoSQL field with focus on the cloud, web and distributed environment. In the last part of the chapter we will look through several case studies highlighting experience of NoSQL being used in a real business environment to leverage existing systems.

2.1 Introducing NoSQL

First mention of NoSQL was made in 1998 by Carlo Strozzi. He used it as a name for his open source relational database that did not offer an SQL interface. The term was reintroduced in 2009 by Eric Evans in conjunction with an event discussing open source distributed databases[18]. At that time it was not used to describe a particular system, but rather a whole new mindset of non-relational distributed data stores that emerged in early 2000's. Hundreds of NoSQL databases appeared in the market since then.

NoSQL properties

Today we have to deal with the broad range of NoSQL implementations and term variations associated with them. However, most NoSQL databases share some common traits that can be used to distinguish them, such as:

- They lack fixed schemas
- They avoid joins (the operation of combining relations)
- They scale horizontally

Another common characteristic of NoSQL databases for which they are sometimes criticized, is that they lack ACID transactions. Those set of properties (Atomicity, Consistency, Isolation, Durability) guarantee that all transactions transform a database from one valid state to another. Once a transaction updates a database item, all database clients (e.g. users and applications) will see the same value for the updated item.

CAP theorem

In order to understand why ACID properties often sacrificed in NoSQL databases, let us examine findings in the area of distributed computing that were presented by Eric Brewer back in 2000 year. He presented a CAP theorem[4][12], the postulate that describes three essential system requirements necessary for the successful design, implementation and deployment of applications in distributed computing systems. They are Consistency, Availability and Partition Tolerance – or CAP. The theorem also states that a system can guarantee only two of the three mentioned properties. Due to the fact that NoSQL databases are occupied with availability and partition tolerance of the large-scale distributed systems, the consistency has suffered, therefore ACID properties could not be maintained.

BASE

Consistency and reliability, however, are still attained in NoSQL databases, by embracing the notion of *BASE*[29]. BASE states for the Basic Availability, Soft state and Eventual consistency.

- Basic availability implies disconnected client operation and delayed synchronization meaning all data is distributed and holds tolerance to temporary inconsistency and its implications.
- Soft state means the state of the system may change over time, even if there is no any input (leading us to the eventual consistency).
- Eventual consistency property guarantees that even when data is not consistent, eventually it will be. After a database item has been updated for long enough period of time, all clients will see the same value for the updated item.

In most of the cases mentioned properties mean that applications should be aware of non-repeatable read results due to the latency in consistency[13]. An item in the internet shop might be shown as available for some time after it has been sold out. The flight ticket price might change during the check out process. In return, however, we get extremely fast insert and read operations.

Unprecedented data volumes, connected data, performance and scalability requirements of modern data-driven applications changed the way we approach data management. And it is not clear yet, if available RDBMS solutions can be flexible enough to be able to satisfy needs of the modern business. One thing that is certain that NoSQL databases can cater to very different needs. Thus, arguably providing better-suited solutions for many today's data storage problems. Consistency, availability and partition tolerance are three primary concerns that determine which data management system is suitable for a given application.

2.1.1 Criticism of NoSQL

Even though NoSQL growth is impressive, not everyone is ready to jump into the wagon just because of the hype. There are structural, idealogical and other constraints in NoSQL that people have to deal with. Only knowing these disadvantages will let enterprise to make well-weighted decisions and build reliable solutions for their business.

Redundancy

A lot of performance gains achieved in NoSQL databases by denormalization, the process of optimizing the read performance of a database by adding redundant data[30]. In practice it means that related data, such as for example customer information and street address are stored in the same place. That way we can extract all data together when we pull the information about customer, thus avoiding performance-expensive "joins" on several tables by customer id. As the result of having arbitrary records stored in any place, our data can be duplicated throughout the database. In our example that makes it a challenge to update street addresses and make sure it is consistent in every record.

Lack of true ACID transactions

We have already mentioned this problem and while there are thousands of businesses that are content with BASE properties of NoSQL databases, there are still certain industries where ACID is a must have property. Any financial or sensitive data requires the database to be guaranteed in a valid state at any given time. We do not want our financial data "disappear" until the state of the database will become consistent again.

Zoo of implementations

After many years of use, SQL have become standardized and well-spread query language. Database engineers are capable of writing SQL queries for virtually any SQL-based database system with rather moderate changes in syntax. Structure and logic are very similar whether one use Oracle DB, MSSQL, MySQL or PostgreSQL. It does not hold true in case of NoSQL, where dozen of paradigms with dozen of popular implementations exist and provide their custom made APIs. While giving us a freedom of choice, variety of different database systems might become a challenge when learning new paradigm and integrating it with existing systems.

Immaturity

Most of the other shortcomings NoSQL database systems have regarding bad support, lack of documentation and questionable reliability can be well explained by the fact that most of the NoSQL systems are much younger than existing SQL implementations. For many years database developers were working on query optimizations, tools, documentation, thus providing very reliable, mature and well-predicted experience of using SQL database systems. NoSQL databases are still on their way to integration with more tools. They get more functionality and number of NoSQL adepts is growing. We are seeing that NoSQl world started with denying everything SQL made so far, but it is slowly adding back things that look like transactions, schemas and standards[1]. This will be the next step to the maturity and wide-adoption of the NoSQL databases.

2.1.2 Types of the NoSQL databases

There have been various attempts to classify NoSQL databases. However, because of variety of implementations and approaches it is difficult to come up with all-suited overview and classification. In addition, we should mention that a lot of databases do not belong to one of the types completely, incorporating hybrid features of several classes. Although classifications that describe seven and more types exist, in this paper we will divide NoSQL databases into four main types:

- Key-value stores
- Column-oriented stores
- Document Databases
- Graph Databases

Key-value (KV) stores

Key-value stores use the associative array (also known as a map or dictionary) as their fundamental data model. In this model, data is represented as a collection of key-value pairs, such that each possible key appears at most once in the collection. This data structure is available across many programming languages and is very well-known to software developers. Query, delete, insert and modify operations for data are executed through the primary key. Being simple structure, still it provides query speed higher than in relational database, supports mass storage and high concurrency[14].

[1] http://www.infoworld.com/article/2617405/nosql/7-hard-truths-about-the-nosql-revolution.html, Retrieved October, 2014

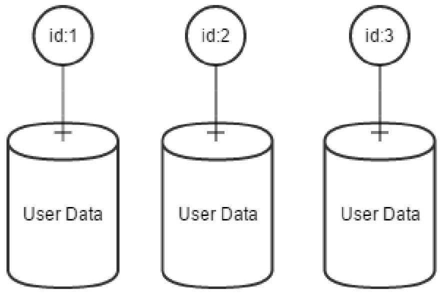


Figure 2.1: Customer data example of Key-value store

Key-values represent buckets of data. For example, in case of a customer database illustrated in Figure 2.1, each user data information is represented in individual bucket and represented using a key-value which in this case is user id. The key-values can be serialized using either Java serialization or XML. This way it is very fast to store as it just writes bits to the discs. Some popular key-value store implementations in the market are Berkeley DB, Tokyo Tyrant, Voldemort and Redis. Voldermort describe their vision of key-value store database as "basically just a big, distributed, persistent, fault-tolerant hash table."[2] It should be noted that such simplicity is often achieved through hiding the details of implementation from an application developer. Even though key-value store may look and act like an associative array, it can rely on tables, indexes and other properties of relational systems to be efficient in practice.

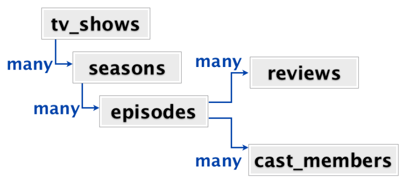
Document databases

Document-oriented databases are semi-structured data storages usually designed around abstract notion of a "Document". Although the implementation of the database might differ, the idea behind is to allow the client applications to address documents and their content in the most convenient way for them. A document may represent collection of tags, meta-data or collection of collections. In some way it can be analogous to a tabular structure with records in it, except we do not have to follow the same structure for all the records. It is possible to add and remove attributes to any single tuple or collection without wasting space, by creating empty fields for all other tuples or collections. That resolves in all tuples can contain any number of fields of any length. As the result, the application programmer gains ease of use and the possibility to create very dynamic data.

Consider an example with set of TV shows. Each show consists of many seasons, each season has several episodes and each episode has many reviews and many cast members[3]. This structure is depicted in Figure 2.2a.

[2] http://www.project-voldemort.com/voldemort/, Retrieved September, 2014

[3]http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/, Retrieved October, 2014



(a) TV show structure

```
{title: 'Babylon 5',
 seasons: [
  {season_number: '1',
    episodes: [
     {ordinal_within_season: '1',
       title: 'Midnight on the Firing Line'
       reviews: [{...}],
       cast_members: [{...}]
      }
    ]
  }
 ]
}
```

(b) Babylon 5 TV season encoded in the document

Figure 2.2

Typically, we want to retrieve the information connected to that TV show all at once, such as user is able to see how many seasons does the TV show has, how many episodes and who are the cast members. If the data we need for a TV show is under one document (Figure 2.2b), it will be very fast to retrieve all this information at once, even if the document is very large.

The drawback of such structure is that we lack consistency and is some cases write performance. Consider an example if one of the cast members have changed their name. In the document database the records of this cast member could be in a very large number of the documents. We will have to find all the documents with this cast member and update the same information in all records. Traditional concepts like indexes and keys are often employed in the same sense as in relational databases. By using these, one is supposed to achieve almost the same performance as it would be possible in a system implemented with tables[18]. Most often documents implemented in a semi-structured file format, accessible both for humans and computers, such as JSON, XML, YAML or BSON. Typical document database examples are MongoDB and CouchDB.

Column-oriented stores

Column-oriented stores in a nutshell, store each database table column separately with attribute values belonging to the same column stored contiguously, compressed and densely packed as opposed to traditional database systems that store entire records (rows) one after the other[1]. To illustrate the concept consider the Table 2.1 of best movies on IMDB[4]. In a row-oriented relational database, this information will be stored as depicted in a Table 2.2. While in a column-oriented database, data in each column will be recorded contiguously, which is presented in a Table 2.3 Reading a subset of a table's columns becomes faster, at the potential expense of excessive disk-head seeking from column to column for scattered reads or updates.

[4]http://www.imdb.com/chart/top, Retrieved September, 2014

Title	Year	Rating
The Shawshank Redemption	1994	9,3
The Godfather	1972	9,2
The Godfather Part II	1974	9,0
...	...	...

Table 2.1: Table with movie data

Title	Year	Rating
The Shawshank Redemption	1994	9,3
The Godfather	1972	9,2
The Godfather Part II	1974	9,0
...	...	...

Table 2.2: Row-oriented Database Layout

Title	Year	Rating
The Shawshank Redemption	1994	9,3
The Godfather	1972	9,2
The Godfather Part II	1974	9,0
...	...	...

Table 2.3: Column-oriented Database Layout

Key characteristics of column database applications are: tolerance to temporary inconsistency, need for versioning, flexible database schema, sparse data, partial record access and high speed of insert and read operations[13]. When a value changes it is stored as a different version of the same value using a timestamp. In other words, the notion of update is effectively nonexistent. Partial record access contributes to dramatic performance improvements for certain applications. Columnar databases perform aggregate operations such as computing maxima, minima, average and sum on large datasets with extreme efficiency.

Column family is a set of related columns. Column databases require predefining column families and not columns. A column family may contain any number of columns of any type of data, as long as the latter can be persisted as byte arrays. Columns in a family are logically related to each other and are physically stored together. Performance gain is achieved by grouping columns with similar access characteristics into the same family. Database schema evolution is achieved by adding columns to column families. A column family is similar to the column concept in RDBMS. Systems in this category include Google BigTable, Apache Cassandra, Apache HBase, Hypertable and Cloudata.

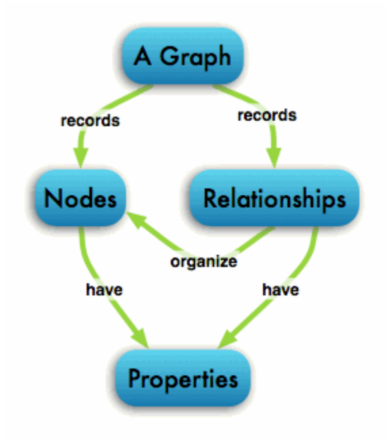


Figure 2.3: Core graph entities

Graph database

Graph database models defined as those in which data structures for the schema and instances are modelled as graphs or generalizations of them and data manipulation is expressed by graph-oriented operations and type constructors[3]. Graph data models work with 3 core abstractions: Nodes, relationships between nodes and key value pairs which can be attached to nodes and relationships (Figure 2.3). Graph databases are optimized for use cases where you have connected data. Today connected data is prevalent in social networking, logistics networks (for package routing), financial transaction graphs (for detecting fraud), telecommunications networks, ad optimization, recommendation engines, bioinformatics (protein calculations) and in many other places. In all these cases graph databases outperform traditional database systems. Today the paradigm has shifted from caring about static records of data to being occupied with the connections between different data nodes and ways we can leverage useful information from their relationship.

Most of the large companies have their own implementations of graph database that is tailor made for their core business. Google uses its Knowledge Graph to enhance the quality of their search. Twitter's Interest Graph and Facebook's Social Graph are used to map users, their relationships and interests. Microsoft's Office Graph is bringing new ways to people that can be productive at their work, using relationships between colleagues and their documents. Graph databases are also popular for implementing access control and authorization subsystems for applications that serve millions of end users. Graph databases include FlockDB, InfiniteGraph, Titan, HyperGraphDB, AllegroGraph, Affinity, OrientDB and Neo4J.

System	Type	API	Language	Storage	License
MongoDB	Document	BSON	C++	Disk	AGPL v3.0.
CouchDB	Document	JSON/REST	Erlang	Disk	Apache
Elasticsearch	Document	REST	Java	Disk	Apache
RavenDB	Document	HTTP/JSON	C#.NET	Disk	AGPL v3.0
Riak	Key-value	JSON/REST	Erlang	Plug-in	Apache
Redis	Key-value	Multiple	C++	RAM	BSD
Voldemort	Key-value	Multiple	Java	RAM	Apache
Berkeley DB	Key-value	Multiple	C	RAM	AGPL v3.0
Cassandra	Column	CQL/Thrift	Java	Disk	Apache
HBase	Column	Java/mult	Java	HDFS	Apache
Hypertable	Column	Thrift	C++	Files	AGPL v3.0
Neo4J	Graph	REST/mult	Java	Disk	Personal
AllegroGraph	Graph	REST	C#.NET	Disk	Commercial
Infinite Graph	Graph	JAVA/DLB	Java	Disk	EULA

Table 2.4: Summary information of popular NoSQL databases

2.2 MongoDB: A Document-oriented database

MongoDB (from "humongous") is an open-source, cross-platform, document-oriented database developed in C++. It was developed in October 2007 by 10gen(now MongoDB, Inc.) with first open-source public release in 2009. It is currently in version 2.6.5 and available to download for Windows, Linux, Mac OS X or Solaris operation systems.[5] Being a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favour of JSON-like documents with dynamic schemas. The format to store documents in MongoDB is BSON – Binary JSON with the maximum 16MB size for each. The maximum document size helps ensure that a single document cannot use excessive amount of RAM or excessive amount of bandwidth during transmission. Like JSON, BSON supports the embedding of documents and arrays within other documents and arrays.

Concepts and structure

In MongoDB there are no database schemas or tables. Instead, documents which are similar to rows, are grouped into collections which are similar to tables. Document is a data structure composed of field and value pairs. The values of fields may include other documents, arrays and arrays of documents. MongoDB automatically generates a primary key (id) to uniquely identify each document. The id and document are conceptually similar to a key-value pair. MongoDB attempts to hold most of the data in memory so simple queries take less time by avoiding expensive hard disk retrieval operations. One problem to this is once the data set becomes larger than the available memory, then MongoDB will have to start querying the hard disk

[5]http://www.mongodb.org/downloads, Retrieved October, 2014

for results. For this reason, it is advised to use 64-bit version of MongoDB, since MongoDB is limited to a total data size of about 2GB for all databases in 32-bit mode.

Indexing

In order to increase performance while working with documents, MongoDB uses indexing similar to relational databases. Each document is identified by _id field and over that field is created unique index. Although indexing is important to execute efficiently read operations, it may have negative impact on inserts. Apart from automatic index created on _id field, additional indexes can be created by database administrator. For example, can be defined index over several fields within specific collection. That feature of MongoDB is called "compound index". However, all indexes use the same B-tree structure. Each query use only one index chosen by query optimizer mechanism, giving preference to more efficient index. Eventually query optimizer re-evaluates used indexing by executing alternative plans and comparing execution cost.

License and adoption

Released under a combination of the GNU Affero General Public License and the Apache License, MongoDB is free and open-source software, with MongoDB, Inc. offering commercial support and other services. MongoDB has been adopted as backend software by a number of major websites and services, including Craigslist, eBay, Foursquare, SourceForge, Viacom and the New York Times, among others. MongoDB is the most popular NoSQL database system up to date[6].

2.2.1 MongoDB competitive features

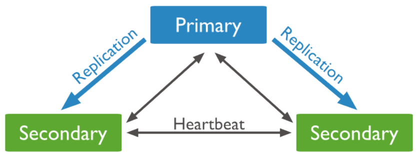
Replication

Replication is the mechanism in MongoDB that gives the database durability and concurrency. It is the process of synchronizing data across multiple servers. This way replication provides redundancy and increases data availability. MongoDB uses Master-Slave replication mechanism. It allows defining a Master and one or more Slaves. Master can write or read files while Slave serves as backup, so only reading operations are allowed. When Master goes down, Slave with more recent data is promoted to Master.

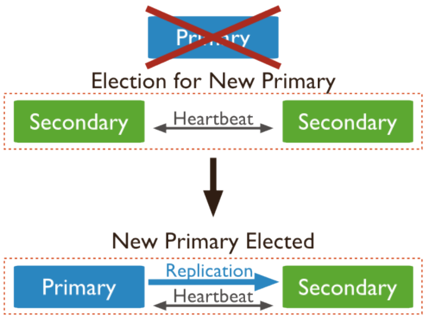
Automatic Failover

Automatic Failover is the name of this mechanism which ensures availability of the service. Figure 2.4 illustrates this process. With multiple

[6]http://www.mongodb.com/leading-nosql-database, Retrieved October 26, 2014



(a) Replication



(b) Automatic Failover

Figure 2.4: Data availability mechanism

copies of data on different database servers, replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup. You can also use replication to increase read capacity and achieve concurrency. Clients have the ability to send read and write operations to different servers. You can also maintain copies in different data centres to increase the locality and availability of data for distributed applications.

Sharding

Sharding is a method of data partitioning across multiple databases. Sharding is one of the MongoDB core features, which allows it to support deployments with very large data sets and heavy throughput operations. With increasingly growing size of a database, number of transactions and application throughput, the response time for querying single database increases exponentially. It is a great challenge for single server to provide necessary CPU, storage, memory and I/O capacity for the data operations.

Scaling by adding CPU and storage resources to increase capacity is called *vertical scaling* and it has limitations. Costs of creating and maintaining high-end servers with large number of CPUs and large amount of RAM are disproportionately high compared to smaller systems. In addition to that, cloud-based solutions does not provide server instances that are big enough to satisfy big data needs. There is a clear practical limitation for vertical scaling.

By contrast, sharding or *horizontal scaling*, distributes the data sets or data shards across a number of much less expensive commodity servers. Data shards have comparatively little restriction as far as hardware and software requirements are concerned. Each shard operates as an independent database, but collectively, the shards make up a single logical database (Figure 2.5). In a lot of cases, database sharding can be achieved fairly simply. One common example is splitting a customer database

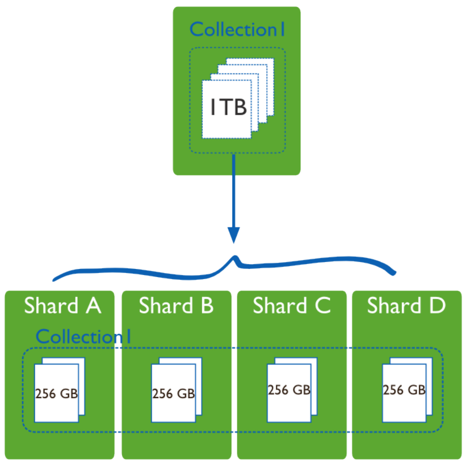


Figure 2.5: Large collection with data distributed across 4 shards.

geographically. Customers located in Europe can be placed on one server, while customers in the US on another. Given there are no customers with multiple locations, the split is easy to maintain.

2.3 Neo4j: A graph database

Neo4j is an open-source NoSQL graph database implemented in Java and Scala. Started by Neo Technology in 2003, it has been publicly available since 2007. Neo4j is used today by hundreds of thousands of users in almost all industries. Use cases include match making, network management, software analytics, scientific research, routing, organizational and project management, recommendations, social networks and more. Neo4j is one of the few general-purpose graph database engines that are mature, robust and have a great supporting community.

Neo4j implements the Property Graph Model down to the storage level. As opposed to graph processing or in-memory libraries, Neo4j provides full database characteristics including ACID compliance, cluster support and runtime failover, making it suitable to use graph data in production scenarios. Neo4j is a high-performance database, especially for highly connected data, where Neo4j can be thousands of times faster than relational databases[26], making it a ideal for managing complex connected data. Its model is intuitive and expressive, mapping closely to the whiteboard domain model. High performance, maturity and ease of use make Neo4j world's leading graph database[7].

[7]According to http://neo4j.com/ and general consensus on that matter

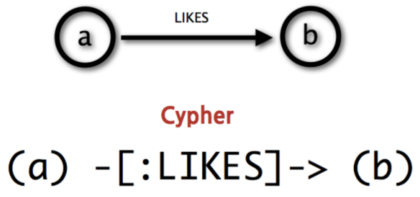


Figure 2.6: Cypher declaring a relationship between nodes "a" and "b"

2.3.1 Neo4j competitive features

ACID compliance

Being a NoSQL database, Neo4j possesses all the advantages of its kind, such as flexible schema, horizontal scalability, high-performance and high availability. However, some particular features make Neo4j very popular among users, developers and DBAs. Among them is proper ACID behaviour. It is the foundation of data reliability in Neo4j. It enforces all operations that modify data to occur within a transaction, guaranteeing consistent data. This robustness does not only applicable for single embedded graphs instance, but it extends to multi-sharding high availability installations. Among other important features in Neo4j is its simple, yet powerful query language Cypher and highly efficient traversal mechanisms.

Powerful traversal

Traversing a graph means visiting its nodes, following relationships according to some rules. Graph databases are all about connected graph data. One of the key features in Neo4j is constant time traversals for relationships in the graph both in depth and in breadth due to double-linking on storage level between nodes and relationships. In most cases only a subgraph is visited, as you already know where in the graph the interesting nodes and relationships are found. Neo4j comes with a callback based traversal API which lets you specify the traversal rules. In combination with compact storage and memory caching for graphs it results in efficient scaling up to a billions of nodes in one database on moderate hardware.

Rich query language

Cypher is Neo4j's Graph Query Language. Cypher is a declarative, SQL inspired language for describing patterns in graphs. Like SQL, Cypher is not only a query language but does also allow data manipulation like updates and deletes from a graph database. Unlike SQL, however, it does not require us to describe exactly how to do it. Cypher is a relatively simple but still very powerful language. We can express very complex database queries with simplicity and elegance Cypher provides. From developer point of view, it allows us to focus on our domain instead of worrying about database structure. The constructs are designed to be human-readable, based on English language and iconography which helps to make queries more self-explanatory as we can see in the Figure 2.6. Cypher is inspired by a number of different approaches and builds upon established practices for expressive querying. Most of the keywords like WHERE and ORDER BY are inspired by SQL. Pattern matching borrows expression approaches from SPARQL. Some of the collection semantics have been borrowed from languages such as Haskell and Python. Cypher focuses on the clarity of expressing what to retrieve from a graph, not on how to retrieve it.

2.4 Related work

Generally, works about NoSQL databases fall into two categories. First is fundamental research or general research about NoSQL databases, their features and properties. Another type of research is comparison between different implementations of NoSQL databases, or between NoSQL and SQL databases. The aim of this thesis is to investigate how NoSQL databases will be used in enterprise. That is why the main area of interest is comparison between different database implementations in terms of performance or other features. General research about NoSQL in distributed or cloud environment is also in our focus. We overview most recent studies in respective areas. However, none of the works had similar approach as in this thesis.

2.4.1 Fundamental research of NoSQL

NoSQL assessed on elasticity

Konstantinou I. et. al. in their work "On the Elasticity of NoSQL Databases over Cloud Management Platforms"[17], performed a study of the elasticity feature in the cloud-enabled environment on some of the popular NoSQL databases. In this work they quantified and analysed the costs and gains of various NoSQL cluster resize operations, utilizing three popular NoSQL implementations. HBase is identified as fastest for reads and scales well with node additions. Cassandra performance described as fast writes and good scalability, without any transitional phase during node additions. Riak is found unresponsive in high request rates, an it can scale only at lower rates but rebalanced automatically. All three implementations achieve small gains from a data rebalance in general, provided they were under minimal load.

NoSQL and horizontal scaling

Another work that focuses on database scalability in the web environment - "NoSQL Databases: a step to database scalability in Web environment"[27] by Jaroslav Pokorny. He described the challenges that cloud-computing bring to the databases that support large-scale, data-intensive applications.

He agrees that in order to achieve horizontal scaling, databases have to relax some of their usual characteristics, such as for example transactions or schemas. And NoSQL databases are a next step to tackle this problem. However, he argues that an adoption of NoSQL data stores will hardly compete with relational databases that represent huge investments and mainly reliability and matured technology.

NoSQL for Big Data

Gudivada V. et. al. wrote a report on "NoSQL Systems for Big Data Management"[13]. They provided taxonomy and unified perspective on NoSQL systems with regard to Big Data and the way it created a need for out-of-the-box horizontal scalability for data management systems. In this work they compared various NoSQL systems using multiple facets including system architecture, data model, query language, client API, scalability, and availability. They concluded that NoSQL systems are predominantly used for new applications which are characterized by horizontal scalability, high performance, relaxed and eventual consistency. However, it is also likely that existing applications will begin to use NoSQL through re-engineering process. The current upheaval in the data management systems will help promote using the system that closely matches the application needs. New services such as Amazon EC2 will make NoSQL systems even more economical and within reach for all organizations, both small and big.

2.4.2 Comparative studies

Because of the variety of solutions and implementations available in the NoSQL market, it is very hard to choose between different database systems, especially when they belong to the same type and provide similar functionality. That is why, hundreds of articles, blog posts and books written about comparison and evaluation between different NoSQL implementations.

MongoDB versus Cassandra

Abramova V. and Bernardino J. attempted to compare two popular open-source databases: MongoDB and Cassandra. In their report "NoSQL Databases: MongoDB vs Cassandra"[2] they performed experiments on the execution time according to database size and the type of workload. They tested six different types of workloads: mix of 50/50 reads and updates; mix of 95/5 reads/updates; read only; read-modify-write cycle; mix of 5/95 reads/updates; and update only. Results showed that with the increase of data size, MongoDB started to reduce performance, sometimes showing poor results. Differently, Cassandra just got faster while working with an increase of data. Also, after running different workloads to analyse read/update performance, they concluded that when it comes to update operations, Cassandra is faster than MongoDB, providing lower execution time independently of database size used in our evaluation. In their experiments Cassandra showed the best results for almost all scenarios.

MongoDB versus SQL DB for moderate data

Most often enterprises have to choose between NoSQL and SQL solution and they do not have problem of big data, such as large corporations have. In order to make calculated decision, it is important to compare performance of processing of modest-sized structured data in an NoSQL database with traditional relational database. Parker Z. et. al. in their work "Comparing NoSQL MongoDB to an SQL DB"[25] highlight this area. Their experiments showed that, MongoDB has better runtime performance for inserts, updates and simple queries. SQL performed better when updating and querying non-key attributes, as well as for aggregate queries. They consider MongoDB a good solution for larger data sets in which the schema is constantly changing or in the case that queries performed will be less complex. Since MongoDB has no true schema defined and SQL requires a rigid schema definition, MongoDB would easily handle a dynamic schema such as a document management system with several dynamic fields and only a few well known searchable fields.

Scalable database solutions

Rick Cattell in his work "Scalable SQL and NoSQL Data Stores"[5] examined a number of SQL and NoSQL data stores designed to scale simple OLTP-style application loads over many servers. He compares the new systems on their data model, consistency mechanisms, storage mechanisms, durability guarantees, availability, query support and other dimensions. After covering over twenty scalable data stores in this paper including Voldemort, Redis, MongoDB, HBase, MySQL cluster, ScaleDB and others, he came to several conclusions. He argues that many developers will be willing to abandon globally-ACID transactions in order to gain scalability, availability and other advantages. NoSQL data stores will not be a “passing fad”. The simplicity, flexibility and scalability of these systems fills a market niche, e.g. for web sites with millions of read/write users and relatively simple data schemas. New relational DBMSs will also take a significant share of the scalable data storage market. Many of the scalable data stores will not prove “enterprise ready” for a while. Even though they fulfil a need, these systems are new and have not yet achieved the robustness, functionality and maturity of database products that have been around for a decade or more.

Neo4j versus MySQL and data provenance

Another comparison between NoSQL and SQL databases have been performed by Vicknair C. et. al. Their paper "A Comparison of a Graph Database and a Relational Database"[33] reports on a comparison of Neo4j with MySQL for use as the underlying technology in the development of a software system to record and query data provenance information. A directed acyclic graph (DAG) is a common data structure to store data provenance information relationships. Both systems performed acceptably on their objective benchmark tests. In experiments Neo4j did generally better at the structural type queries than MySQL. In full-text character searches, the graph database performed significantly better than the relational database. However, they make a conclusion that it is premature to use the graph database for a production environment where many queries will be on parameters stored in a semi-structured way, due to security issues and lack of support, even in the face of Neo4j's much better string searches.

2.5 Case studies

There are plenty of various of use cases from different industries and companies that emphasize customer stories when they used NoSQL and it benefited from them. Here we will take several samples from several different industries, their challenges and how they solved it using NoSQL.

2.5.1 Neo4j case studies

Telenor and resource authorization

In the Telenor case study "Resource Authorization Challenge Solved with Graph Database"[22], we can see the challenge the leading supplier of Norway's telecommunications and data services has encountered. With more than 3 million mobile subscribers, Telenor's online self-service management portal experienced performance issues. The existing solution was backed by Sybase, with queries implemented as stored procedures resulting in resource authorization would take as much as 20 minutes for the largest customers. Neo4j database was chosen to address this problem. Modelling the resource graph in Neo4j was quite natural, since the domain being modelled is inherently a graph. Neo4j provided fast and secure access and answers to important questions like: Which subscriptions can a user access, does the user have access to the given resource and which agreements is a customer party to? Neo4j enabled high performance and reliable execution of authorization rules during all access to protected data. "The transition resulted not just in faster performance, but in more maintainable code, because the access rules could be expressed so much more easily in a graph. Query and response times were reduced to seconds and even milliseconds in many cases, from many minutes", stated Telenor representatives.

Neo4j leveraging dating sites

Manhattan-based SNAP Interactive, Inc. likes to give people the third degree. It asks everyone, via web-and-mobile social dating app, Are You

Interested? The question has made SNAP a leading app developer for social networking websites and mobile platforms. From the case study "SNAP Interactive Conquers Online Dating with Neo4j"[23], we can learn about the challenge, that was to represent friends-of-friends data. With data of over a billion person nodes and over 7 billion relationships, they could not find efficient way to search for 3rd degree connections with their existing Apache Solr built solution. The indexing and joins would take too much time to develop and maintain, and the solution wouldn't handle existing traffic or scale. Fox made the decision to move to a graph database and selected Neo4j for its flexibility, speed and ease of use. As the result, they significantly improved dating recommendations through use of friends-of-friends connections. It is also mentioned how Neo4j's Cypher query language sped up the implementation by greatly simplifying coding. According to their data 98% of queries are running faster than expected. SNAP also gained powerful visual insights through Neo4j, by displaying available data geospatially.

2.5.2 MongoDB case studies

MongoDB for content publishing

"Forbes Overhauls Publishing Platform"[19] is the name of the case study, which describes, how Forbes decided to overhaul their entire platform and rebuild their content management system (CMS) on MongoDB. A leading source of business news since 1917, Forbes has always produced quality content, but they lacked speed and robustness with their old, closed system. Outages were common, changes to the architecture were challenging and costly. Forbes first built a custom CMS on MongoDB in just two months. Then they launched a brand new mobile site in less than one month. MongoDB aggregates real-time data, including over one million articles and hundreds of thousands of comments and delivers immediate insight into how readers are responding to content. MongoDB also simplifies the capture and analysis of Forbes' clickstream data: what people are looking at, what parts of the page they're viewing and most importantly, what they're sharing. Now their publishing platform is incredibly fast, open to contributors globally and easy to change without going offline. All at a fraction of the time and cost of their old approach.

Expedia goes global with MongoDB

In another case study, we learn about Expedia and their new system for performing various activities such as finding flight tickets, hotels and car rentals. In their study "Online Travel Gets Personal"[20], they describe a challenge dealing with data that have so many different attributes. Customers want instantaneous and relevant results, which should be available on any device. MongoDB's flexible document store and simple horizontal scale made it possible for Expedia to create a feature that collects highly-dynamic customer information in real-time. That allows them to present personalized offers on the fly. They were able to scale solution from prototype to production in less than two months. After the release, Expedia got a lot of feedback from customers and made a lot of changes into the solution to satisfy their needs. MongoDB's flexible schema allowed them do do radical changes in the database design, without disrupting the solution or any impact on the customer experience. Expedia expects even more growth of the data in the future and they expect that MongoDB will support that growth, thanks to its native sharding capabilities.

2.5.3 Summary

Looking at all these cases, we can find similar traits in all of them, even though they come from completely different industries. First of all, companies come to the power of NoSQL databases, when they need to store data that does not "bend" naturally into table system provided by relational databases. It makes it harder to architect such database, as well as requires a lot of knowledge and experience to make it right. Cost of maintenance of such database rises as well. Another important trait, is that for all companies it was crucial to retrieve data in efficient and instant way. Whether it was social, analytic or game data, the time constraints were limited, and thus the speed of data response and retrieval was the main factor. Not only that, but also data availability is another major concern of companies that deal with large sets of data. Being able to retrieve data quick helps to avoid data congestion. But ability of database to scale is one of the important features that NoSQL databases should provide. Last but not least important property of common problem is how quickly it was possible to model, deliver and operate such database. In fast pace business environment as today, it is highly important to be able to create new features, deliver new values and develop new systems for customers in quick and efficient manner. Therefore, it was important for all companies that NoSQL provide tools, which are easy to use, yet powerful to create small data models and scale it to enterprise sizes.

NoSQL and maturity

Among the reasons why companies have chosen particular solution, is that they consider it as a mature and reliable system. And that is true, having all the success stories behind, many articles written about them, having a great support community and activity in promoting they solution, it seems that NoSQL is here for long time. And unlike some open source alternatives, we can with high probability say it is quite unlikely that it will suddenly cease to exist, leaving all the companies without support and bug fixes. Which is one of the things long-term enterprise companies should think about. Talking about bug-fixing, we should also mention that open-source community supported databases showed themselves as highly reliable databases, without any notable or major issues. Most of respondents agree that error-freeness of the particular chosen database was among the advantages of the solution. We can conclude that NoSQL databases are best choice in following cases:

- When domain data does not fit tabular relational model.
- When instant, real-time response/retrieval of data is required and central.
- When scalability might be an issue as well as demand for high data availability
- When data modelling should be done in quick and efficient way. With a lot of possible changes in the model

In majority of the cases, companies and individual developers choose particular NoSQL solution over its competitors because of the following qualities:

- Maturity and reliability
- Track of success stories and large-scale companies that are already using this database for their purposes
- Good documentation and large community support
- Simple and error-free solution. Easy to manage, maintain and work with

All that have been taken in consideration, when Neo4j and MongoDB have been chosen as solutions for Tomra AS project. We will evaluate preconditions and current situation to assess necessity of using Neo4j graph database and MongoDB document database, as well as we will compare them with the relational database solutions.

Chapter 3

Research Method

In this section we will describe the research methods used for collecting data and analysing topics of this work. We will talk about different existing qualitative and quantitative research methods that were used for data collection, development and decision making. We will discuss following methods that have been mainly used:

- Comparative research
- Secondary source research
- Data analysis research
- Case study research
- Observation and fieldwork

3.1 Qualitative and quantitative research methods

There are several classifications available on the types of the research methodology. One of the most common ones is distinction between qualitative and quantitative methods. The research topic usually dictates which kind of research methodologies we use to build our work on and which methods we should use to collect relevant data. Methods from both methodologies can be used together in some cases.

Quantitative research in Information Systems is a set of methods that allow us to answer research questions about the interaction of humans and computers[1]. In this case researcher is motivated by the numerical outputs and how to derive meaning from them. Examples of accepted quantitative methods include survey methods, laboratory experiments, formal methods (e.g. econometrics) and numerical methods such as mathematical modeling. If one to collect quantitative data he or she is probably measuring variables and verifying existing theories or hypotheses or questioning them. Data is often used to generate new hypotheses based on the results of data

[1] http://dstraub.cis.gsu.edu:88/quant/, Retrieved October, 2014

collected about different variables.

Qualitative research methods were developed in the social sciences to enable researchers to study social and cultural phenomena[21]. Examples of qualitative methods are action research, case study research and ethnography. Qualitative data sources include observation and fieldwork, interviews and questionnaires, documents and texts and the researcher's impressions and reactions. Qualitative research takes place when statistics and numerical methods are not the answer to understanding meanings, beliefs and experience. Qualitative research methods are designed to help researchers understand people and the social and cultural contexts within which they live.

3.2 Data collection in the thesis

One of important principles of data collection is that everything is potential data. It is a challenge to rigidly restrict the scope of data collection in advance, or use formal rules to decide that some data are inadmissible or irrelevant. In this work we mainly use qualitative research methods to find information and to analyse data. The research methods, how they are used and their advantages and disadvantages are described below.

3.2.1 Comparative research

There are many opinions on what is comparative research and when it is used. In general it is the act of comparing two or more things with a goal to discover something about the things being compared. Comparative methods are used when there is an insufficient data to use the statistical method[2].

In section 2.1.2 we provide comparison of two different classes of data store solutions, such as NoSQL and SQL. While we do not cover in great detail, features that are typical of SQL databases, we emphasized on the features of NoSQL that are distinctive from SQL. We also compared different NoSQL solutions available in the market. We compare them on several levels, beginning with classification comparison, ending with implementation comparison. Research did not aim cover all possible NoSQL variations. Rather, it gives general overview and comparative analysis based on characteristic features. We focused on the most relevant comparisons that will help us to get better understanding of the topic and will allow us to answer the research questions.

Chapter 5 compares two approaches in database modelling and implementation. We emulate the course of decision making and design process that

[2] http://www.academia.edu/3510091/Qualitative_research_and_comparative_methods, Retrieved October, 2014

occurred during the industrial case and implemented the same functionality using different database engines. Both the development process and finished applications have been investigated. The emulations demonstrate the use of databases for data modelling, architecture and basic queries, pointing out the strengths and weaknesses of each approach. The comparison of the process is done with a qualitative approach, because it focuses on understanding the differences between development approaches and investigates details in the database model. Not all the features that are part of respective databases are compared and the comparison does not give a comprehensive understanding of the database solutions.

3.2.2 Secondary source research and data analysis

Secondary source documents, scientific texts, pictures and artifacts also can be valuable sources of qualitative data. We can analyze published texts, case studies and articles written by engineers or other researchers and benefit from knowledge construction in information systems. Dealing with the secondary source we have to generalize, analyse, synthesise, interpret or evaluate the original information. In chapter 2 we gather and discuss a lot of material concerning NoSQL databases and research that have been done on the topic. We take a look at the various works that have different focus on the same topic, providing all-round overview on the given research area. It gives us solid foundation for research and help us to explore research questions.

3.2.3 Case study research

A case study is an empirical inquiry that investigates a phenomenon within a specific natural setting and uses multiple sources of evidence[16]. A qualitative case study is used in this thesis to investigate different approaches to database modelling and application development with data-centric view on design and implementation. It is important to discover how the NoSQL technology is used, and to get an understanding of how it affects development process. For this reason, qualitative case study particularly well-suited in our research since the object of our discipline is the study of database systems in organizations.

In this work we use case study as one of the foundations for our research. Described in chapter 4 the implications and reasons of the study, as well as practical and scientific importance of the results. We investigate NoSQL database usage in an enterprise environment and will have visible evidence in terms of the project results and metrics. We also take use of existing secondary case studies in section 2.5 to get a better overview of the topic and understand the flow of the case study.

It can be a challenge to draw a concrete conclusion from a case study, because case study is very influenced by the performer and the setting of the study. It is hard to do a generalizations, when retrieving a data from a case study, because most often it highlights only small part of the big issue, it is highly contextual and depends on the goals of the study.

3.2.4 Participant observation and fieldwork

Observation in qualitative studies typically involves the observer's active involvement in the setting studied[16]. Participant observation allows the observer to ask questions for clarification of what is taking place and to engage in informal discussion with system users, as well as to note the on-going activities and descriptions of the setting. As the result, we have very detailed description of what is going on. It also provides an opportunity to bring up actors' own explanations, evaluations, and perspectives in the immediate context of use, rather than retrospectively.

Chapters 5 and 6 describe all this experience from being part of the whole project from design to eventually implementation and evaluation part. It brings all the observations that have been made during this work, as well as expert and interview data from getting in the conversations and discussions with the system actors. Being part of the project gives a great insight on the processes going on and internal work that is being held. We can observe implications and reasons behind each decision and solution in their contextual environment. It is crucial to assessment of the system as a whole.

3.3 Scientific method

The scientific method is the logical scheme used by scientists to approach a scientific problem. Scientific method is used to produce various scientific theories and find answers to the questions posed within science[11]. The simple version can consist of the following steps:

1. Pose the question in the context of existing knowledge.
2. Address the problem related to the question.
3. Offer a hypothesis as a tentative answer to the problem.
4. Test the hypothesis in a specific experiment.
5. Obtain consistency in the results and analyse them.
6. Evaluate theory and publish the results.
7. Start the process from the beginning if theory requires improvements or additions.

This structural approach to research provided a framework for this thesis. We started with establishing the research questions and analysed importance and actuality of them. After describing a problem, we have developed a hypothesis that certain problem can be solved in particular way. The solution is examined through implementation and experiments. Evaluation and analyse of the results provided an answer on the validity of the hypothesis. In this particular work, we analysed and tested two NoSQL solutions in the industrial context and were able to draw a conclusion and give answers on the research questions. The results have stated whether our hypothesis is invalid or confirmed.

3.4 Development research

Development research is different from traditional research approaches in its focus. Experiments, surveys, correlational analyses are focusing on descriptive knowledge. They hardly provide useful guideline for a variety of design and development problems in information systems. Development research deals with uncertainties in the complex tasks in very dynamic contexts[32]. It aims to make not only scientific contributions, but also practical ones. Developers do appreciate more adequate information to create a solid ground for their choices and more timely feedback to improve their products. Development research benefits the professional community of developers as a whole by growing knowledge of theoretically underpinned and empirically tested design principles and methods.

In some cases development research can be referred as the action research. Action research aims to contribute both to the practical concerns of people in an immediate problematic situation and to the goals of the social science by joint collaboration within a mutually acceptable ethical framework[21]. However, action research roots much older than development research. We can consider development research as a new term, characterized by a proliferation of terminology and a lack of consensus on definitions.

During this thesis, two applications have been implemented. They are part of one system, which solves the problem dictated by the industrial case. On top of that, we have created four emulations, two for each of the applications. The aim of emulations is to show the development process of the application from decision making and design part to implementation and evaluation part, using two different approaches in the database design and using different tools.

The development process consists of several phases:

1. Gather requirements for the system, understand context and problem it solves.

2. Propose several solutions to the problem and analyse them.

3. Test solution prototypes against the problem and gather related information.

4. Design a process of development.

5. Incrementally implement main features of the application.
6. Evaluate after each step and refactor.
7. Test the application and evaluate the results.
8. Deploy.

These steps are very similar to the scientific approach and the goal is to learn after each step and improve the process. The knowledge is distributed between the actors and we focus on incremental improvements of the system and the process. This process correlates to the Agile Methodologies, which are the set of light-weight practices and guidelines for incremental development. However, we are not constrained by these methodologies and rather focus on important aspects for us during the development phase.

Part II

Industrial Case, Implementation and Evaluation

Chapter 4

Industrial Case

4.1 Tomra Systems ASA Industrial Case

Tomra Systems ASA is a Norwegian multinational corporation which creates sensor-based solutions for optimal resource productivity. The company's products and services fall within two main business areas: Collection Solutions (including the business streams reverse vending, compaction and material recovery) and Sorting Solutions (including the business streams recycling, mining and food) The business stream that is of our interest is reverse vending.

A reverse vending machine or RVM, depicted on Figure 4.1, is a device that accepts used (empty) beverage containers and returns money to the user. Tomra is one of the main world vendors of reverse vending machines. Not only that, it is provides maintaining and monitoring of them for the client. Clients of Tomra are usually large retail (grocery) chains that put RVMs inside their stores, thus maintaining full-cycle process by selling the beverages and providing recycling matters for the emptied containers later.

4.1.1 Problem description

In a lot of places around the world RVMs issue money to the user directly. However, Norway and other Scandinavian countries have adopted another system. After recycling containers from beverages in the vending machine,

Figure 4.1: T9, one of the flagships in the reverse vending machines market.

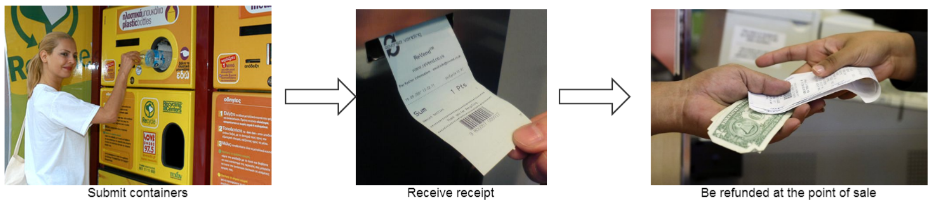


Figure 4.2: Three step process.

the user receives a receipt that is to be taken to a point of sale where it may either be exchanged for cash directly or the amount is discounted from the customers' shopping cart. Therefore we have three step process:

1. Submit containers
2. Receive a receipt
3. Receive refunded money at the point of sale in one or another way

This additional step, as illustrated in the Figure 4.2, introduces possible breach in security. Getting a receipt instead of money directly, can leas to a situation when the same receipt can be possibly used several times, or can be faked. In this case, receipt is not that different from the paper money, as it represents value, yet has less security signatures in it. While most of the retail stores dealt with this problem in their own way, some until recently did not take it as their priority. As they assumed, the cost of any security check would be bigger than possible losses from small receipt misuses. Unfortunately, recently there has been recorded more and more cases of exploiting this breach, thus resulting in possibly significant money losses for the vendors. Therefore, one of the large retail chains in cooperation with Tomra, decided to build a system, one that will ensure the validity of all receipts from the RVMs.

The goal of this project raises a lot of questions and brings challenges. First of all, the large scale of the project. Some of the retail chains have hundreds of stores that operate in several countries, with thousands of reverse vending machines with different configurations and of different generations. The usage of data storages is expected to be extensive. As well as search in this data. We need to ensure not only smooth data handling from all these machines simultaneously, but also provide near 100% availability of the service in a real-time access manner. This requirement has crucial value for retail chains, since it is related to the image of the chain, customer relations and loyalty. No one wants to make customers of grocery shops wait for the validation of receipt, when the system is not responding in its expected time frame. The possible loss of couple pennies from recycled bottles is nothing compared to dissatisfied customer that was made to wait.

4.1.2 Solution proposal

Validation of receipt will follow this procedure: when customer submits containers from beverages to the reverse vending machine, the data about containers, date, refund value and other sent to the Tomra, as it always does for reporting and monitoring reasons. All machines are always connected to the intellectual network of Tomra, which oversees the status and data coming from the machines. At the same time, reverse vending machines print out receipts to the customers, with similar data encoded in the barcode. After that, this receipt is taken to the cashier where the barcode scanner reads the encoded information. At this point, this information should be sent to the validation system, where it is compared against the data received earlier from reverse vending machine. Depending on this comparison result, we get approval or refusal of payout of refund value. There are a lot of things that should be taken into consideration in this scheme. That is why it is extremely important to ensure the validity and availability of the data in an efficient manner and available within strict time constraints.

4.1.3 Disclaimer

Project at Tomra AS that we discuss here, is a commercial project that has been executed by Tomra AS and their partners. I was part of the team of external consultants from Altran Norway AS, that were invited by Tomra AS to help accomplish before mentioned task. All rights and ownership of developed solution belongs to intellectual property of Tomra AS.

Although, solutions that we will describe later, as well as code samples were inspired by real application, they do not represent original solutions that were developed by Tomra and consultants from Altran. They are simplified and framed examples that are used to demonstrate features of NoSQL databases that were used in production, compared to RDBMS solutions.

Chapter 5

Implementation

During the process of software system design there are multiple approaches that exist when it comes to the system architecture. One of the most commonly utilized approaches is domain-driven design, a process where domain of the problem constitutes the ways it should be designed and solved. For this project we have extended this approach into the domain-data-driven design, a data-centric process of modelling of the architecture. Data-centric solutions are very common in industries where business is heavily driven by information, thus actuality, responsiveness and reliability of it is crucial.

Next step of design process is to base our decisions upon information about possible use cases and main actors. The choice of a data storage is extremely important, as it will influence the flow of development process, all the way from design part until evaluation. This section describes a simplified and shortened implementation process that occurred during the executing project at Tomra AS. In the following sections we will present outcomes of two different paradigms when approaching problem of designing a system architecture.

5.1 Overall project architecture

The goal of the project is to prevent misuse of receipts from reverse vending machines. In order to achieve that we have to implement a simple validation mechanism that compares data from receipt against data, in authenticity of which we can be absolutely certain. System responsibilities are summarized in the following list:

- Persist data from reverse vending machines every time customer uses them. We call it consumer data or a consumer session, since data is an output of one to several empty containers that customer submits to reverse vending machine. One single session is completed by pressing a finish button. Receipt for the session is issued right here as well.

- Analyse and decode incoming data from receipts. Incoming data usually arrives in form of number sequence that was encoded into the barcode.
- Match data from receipt with data available in the database. Here we perform extensive search by many parameters in a storage of millions of records.
- Depending whether search result has been successful or not, give a response whether receipt data is valid or not.
- Persist receipt data and mark it as the one that has been processed. This is performed in order to avoid duplicate checks.

Increasingly growing data

From the aforementioned list we can clearly identify the most data-heavy operation - the search by parameters in the database where we expect millions of records. The challenge is that this data will grow exponentially as it will write all consumer sessions from all machines that Tomra owns on daily basis. There are many ways to optimize the process in order to avoid excessive data search. One of the possible optimizations is to archive old data. We assume that customers will likely claim money at the same moment they have been issued the receipt, right after submitting empty containers.

Configurability requirement

Another way to avoid unnecessary data writing and search, however, may change the way we design our system. In the beginning we were assuming that all machines that belong to Tomra will be part of the new receipt validation system by default. It is not, however, the real case yet. First of all, as we already mentioned every retail chain decides how to deal with receipt frauds in their stores independently. That means that the decision to use new system should be done by the owners of the chains. Up to the moment receipt validation system was designed to be used in stores of the one particular retail chain. From the developer perspective of view it means that we have to implement a mechanism to record consumer session data from machines that operate in the stores of one particular chain, with possibility to easily add other chains and stores when necessary.

Geographical location, such as region or country can also determine whether the system should be in place. A lot of large retail chains operate in several countries and they might have different policies in different places. There is a real case when particular number of frauds have been detected in Sweden and one retail chain wants to place validation of the receipts there. It also operates in Norway, however, since cases of abuse have not been recorded so often, it will be not necessary to put validation just yet.

Other machines might be exempt from the validation system because of technical and other constraints. For example when the model of reverse vending machine is too old and therefore does not allow to print out the barcode in the required format. All aforementioned requirements show us a necessity of the mechanism that will quickly determine whether we should process, persist and operate data that comes from certain machine in a particular store of a chain in some region. As the result, designed system should have following responsibilities in addition:

- Keep track of all machines in the network.
- Be aware of contextual information, such as where machine is located, which store it belongs to, in which chain and in which country.
- Be able to set machines on and off from the validation system.
- Being asked about particular machine be able to respond in a real-time fashion whether validation is enabled or not for this particular machine.
- Being able to remove existing and add new machines to the network.

Having all this additional features, we can clearly see that we have two areas of responsibilities here. First is to store and validate consumer data, second is to keep track of machine network. In order to address these two problems we have to create two separate applications which we call **Receipt Validator** and **Machine Network**. We should note that here we are not talking about databases. These two systems can be built on top of the one or multiple databases. The separation is based on the two different responsibilities these subsystems have.

5.1.1 Receipt Validator

Receipt Validator's responsibility is to handle receipts. First, we need to record consumer data from RVMs. Second, we have to validate incoming data from receipts. *Consumer session* is the name given to the data about consumed(accepted) empty containers. Usually one session contains information about number of consumed containers, amount of money that should be refunded, timestamp and also some information that will help to identify the machine itself.

Etalon data

Information about each session is sent to the intellectual systems of the Tomra and used for monitoring, statistics and other tasks related to the main Tomra business. We are also interested about this information since it will give us a foundation for validation of receipts. Consumer data will become our example data that we will check data from receipts against. When the customer finishes his session of submitting containers, machine will issue him a receipt with a barcode that will be read by POS terminal.

7 046225 034643

Figure 5.1: Sample barcode

Essentially, the information that coded in barcode should be the same as in the consumer data. It contains information about the amount of refunded money, machine itself and timestamp. But since the format of encoded information is decided by the client company of Tomra, it does not necessarily correspond to the consumer data completely.

Decoding of the barcode

If we take a closer look at the typical barcode in Figure 5.1, we can find that simple barcode can contain fairly large amount of data. First two digits can be used to identify that barcode belongs to certain chain or used to identify which purpose this barcode serves. Next three digits are identifiers to the particular store in the chain and reverse vending machine id inside this store, the one that issued this receipt. Next four digits have information about month and the day receipt has been printed. Information about a year can take two digits, but even one digit will be sufficient. In our case number 4 will indicate the current year of 2014. That would require us to archive our receipt data at least after 10 years of usage. However, archiving of old data will most likely happen much sooner than 10 years. Last digits can be used to indicate the refund value.

Receipt identity

The information that we have describes is still not sufficient to determine exact consumer session. A possibility exists that same machine will issue a receipt for the same amount of refund value in the same day more than once. In this case our search by parameters will return several records which is wrong by design. To address this problem it is necessary to introduce a four digit receipt serial number in addition. It will contain an id of the receipt during the day and will be reset after every twenty four hours.

Description we provided in this section is rather explanatory. The way data is encoded is decided by a contract between companies. There could be different formats and ways data can be written in a barcode. The baseline is that we can get enough information to find exact record about the consumer session.

5.1.2 Machine Network

The responsibility of the Receipt Validator is to record consumer data from RVMs as well as validation of the receipts from POS terminals. However, not all the data from RVMs is supposed to end up in the validator. There are many thousands of machines that are connected to the Tomra network which belong to the hundreds of stores from different retail chains. Not all stores, chains and machines are supposed to be part of this project, at least initially.

Switch mechanism

Simple flag mechanism on the every machine record, will help us to identify if the particular machine or data from it supposed to be validated later on. It will help us to avoid recording millions of redundant records. When need to know if receipt validator is enabled for particular machine or not the information should be available within milliseconds. We can not afford slow check, since consumer sessions are coming from the machines every second from different places. At every point we should have information about whether we should drop incoming consumer data or save it for future validation. The administration of such settings (enabled/not enabled) should on contrary be fairly easy even for non-technical personnel to be able to switch on/off machines from the validation network. For this sole purpose we have built a Machine Network Service.

5.1.3 Performance requirements

The are certain requirements that are provided by retail chains in order to have this system approved. One of their main concerns is the response time. The idea is that customer usually comes with the receipt, issued by reverse vending machine inside the store, to the cashier after he or she has done some shopping as well. It is very important for the cashier to process the client as quickly as possible. It would be unacceptable to make a client wait for validation of the receipt which might have refund value of several pennies. Satisfaction of store clients is the first priority for a chain.

During the process of the system design we should keep in mind performance requirements and plan ahead about how can we guarantee them. Requirements from one of the big retail chains, customers of the Tomra are presented below:

- System should be able to process 1000 requests at the same time
- Response time should never exceed 6 seconds
- Uptime of the system should not be less than 98%
- Logs and error messages should be available for analytic and monitoring reasons

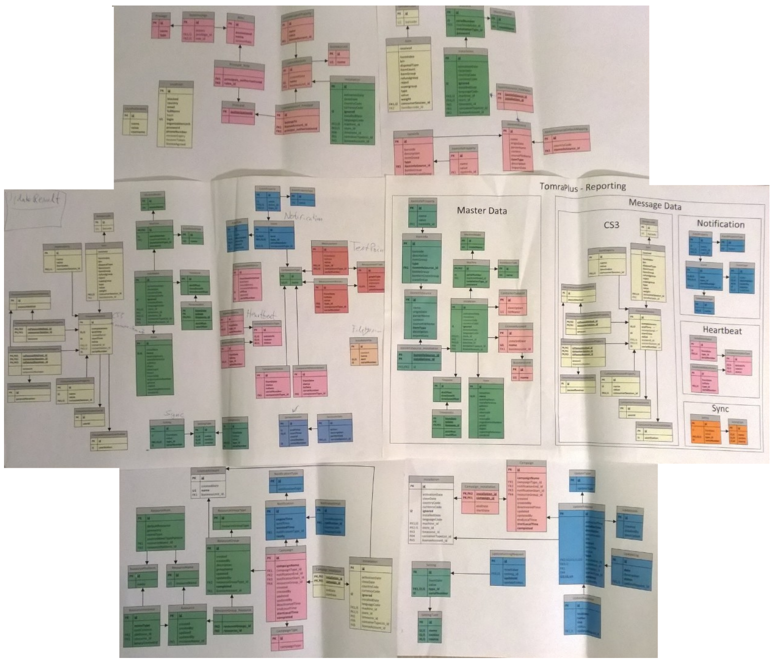


Figure 5.2: Monolithic database architecture

5.2 Relational Implementation. Legacy approach

5.2.1 Monolithic architecture

Most straightforward way of designing a system would be to think of a it as the whole and give the system all the necessary responsibilities. It will perform all basic operations and be designed in non-modular way. In this case we would have a central data storage, most likely using some relational database management system. Mentioned design is called monolithic which is meant to suggest the fact that it is *massive*. As the result, such structure to a large extent is *intractable*, meaning that altering individual components of the system will be difficult[7].

Tomra systems already incorporates monolithic system architecture. It collects all information from all the machines in one data storage. Processing, analysing and monitoring of data happens in the same place as well. Figure 5.2 illustrates the scale and size of the processing database. It seems reasonable to use all the information that Tomra already collects to keep track of consumer session data. In this case we will just need to add one more layer of responsibility and access point where we can validate barcode data against the data in the main data storage.

5.2.2 Data storage for Receipt Validator

Microsoft SQL Server is a relational database management system developed by Microsoft. Its primary query languages are T-SQL and ANSI

SQL. SQL Server is without doubt one of the most popular RDBMS solutions available in the market, with large number of enterprise customers throughout all possible industries. Tomra AS uses MSSQL as their main data storage solution.

Data fields

In order to incorporate our solution into the existing, we have to identify the data that we need to store and how we will interact with it. Central information that we need is the consumer session sent by the reverse vending machines. The data they send among other consist of:

- Date and time of the beginning of the session.
- Date and time of the end of the session.
- Refund value (in local currency).
- Reverse vending machine serial number.
- Receipt serial.

Since we chose the RDMBS solution to store the data, mentioned items will be our columns in a table called "ConsumerSession". We will also need to have a table called "Receipt" to store information that we get from receipts.

- Date the receipt has been issued.
- Refund value (in local currency).
- Reverse vending machine internal id.
- Store id.
- Receipt serial.

Note that RVM serial number we get directly from machines and RVM internal id that encoded in the receipt, are not the same. *Serial number* is unique number issued to all machines by Tomra. *Internal id* is an arbitrary id that is given to the machine inside the store. It is unique inside the store, but is not guaranteed to be unique between all the machines. Due to this fact, in order to identify a machine, we have to use a combination of the *store id* and internal machine id inside the receipt. These values are our input parameters for Machine Network, which will be responsible for mapping store and internal machine id numbers to the machine serial numbers.

Status handling

In addition, we also need to keep record of statuses of the receipt. We need statuses for several reasons. First of all is to avoid multiple payouts for receipts. To ensure that we distinguish new and processed receipts we need to have at least two statuses: NEW and CONSUMED. When we look up in a database for receipt match, if it has been already CONSUMED, meaning that money have been already paid out for this receipt, then we will refuse the second payout.

There is still a possibility for another error due to distributed nature of the process. We should account for possible network loss, meaning that system can get validation request from the point of sale, but response holding an information about validity or invalidity of the result may be lost due to the connectivity or other network problems. We might have a situation when receipt has been marked Consumed, but point of sale never got approval for the refund value payout. All future requests will be marked as duplicates and hence rejected. To avoid this situation we have to have two-factor verification. First we send request and mark receipt as RESERVED. Once we got a response and confirmed the payout (or decline of payout), we send another request with different meta-data that will finalize receipt and mark it as CONSUMED. RESERVED in this case is temporary and intermediate status to ensure correctness of the data. If there were no further requests after receipt has been RESERVED, after certain timeout it will be marked as NEW again and be available for new validation requests.

5.2.3 Implementation of Receipt Validator

We have already identified a database design with two tables that will be required to implement such solution. We can also note that information inside "ConumserSession" and "Receipt" tables is quite similar. Seemingly we could merge two tables into the one. However, we need to distinguish two types for the logging and monitoring purposes. We need to have a possibility to monitor consumer sessions by the given period and also see logs of the receipts in order to identify possible problems. The whole process is described as following:

1. We receive a consumer session data from the reverse vending machine.
2. Using Machine Network we check if this machine is in the network. In case if it is not, we skip the data and receive the next consumer session.
3. We record consumer session data fields inside MSSQL database in the "ConsumerSession" table.
4. After submitting containers client of the store proceeds to the cashier and asks for refund money. At this point, when cashier scans barcode on the receipt, there is a *validate* command sent to our system.

5. Barcode is decoded and all vital information that identifies a consumer session is fetched

6. First, we look up for records inside "Receipts" table, since we want to be sure that this receipt has not been already used. If we do not find any records or we found one record with NEW status we can safely proceed.

7. As the next step we search in the "ConsumerSession" table by parameters. In case if there are no matches or more than 1 record found in "ConsumerSession" we return error and receipt is refused to be paid out.

8. In case if we found exactly one corresponding match in the "ConsumerSession", we record a new entry in the "Receipt" table, with all the data and RESERVED status.

9. After sending a response to the cashier, the money are paid out and *consume* request is sent to our system. At this point the entry is marked as CONSUMED and secured against duplicate requests.

10. In case when our response is lost during the connectivity, after some time RESERVED status is changed to NEW and available for another attempt of the validation

From this list we can identify that we need at least two database lookups in order to find corresponding match of the receipt data. It is highly inefficient as we expect very large samples of data in both of the tables and they will grow very quickly on the daily basis.

Denormalization

Since we have a lot of duplicating data in both tables, we violate the normalization rules in MSSQL. Database normalization is the process of organizing the fields and tables of a relational database to minimize redundancy. This is usually considered a bad design, however intentional denormalization can be performed for performance purposes, like in our case. The drawback of such design, is that if we need to make changes in one table, it is quite possible that we also need to make changes in another to avoid inconsistency of data.

One of the possible ways to optimize queries is to link two tables with one-to-one primary key - foreign key relationship between "ConsumerSession" and "Receipt" tables. This will help us to avoid double database lookup, in those cases when we already have record from one of the tables. However, it does not improve our main case, when we get first validation requests and thus do not have record in Receipt table yet. We are limited by the RDBMS constraints.

5.2.4 Data storage for Machine Network

The central part of the Machine Network is the data storage that contains information about all the machines in the network and their statuses with regards to the Receipt Validator. The most natural way to organize data in the database is to mimic real-life structure. In our case all machines usually installed at the stores that operate and maintain them. Those stores are usually part of the big retail chain that oversees the stores and operations in them. On top of that some of the chains operate in several countries and as the result, due to possible differences in legislations, rules and other reasons, their stores should be divided in different regions in order to be able to make amendments that will affect only particular region.

Modelling graph into the tables

Described structure may remind us a graph structure with "Chain" as a root node, "Stores" and "Regions" as child nodes and "Machines" as leafs. First, we should look into the common search queries that we are going to execute here. In our case the most common operation will be to find a certain machine or rather its status (enabled/disabled) if we know the store where this machine is located, which chain does this store belongs to and in which region and machine id within this store. Given all this information we should be able to fetch individual store in a least possible time. In this section we will describe how we can use SQL data storage to model the data and implement this solution

5.2.5 Implementation of Machine Network

If we choose to store the data in RDBMS way, we need to have separate tables for each of the entity types. We need to create "Chains", "Regions", "Stores" and "Machines" tables. Luckily, we have only one-to-many relation, therefore we do not need additional tables to provide many-to-many relation. In appendix the readers can find SQL listings that were used to create the test database (Listings A.1 - A.4). As the result of those queries we will get region and chain tables with populated data in them. We create a store for each combination of region and chain, also using one of the available unique addresses for stores, resulting in different stores described by region, chain, address and name. In the final step we generate certain amount of machines per each store.

Search query

Let us take a look at the common search query. We need to find the machine that is located inside the store that belongs to chain with name "Bunnpris" in Norway with address "Blindern". We know that inside this store it has internal id equal to "TMR1" (all this information directly or indirectly we get from the receipt barcode). To find this machine we need to execute the following SQL query:

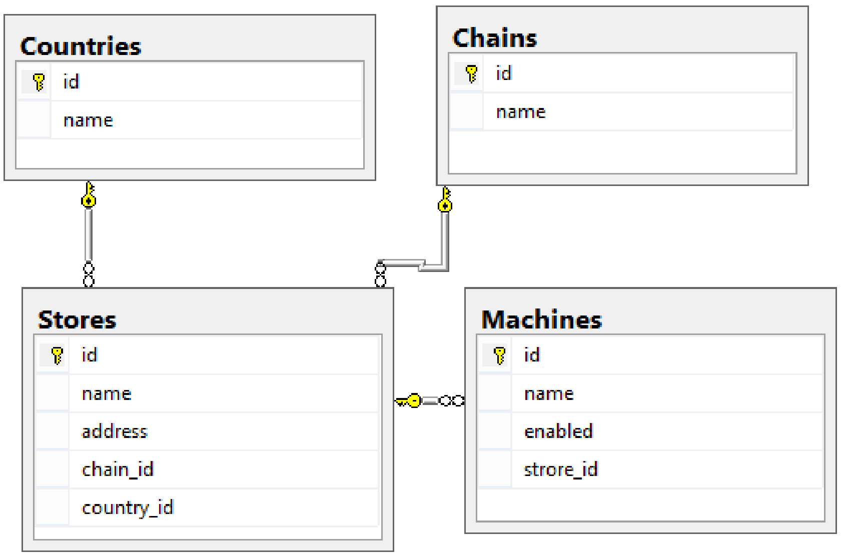


Figure 5.3: MSSQL structure

Listing 5.1: SQL query style search

```
SELECT *
 FROM [dbo].[Machines]

 INNER JOIN dbo.Stores ON Machines.strore_id = Stores.id
 INNER JOIN dbo.Chains ON Stores.chain_id = Chains.id
 INNER JOIN dbo.Countries ON
         Stores.country_id = Countries.id

 WHERE Countries.name = 'Norway'
         and Chains.name = 'Bunnpris'
         and Machines.name = 'TMR1'
         and address = 'Blindern'
```

From this query we can clearly observe a performance problem. For this and virtually almost all operations that require information about machines in context, we need to have joins on many separate tables. This is a common problem when the data that we map into the database does not follow the structure of it. Every join slows down performance drastically.

5.2.6 Summary

Incorporating new applications in the monolithic architecture is not an easy task. For simplicity reasons we have talked about implementation of the data storage for Receipt Validator and Machine Network as if they were in a separate database. In reality, it will require mentioned information to be part of the bigger database. We might have to reuse some of the existing tables Tomra already has, since adding new tables with similar data contributes to the denormalization and data inconsistency.

Flexibility issue

Tomra has been in the market for more than thirty years and operates in more than forty countries throughout the world. To use, search and operate with the data that has been accumulated so far is a great challenge. It is difficult to isolate data that we need from data that other services require. Consumer sessions used for all kind of reports and monitoring purposes. Archiving data and any kind of manipulation with it is also not feasible for the same reasons. Search in a large quantities of redundant data will effect performance drastically.

Data congestion

Data congestion is another major issue. Same data and database used by many services and tools inside Tomra and it is not guaranteed that validation request will be processed in the expected time frame. Some of the reports that use consumer session data are known to take up to several minutes and may hinder the validation requests. However, we can not prioritize some requests over another. Hence we do not have control over the execution time when it comes to data which is used by many services.

The longer monolithic system exists, the more it grows with interdependencies, redundant data and responsibilities. It is harder to reuse available components, refactor existing code and find errors and faults in a large system.

5.3 NoSQL Implementation. Separation of concerns

5.3.1 Distributed architecture.

In order to meet customer requirements in efficient manner we have to think about ways to guarantee quick response time and 100% availability of the service. To ensure that we have this kind of control, we have to create a standalone service that will work separately from the other intellectual services of Tomra, yet reusing some of them if that is possible.

Separation of concerns

There are two sets of responsibilities that have to deal with the receipt validation. One area is dealing with recording consumer session data and validating receipts. Another keeps track of machine network within Tomra. Those tasks are different in their nature and do not overlap in their functionality. The best way to approach both problems is to have separate subsystems with separate sets of responsibilities with some common API to interact with each other.

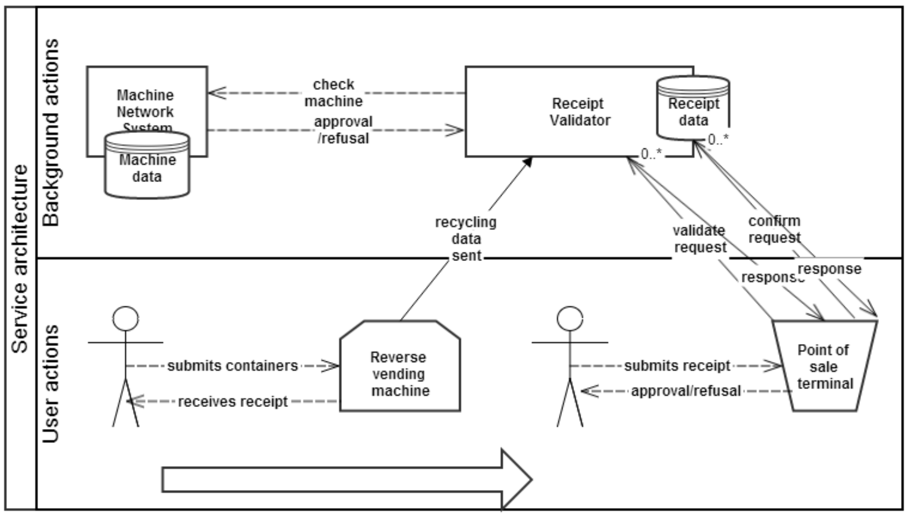


Figure 5.4: Distributed project architecture

Separation of concerns achieved by dividing our system into two subsystems with different responsibilities, architectures and databases. From here we will talk separately about **ReceiptDB** database which will store and validate consumer data and **MachineNetworkDB** database which will be responsible for having up to date information about machines in the Tomra network and their statuses (Figure 5.5).

Distributed architecture

Aforementioned design decision was made in accordance with Tomra's strategy of overall moving all intellectual services of Tomra from monolithic architecture to distributed one. There are a lot of advantages of such architecture - speed and reliability are the ones that of our main concern, as well as possibility to narrow down the search frontier of the problems when they occur. According to Coulouris et. al. [7], a distributed system should make it possible to:

- Allow the software implementing any particular service to be changed independently of the other facilities.
- Allow alternatives of the same service to be provided, when this is required to suit different users or applications
- Introduce new services without harming the integrity of existing ones.

5.3.2 Implementation of ReceiptDB

To improve performance and consistency of the data we we need a way to fetch all required information in one database lookup. Yet, we should still

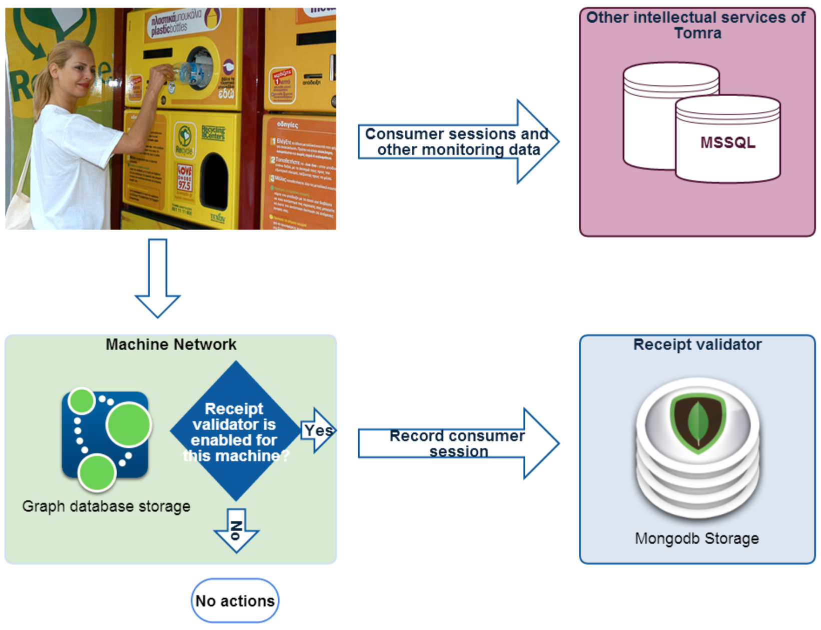

Consumer sessions and other monitoring data
Other intellectual services of Tomra
MSSQL
Machine Network
Graph database storage
Receipt validator is enabled for this machine?
Yes
No
No actions
Record consumer session
Receipt validator
Mongodb Storage

Figure 5.5: Two areas of responsibility

have the possibility to distinguish two different types of data (consumer sessions and receipts) for logging and monitoring purposes. To achieve this we might need to look at the structure from the different angle. In a sense "ConsumerSession" type is subtype of the "Receipt". "Receipt" contains all the data that "ConsumerSession" has. In addition, it has some arbitrary data to identify receipts and statuses. In order to distinguish two types, yet provide interface to fetch Receipt and ConsumerSession records separately in one query we can to use document-base database like MongoDB.

Embedded documents

Data in MongoDB has a flexible schema. Collections do not enforce document structure. Decisions that affect how you model the data can affect application performance and database capacity. MongoDB has a data model that uses embedded documents to describe relationships between connected data. Consider our example that maps "ConsumerSession" and "Receipt" relationships. In this one-to-one relationship between the "ConsumerSession" and "Receipt" data, the "ConsumerSession" belongs to the "Receipt". In the normalized (or RDBMS) data model, the "Receipt" document contains a reference to the "ConsumerSession" document.

Listing 5.2: Normalized structure of database

```
{
    _id: "receipt1",
    status: "NEW"
}
```

```
{
   receipt_id: "receipt1",
   datestart: "29.07.2014␣14:52",
   dateend: "29.07.2014␣14:52",
   machineserial: "98767687324",
   receiptserial: "12345"
}
```

If the receipt data is frequently retrieved with the consumer session information, then with referencing our application needs to issue multiple queries to resolve the reference. The better data model would be to embed the "ConsumerSession" data in the "Receipt" data, as in the following document:

Listing 5.3: Embedded document

```
{
   _id: "receipt1",
   status: "NEW"
   consumersession: {
           datestart: "29.07.2014␣14:52",
           dateend: "29.07.2014␣14:52",
           machineserial: "98767687324",
           receiptserial: "12345"
        }
}
```

With the embedded data model our application can retrieve the complete receipt information with one query. This example illustrates the advantage of the embedding over the referencing if we need to view one data entity in the context of another.

5.3.3 Implementation of MachineNetworkDB

As we have already mentioned, the structure of the data that we should model remind us the graph structure, with "Chain" as a root node, "Stores" and "Regions" as child nodes and "Machines" as leafs. Neo4j was used for graph implementation of this database. Here we focus on the implementation part and will describe the advantages in the next chapter

Architecture as query

It is very easy to create a database model with Neo4J database. We need to create nodes of type "Region", nodes of type "Chain" and generate stores that have different addresses, belong to different chains and located in different regions. After that we populate nodes of type "Machine" and associate them with respective nodes of type "Store". We will end up

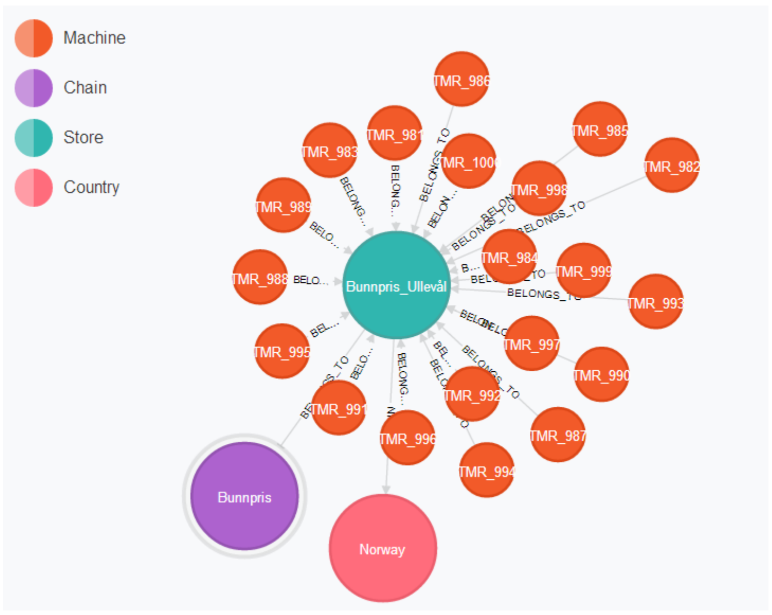


Figure 5.6: Graph Structure

with the following structure that can be described in Cypher, Neo4j query language:

Listing 5.4: Structure of the database depicted in Cypher

```
(Machine)−[:BELONGS_TO]−>(Store)−[:BELONGS_TO]−>(Chain)
(Store)−[:IN]−>(Region)
```

It is interesting to note that it does not only describes the structure of the database. With small changes we can make it a valid query for Cypher that returns all the entities from the database. Limiting the return data size by twenty nodes, we will get visualisation of the graph structure as depicted in Figure 5.6.

Listing 5.5: Cypher Query that returns all nodes and relationships

```
 MATCH (m:Machine)−[:BELONGS_TO]−>
           (s:Store)−[:BELONGS_TO]−>(c:Chain),
           (s)−[:IN]−>(co:Country)
 RETURN m,s,c,co
```

Similarly to SQL approach we create necessary test data in Neo4J database (see Listings A.5 - A.8 in the appendix for detailed queries). After that we can execute query described in Listing 5.6 in order to find a machine by chain, region, address and personal id.

Listing 5.6: Cypher query style search

```
MATCH (m:Machine)−[:BELONGS_TO]−>
          (s:Store)−[:BELONGS_TO]−>(c:Chain),
         (s)-[:IN]->(co:Country)
WHERE  m.pid = 'TMR_1' and s.address = 'Blindern' and
         co.name = 'Norway' and c.name = 'Bunnpris'
RETURN m;
```

Quick glance to this query allow us to see that it is very similar to the query that represents our graph structure with additional WHERE clause that narrows down the search to the specified parameters.

Chapter 6

Evaluation and experiments

In this chapter we evaluate NoSQL solutions that we have used to build **ReceiptDB** and **MachineNetworkDB** data storages for Receipt Validation and Machine Network systems. Our assessment involves determining whether the software and the project responsible for developing it conforms to various characteristics or exhibits various qualities that are expected of efficient and *evolvable*[24] software that is the ability of the system to perform well under given conditions and to be cost-efficient in its maintenance and the addition of new features.

In the following section we will focus more on the performance aspect of using NoSQL. For many enterprises, good performance is one of the main conditions for the choice of the data storage. To answer on the MRQ we should provide enough evidence that using NoSQL is at least not worse than using any other SQL database in terms of performance. We will explore the requirements that were laid on the system and will argue how NoSQL helped us to fulfil those.

We also describe the experiments we performed to compare performance of processing moderately connected data by Neo4j against MSSQL Server. We emulated the data structure to be in accordance with requirements of the industrial case. The idea of evaluation is to understand advantage(or disadvantage) of using Neo4j for cases that do not require processing big or highly connected data, like in our case.

6.1 Evaluation

In this section we evaluate MongoDB and Neo4j from the enterprise perspective and take a closer look to such properties as usability, maintainability, ease of use, learning curve and licensing. All these properties are important during the development process and directly affect the costs and time frames involved with development.

First, we discuss properties and our understanding of the criteria we use for the evaluation. After that we measure and evaluate NoSQL solutions based on these criteria. We assess both, technology we used to build our solutions and developed systems themselves: ReceiptDB that uses MongoDB and MachineNetworkDB that incorporates Neo4j for its main data storage.

Our evaluation is based on the criteria of a well-performing and evolvable system, as well as expectations of the Tomra and their partners:

- Performance
- Scalability
- Ease of use
- Integrability
- Cost Efficiency

6.1.1 Scalability and Performance

First, we should define what we mean by scalability and performance and how they differ. **Performance** refers to the capability of a system to provide a certain response time, host a defined number of users or process a certain amount of data. In section 6.2 we explore the topic of performance in greater detail.

Scalability refers to the characteristic of a system to increase performance by adding additional resources (such as additional servers for example). When we realize that our performance requirements change (e.g. we have to serve more users, we have to provide lower response times) or we cannot meet our performance goals, we have to deal with the scalability issue.

Scalability in MongoDB

In order to ensure the future maintainability of the designed system, we have to make sure that we have possibilities to guarantee same or better response time in the case of explosive growth of data or new requirements. To scale its performance MongoDB uses a sharding, which is the process of splitting the data evenly across the cluster of servers with parallel access. This is implemented by breaking the MongoDB server into a set of front-end routing servers (mongos) that route operations to a set of back-end data servers (mongod). This way large data sets are split across multiple nodes, keeping shards balanced as new records and/or nodes are added to the system. We did not use this technique in our case yet, because our architecture and data size did not require it. However, knowing that horizontal scaling will be as easy as it is done in MongoDB allows us not to worry about the future performance and gives us advantage in the planning process.

Scalability in Neo4j

Neo4j on the other hand does not support sharding. Yet it has massive scalability - it can handle graphs of several billion nodes/relationships/properties on a single machine[28]. With ability to traverse depths of 1000 levels and beyond at millisecond speed is a guarantee that performance will not fall even if we need to add new entities and relationships into the database. That means that Tomra has a flexibility to change database schema without risk of affecting performance. We can add new entities or attributes to the database, such as general node location, if Tomra will deploy their machines not only in the stores but in the public places, offices and other types of spaces. Such addition will require to have deeper graph structure, which still be computed by Neo4j in an efficient manner.

6.1.2 Usability and maintainability

Usability describes how well systems are supported by frameworks and common programming libraries. Possibility to find comprehensive, appropriate and well-structured documentation or help from community. Understandability of how the software works and easiness to use basic functions[15].

Both MongoDB and Neo4j have great shell applications, allowing to do simple tasks very quickly and efficient. Neo4j went further and ships with web-gui console out of the box (Figure 6.1). Simple web interface allows developers to visualize the graph data on the fly. It is a great tool that gives a lot of value when modelling and testing graph structure and queries. It can also become a very useful tool of learning about Cypher query language

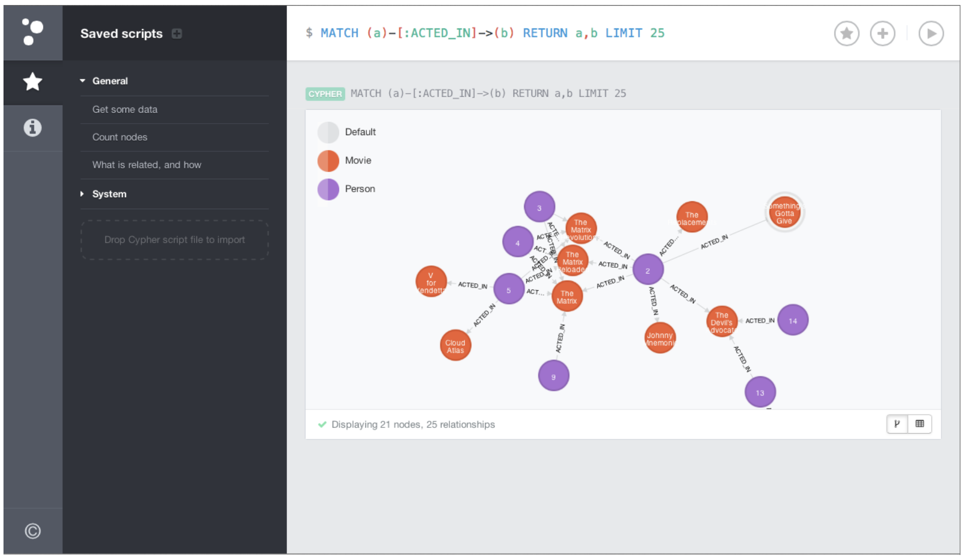
Maturity

During the implementation of **ReceiptDB** and **MachineNetworkDB**, we found that both technologies MongoDB and Neo4j are mature, well-documented and well-supported databases. It was easy to find accurate and appropriate information about features of the database. Following basic steps provided by documentation it is easy to install and incorporate databases into the application. Strong community support and wide-use ensures that we can get information and help about specific cases or problems.

Learnability

When describing a data model, MongoDB uses notation that resembles *JSON* - JavaScript Object Notation, Python dictionaries or Ruby hashes. This notation is familiar to a lot of developers, since it became another standard of data representation after *XML*. This is a rich data structure capable of holding arrays and other documents. It is very easy to operate with documents and query language is intuitive and powerful.

Neo4j provides a very thorough, yet easy to understand introduction



```
$ MATCH (a)-[:ACTED_IN]->(b) RETURN a,b LIMIT 25
```

```
CYPHER MATCH (a)-[:ACTED_IN]->(b) RETURN a,b LIMIT 25
```

```
Displaying 21 nodes, 25 relationships
```

Figure 6.1: Web interface for Neo4j console

course after which one can simply use the power of the graphs at ease. A lot of features of the query language Cypher are inspired by SQL and will be familiar to a lot of developers. Featuring a lot of traits of functional programming the cypher is very powerful, yet simple language to master. It allows us to write queries that resemble a sentences in English language. In a Listing 6.1 even a person that is not familiar with cypher can derive that we try to Match all the nodes that conform to certain structure, such as Machines belong to Stores which belong to Chains and located in certain Countries. In addition, we return those nodes that correspond to certain properties, such as "located in Blindern, Norway".

Listing 6.1: Cypher query style search

```
MATCH  (m:Machine)-[:BELONGS_TO]->
        (s:Store)-[:BELONGS_TO]->(c:Chain),
        (s)-[:IN]->(co:Country)
WHERE  m.pid = 'TMR_1' and s.address = 'Blindern' and
        co.name = 'Norway' and c.name = 'Bunnpris'
RETURN m;
```

Cross-language portability

Both Neo4j and MongoDB technologies have bindings for a number of languages like Java, Python, Jython, Ruby and Clojure. There are number of ways to communicate via well-defined API. REST (Representational state transfer) interface is also available and recommended for use. Java has a number of frameworks that provide seamless integration with a database API. Spring Data framework for Java has well developed APIs that will allow us to incorporate MongoDB in our application easily and without much code. One of the nice features of Spring Data Repository class, is that we can create abstract interface class with virtual methods and we do not need to worry about implementation of those. Spring Data generates implementation of search queries for us based on interfaces we define.

Robustness to requirements changes

One of the common risks of software development is a change of requirements. Business always needs to evolve, so do software systems that are crucial for the business. Being able to change the data model of application without much effort is something that can can become a game changer. RDBMS is thriving in an environment when we have decided, static, relational data, that can be easily modelled into the number of tables. However, the moment we need to change our schema in already running database, we face a lot of challenges, such as how to change architecture of the database without breaking existing software and halting the business processes.

Flexibility

NoSQL databases solve this problem easily, by just not having schemes at all. By having such property we can easily change, adapt and restructure our database model according to new requirements and our needs. It is much easier to prototype the database and grow it with more functionality, whilst RDBMS do not forgive architectural mistakes early on. To maintain software that incorporates NoSQL data storage is much easier and safer. It gives us desired flexibility by almost no cost. We were able to focus on developing of application without spending too much time on the database architecture, since we knew that we can change the schema at any time and it will cost us very little. During the implementation we have amended the structure of the database several times and have been able to continue improvement without loosing the implemented work.

6.1.3 Cost reduction

In an enterprise environment the cost of introducing new software system is something that can decide the success of the system. Our aim is to achieve business goals by the least possible cost. Implementation of **ReceiptDB** and **MachineNetworkDB** has shown us that using NoSQL databases as main data storages can significantly reduce the cost of production. Both by having a cheaper licenses and reducing the time of developers involved.

Licensing flexibility

Both MongoDB and Neo4j are open-source solutions with flexible licensing. They provide free community editions that allow us not only to prototype solutions and test performance and suitability, but actually run a full-production scale data in an enterprise environment, since they do not have any large functional limitations compared to enterprise licenses. To our last knowledge, free community editions are still used for both implemented systems. Oracle and MSSQL also have free editions, however, they are severely limited compared to standard editions.

Oracle Database XE, for example will store only up to 11GB of user data, use up to 1GB of memory and use one CPU on the host machine[1]. These limitations allow prototyping and initial testing of database, but make real industrial use of them impossible. In the Table 6.1 we can compare pricing for standard licensing between MongoDB[2], Neo4j[3], Oracle[4] and MSSQL[5]. As we can see, even standard database editions of popular RDMBS solutions require significant budgets.

Database	Price for standard license	Price for standard license + support
MongoDB	free	on demand
Neo4j	free	12 000 USD[6]
Oracle	17 500 USD	21 350 USD
MSSQL 2012	3 189 USD	on demand

Table 6.1: Pricing for standard licensing per core

Efficiency in usage

Time of developers consumed by planning, implementation and maintenance was significantly reduced. By having a good documentation and simple installation process it was easy to start productive work with a new database. Both databases have good support community that allow us to solve emerging problems quickly. Flexible non-schema structure of the databases saved a lot of time on architectural and data modelling discussions. Being able to use the data models that intuitively follow the database structure saved up a lot of time, instead of trying to bend the data and hack it into the tabular representation. Good integration with popular programming languages and frameworks made development process simple and effortless.

[1]http://www.oracle.com/technetwork/database/database-technologies/express-edition/overview/index.html

[2]https://www.mongodb.org/about/licensing/

[3]http://neo4j.com/subscriptions/

[4]http://www.oracle.com/us/corporate/pricing/technology-price-list-070617.pdf

[5]http://www.microsoftstore.com/store/msusa/pdp/SQL-Server-Standard-Edition-2012/productID.281182400

[6]Price listed per year

6.2 Performance experiments

NoSQL database solutions are becoming more and more common in a world currently dominated by SQL relational databases. The idea behind NoSQL movement was to provide a tool to tackle large volumes of unstructured data. However, it is still unclear if it is advantageous or disadvantageous to use NoSQL database for moderate volumes of data which is the case for most of the world's small and medium-sized enterprises. There are few studies that compare the performance of processing a modest amount of unstructured data in a NoSQL database with a traditional relational database. MongoDB, for example was found to be equally performing well or sometimes better than the SQL Server, except cases when aggregate functions are utilized[25]. However, there are no studies that assesses performance of Neo4j graph database when processing data that is not so highly connected with a depth from a root node up to five levels.

Test setting

In order to compare performance of Neo4J graph database against MSSQL RDBMS, we have created instances of Neo4j Community Edition version 2.0.1 and MSSQL Express Server 2012. We have not been using any tweaks or measures to improve performance of any of those databases (e.g. heap and stack memory settings amendments). All tests has been done with instances "out of the box".

System information of the computational machine in test is following:

Operating System:	Windows 8.1 Pro 64-bit (6.3, Build 9600)
Processor:	Intel(R) Core(TM) i7-3667U CPU @ 2.00GHz (4 CPUs), 2.5GHz
Memory:	8192MB RAM
Disk Drive:	225GB SSD INTEL SSDSCMMW240A3L

6.2.1 Experiment results

Tests have been executed in the following way:

1. Test data of different sizes is populated (ranging from 20 000 elements to 2 millions)
2. For each size of the test data respective search queries are executed several (at least ten) times.
3. Out of the results the median value is selected as representative and added to the table.

To populate the data we used queries that are listed in appendix (Listings A.1 - A.8). After running the tests we got following results depicted in the Table 6.2.

Data size	SQL	Neo4J
20 000	9 ms	56 ms
200 000	66 ms	63 ms
400 000	117 ms	79 ms
600 000	138 ms	83 ms
800 000	167 ms	95 ms
1 000 000	205 ms	106 ms
1 200 000	216 ms	122 ms
1 400 000	235 ms	135 ms
1 600 000	285 ms	144 ms
1 800 000	293 ms	144 ms
2 000 000	316 ms	156 ms

Table 6.2: Time in ms, required to perform search query in respective db at a given data size

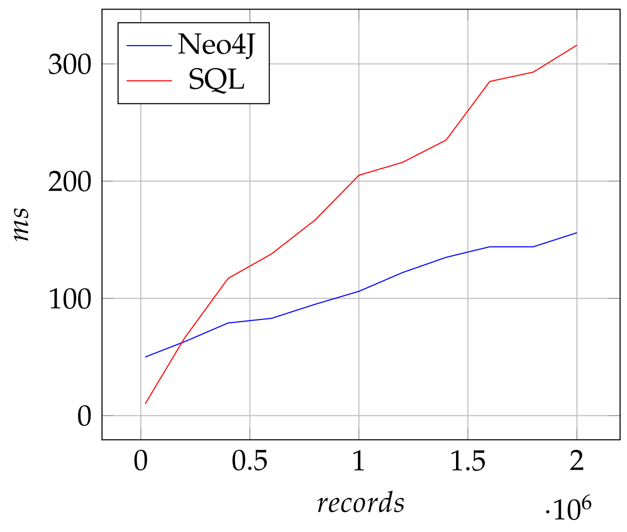


Figure 6.2: Performance of Neo4J against SQL

These results give us an interesting insight into the problem. Performance of the Neo4J database is far from the promised one (5x times better than MySQL according to Graph Databases book[28]). However, we should take into consideration that our problem is different from the typical graph problem like finding friends of friends. In addition to that, MSSQL is running in almost ideal situation with only one-to-many relationship, avoiding creation of additional tables just to provide many-to-many relations.

With small data set SQL is faster, due to initial overhead Neo4J has. However, at two hundred thousand data elements performance of Neo4J is comparable to SQL and it is twice as better with a very large dataset. This trend is well visualized in the Figure 6.2.

6.2.2 Indexes

As we already have mentioned, we did not used powerful means of query optimizations available in MSSQL, neither we used any memory tweaks to boost up the heap of the Neo4J. However, one of the easiest and most efficient ways to boost performance of queries is by the usage of indexes. A database index is a data structure that improves the speed of data retrieval operations in a database. As the cost it requires more storage space to maintain the extra information about the data. Indexes are used to quickly locate data without having to search every entity in a database every time a database is accessed.

There are a lot of possible ways to create indexes and there are also a lot of different types of indexes. In our case we will use simplest non-unique indexes for all properties (columns) that are used for 'where' condition. To create an index in Neo4J, first we need enable auto indexing for newly created nodes, since by default it is disabled. In neo4j.properties file we change node_auto_indexing to true, as well as providing list of property names that should be indexed.

Listing 6.2: Enable indexes in Neo4j

```
# Enable auto-indexing for nodes,
# default is false
node_auto_indexing=true

# The node property keys to be auto-indexed,
# if enabled
node_keys_indexable=name,pid,address
```

After that, we can create indexes on individual labels using following commands in the Neo4j shell:

Listing 6.3: Create indexes in Neo4j

```
CREATE INDEX ON :Chain(name)
CREATE INDEX ON :Country(name)
CREATE INDEX ON :Store(address)
CREATE INDEX ON :Machine(pid)
```

Similarly, we created indexes for columns that we are using for our search in the MSSQL.

Listing 6.4: Create indexes in MSSQL

```
CREATE INDEX IX_Name_Machines -- specify index name
  ON dbo.Machines (name) -- specify column name

CREATE INDEX IX_Name_Chains
  ON dbo.Chains (name)
```

```
CREATE INDEX IX_Name_Countries
  ON dbo.Countries (name)

CREATE INDEX IX_Address_Stores
  ON dbo.Stores (address)
```

Data size	Neo4J	Neo4J with indexes
20 000	56 ms	29 ms
200 000	63 ms	39 ms
400 000	79 ms	52 ms
600 000	83 ms	66 ms
800 000	95 ms	79 ms
1 000 000	106 ms	91 ms
1 200 000	122 ms	103 ms
1 400 000	135 ms	118 ms
1 600 000	144 ms	133 ms
1 800 000	144 ms	137 ms
2 000 000	156 ms	147 ms

Table 6.3: Time in ms, required to perform search query in Neo4j at a given data size

Data size	SQL	SQL with indexes
20 000	10 ms	9 ms
200 000	66 ms	59 ms
400 000	117 ms	103 ms
600 000	138 ms	135 ms
800 000	167 ms	158 ms
1 000 000	205 ms	183 ms
1 200 000	216 ms	202 ms
1 400 000	235 ms	225 ms
1 600 000	285 ms	236 ms
1 800 000	293 ms	252 ms
2 000 000	316 ms	275 ms

Table 6.4: Time in ms, required to perform search query in SQL at a given data size

There are a lot of options available for tweaking indexes further such as making them UNIQUE or NON-UNIQUE, CLUSTERED or NON-CLUSTERED, but for our experiment simple index would be enough to see the difference. Tables 6.3 and 6.4 show the response time results of Neo4J and SQL search queries with and without indexes.

If we try to analyse these results visualised in Figures 6.3 and 6.4, we can clearly see that indexes give immediate advantage for Neo4J database even

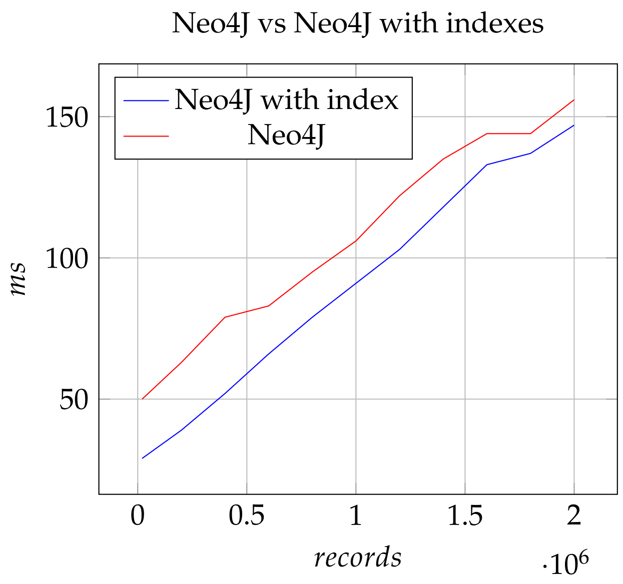


Figure 6.3: Performance of Neo4J using indexes

for small sizes of data. While indexes in SQL start to show their efficiency on rather large data samples. However, usage of indexes was advantageous for both databases at almost no cost except slightly larger disk space usage (1.29 GB against 629MB with 2 million records in Neo4J) which is not a problem in most of the cases when databases are used in the enterprise scale.

Let us now compare performance of Neo4J database with indexes against SQL. In Figure 6.5 we can clearly see that retrieval speed of Neo4J is still superior to one with SQL. For the sake of purity of the experiment we would like to mention, however, that at the very large data samples (1.8 - 2 millions) SQL showed rather anomalous results, having a retrieval time of sample search query of just 20-30 ms opposed to usual 250ms. This possibly can be explained as internal mechanism of MSSQL to cache pre-calculated results for often called queries. In these experiments we did not take those numbers in consideration. Future work might be done to analyse the behaviour of MSSQL databases with indexes and have better research in this regard.

6.2.3 Summary

Initial performance criteria for us was to meet the requirements from Tomra partners. In order to ensure acceptance of the system our goal was to guarantee:

- Response time within **6 seconds**.
- Processing of the **1000 requests** at the same time

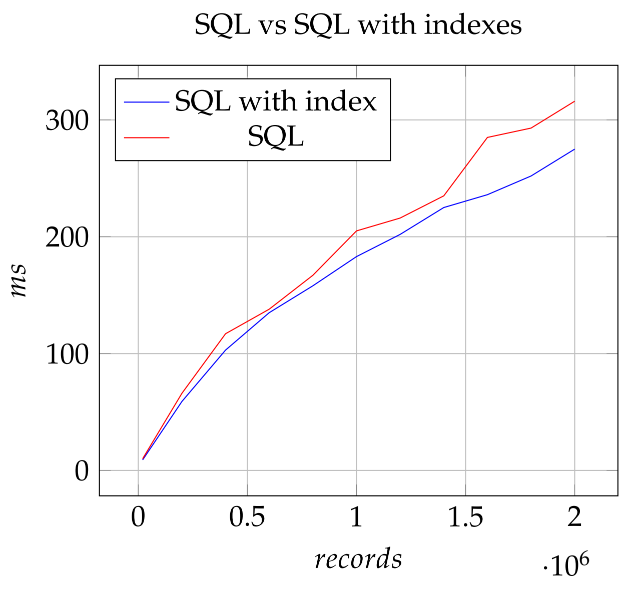


Figure 6.4: Performance of SQL using indexes

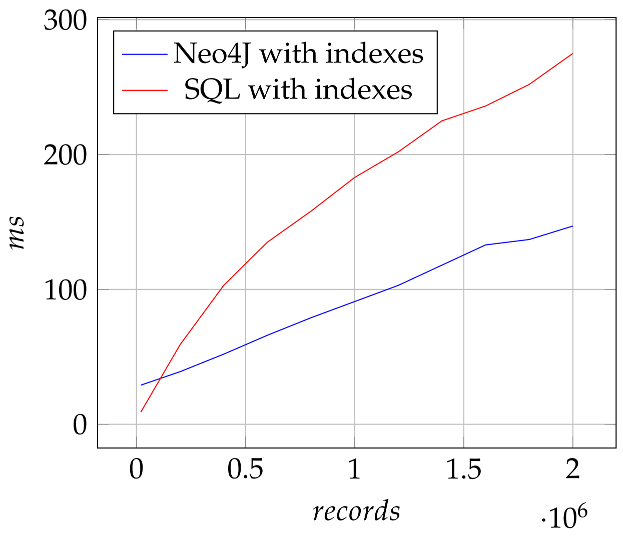


Figure 6.5: Performance of Neo4J with indexes against SQL with indexes

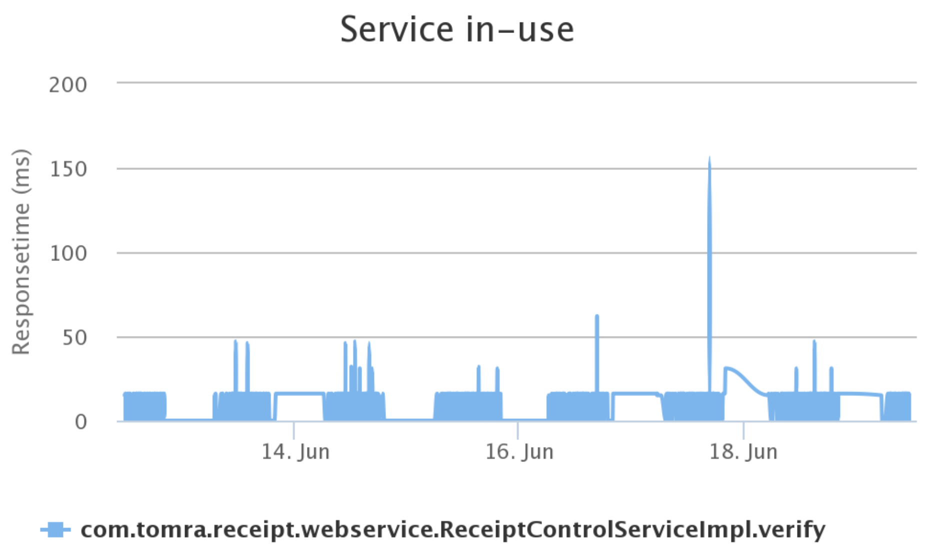


Figure 6.6: Response times on verify request

- **98% uptime** of the system

Performance of the designed system far out-performed all expectations. In Figure 6.6 we can see a period of time represented on x-axis that was observed during the current year. On the y-axis we can see typical response time values of the system expressed in milliseconds. We can observe that normally system under test processes request and returns response within 10 milliseconds with rare spikes up to 150 milliseconds, most likely due to interference with other data-heavy write/read operations that might occur at the same time. We tested system on data sizes ranged from millions to billions of records and it performed well within desired borders.

Outstanding performance results became possible due to several factors. One of the first things that we have identified, is that noticeable performance gains are achieved by the fact that different NoSQL databases can provide appropriate data structures for our applications, such as key-value, document or graph data stores. Due to possibility of having more suitable database model according to our needs we can avoid redundant data constructs or inefficient relationships. In practice we can very often represent in a single entity a construct that would require us to build several tables in order to be properly modelled in a relational database.

In case of **ReceiptDB** we could reduce the database structure from two tables to just one embedded document using MongoDB, yet providing full support of distinguishing two different entities: "Receipt" and "ConsumerSession". Instead of searching records in the join of two tables we utilize only one simple query that retrieves either part of the embedded document. Another part is retrieved for us 'for free'. This is the powerful mechanism that gives us desired performance gains and flexibility.

Similar gains are achieved by using Neo4j database for holding graph data of **MachineNetworkDB**. Instead of spreading the data between four tables and joining them again for search purposes we have a simple graph structure with four entity types and relationships between them. The depth of this graph is never more than five making search very quick and efficient.

Indexes are usually used to improve performance of the relational databases. However, both MongoDB and Neo4j have also full support of indexes. After using indexes on the fields that are used for search in ReceiptDB, the retrieval time improved by the order of magnitude power of ten. Being non-relational, MongoDB implements many features of relational databases, such as sorting, secondary indexing and range queries[10]. Indexes are very powerful mechanism in Neo4j as well. Our experiments showed that indexes in Neo4j improved performance of the data retrieval to a greater extent than indexes in MSSQL.

Chapter 7

Summary and Conclusion

In this chapter we sum up the main results of this thesis. The results are primarily based on the research described in Chapter 5 and Chapter 6. We answer the research questions in the following sections.

7.1 NoSQL databases

There are many data storage solutions available today in the market. There is a challenge for application developers and the leadership to make the right choices. Using the wrong database can be costly. It can slow down the process of the development, it can not perform well in the given conditions and it can be difficult to scale. In Section 2.1.2 we describe various NoSQL solutions that provide new tools to tackle today's data storage problems. Most of the mentioned solutions are mature and reliable databases that can be used safely in the enterprise.

In Chapter 5 we compared the database design and data modelling implementation using Neo4j and MongoDB NoSQL databases with implementation in MSSQL. These two particular NoSQL solutions have been chosen for their maturity, good support and their good fit for the problem. As the result, we achieved very light-weight, scalable and well-performing solution at a low cost. In Section 6.1 we discuss the advantages of using NoSQL in this particular industrial study in more detail.

Chapter 6 provides results of performance experiments on Neo4j and MSSQL databases. We populated both applications with the same data and performed series of similar query searches in the database. Experiments showed that performance of Neo4j is not worse and in most cases better than SQL solution, even if the data is not highly connected. We can conclude that Neo4j can be used not only for typical graph problems, but also in larger range of general data storage problems in enterprise.

7.2 Research questions discussion

We have presented three research questions in Section 1.5. The industrial case experience and conducted experiments allowed us to investigate these questions in more detail. Brief discussion of each question is presented below.

What are the advantages of using NoSQL data storage in the enterprise environment?

Tomra AS used Neo4j Graph database and MongoDB document database for their project and successfully delivered solution within designated time and within performance constraints provided by the customers of Tomra. The performance results were ten times better than expected. Creating and maintaining of NoSQL databases was much easier compared to SQL solutions. Developing applications on top of the NoSQL databases was more straightforward with a help of the common APIs and good integration with Java technology that was used during the implementation. Having schema-less structure was beneficial in terms of prototyping, creating and making future amendments in the database. It is a great advantage over SQL schema which is highly difficult to edit after having production data in it.

How does the choice of data storage influence the flow of the application development and architecture of the system?

In this particular project NoSQL databases provided us with flexibility during the design process of the architecture of the system. We were able to choose better suited data storage solution from the data-centric point of view. We could achieve better separation of concerns using two different NoSQL databases. It allowed us to treat both subsystems separately, which simplified the development process, architecture of application and testing. Usage of NoSQL databases helped the developers to focus more on the domain problem, rather than thinking about database architecture. The development process was more light-weight, since we could change the schema of the database without discarding the application layer.

How easy is it to integrate a new solution with the legacy one when using NoSQL databases?

An industrial case showed us that there are no barriers to integrate a NoSQL solution into the legacy system. Simple REST or other entry points allow us to easily communicate between the systems. Integration of the database with the application was possible through multiple language APIs. Both Neo4j and MongoDB provided very rich functionality of communicating with the code and other services.

7.3 SQL or NoSQL?

We do not provide one answer to this question in this thesis. Usage of both is highly depends on the context of the problem. We need to know what are the requirements for the solution? Is the scalability an issue? Do we have to deal with large amounts of unstructured data? These and more questions should be answered before any attempt on choosing the right database solution. SQL databases have been and still are the top choice for many data storage problems. Powerful query language, robust schema and ACID compliance make it the best choice for many industries like finance and government. There is a lot of support, tools and documentation available for developers and companies when deploying SQL solution. It is well-known paradigm that provides users with expected results. However, SQL have been far too long the default choice in many companies. SQL databases have been chosen without consideration because there were arguably no alternatives.

Today the paradigm have shifted. Today decision makers in the ICT industry should not only consider NoSQL solution as well, but consider it first. We believe that use of NoSQL can be beneficial for many enterprises that still use RDBMS. A lot of companies use SQL databases just out of the habit and lack of information about available options. It is apparent that use of NoSQL databases will increase and NoSQL based solutions will become a strong competition to the available SQL solutions. They will also change the way we see, map and program our data. And when that happens, the data professionals across all industries should be aware about all the options and provide well-weighted solutions.

Appendix A

Listings of queries for populating databases with data

Listing A.1: SQL. Creation and populating 'Countries' table

```
USE [TEST]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[Countries](
        [id] [int] IDENTITY(1,1) NOT NULL,
        [name] [varchar](50) NOT NULL
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
GO
INSERT INTO Countries values ('Norway'), ('Sweden'),
        ('Denmark'), ('Germany'), ('Finland'), ('USA'),
        ('Ireland'), ('UK'), ('Iceland'), ('France')
GO
```

Listing A.2: SQL. Creation and populating 'Chains' table

```
USE [Test]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
```

```
GO
CREATE TABLE [dbo].[Chains](
        [id] [int] IDENTITY(1,1) NOT NULL,
        [name] [varchar](50) NOT NULL
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
GO
INSERT INTO Chains values ('Bunnpris'), ('Kiwi'),
        ('Rema1000'), ('Rimi'), ('Coop'), ('ICA'),
        ('Meny'), ('Spar'), ('Netto'), ('7-eleven')
GO
```

Listing A.3: SQL. Creation and populating 'Stores' table

```
USE [Test]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[Stores](
        [id] [int] IDENTITY(1,1) NOT NULL,
        [name] [varchar](50) NOT NULL,
        [address] [varchar](50) NOT NULL,
        [chain_id] [int] NOT NULL,
        [country_id] [int] NOT NULL
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO

declare  @tab table([id] [int] IDENTITY(1,1) NOT NULL,
        addresses  varchar(50))
insert into @tab   values('Ullevål'),('Blindern'),
        ('Centrum'),('Kringså'),('Lysaker'),('Sköyen'),
        ('Fornebu'),('Stortinget'),('Nationalteatre'),
        ('KarlJohan'),('Gamle'),('Grunerlökka'),
        ('Grönland'),('Majorstuen'),('Snaröya'),
        ('Asker'),('Sandvika'),('Drammen'),
        ('Gothenburg'),('Stockholm')

Declare @NumberOfChains int =
        (Select Count(*) From dbo.Chains);
```

```
Declare @NumberOfCountries int =
        (Select Count(*) From dbo.Countries);
Declare @NumberOfAddresses int =
        (Select Count(*) From @tab);
Declare @i int = 1;
Declare @j int = 1;
Declare @k int = 1;
Declare @ChainName varchar(50);
Declare @Address varchar(50);

While(@NumberOfAddresses >= @k)
Begin
        Set @Address =
                (Select addresses From @tab Where id = @k)
        While(@NumberOfChains >= @i)
        Begin
                Set @ChainName =
                        (Select name From dbo.Chains
                        Where id = @i) + '_' + @Address;
                While(@NumberOfCountries >= @j)
                Begin
                        Insert into dbo.Stores values
                                (@ChainName, @Address,
                                @i ,@j)
                        Set @j = @j + 1;
                End
                Set @i = @i + 1;
                Set @j = 1;
        End
        Set @k = @k + 1;
        Set @i = 1;
End
```

Listing A.4: SQL. Creation and populating 'Machines' table

```
USE [Test]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[Machines](
        [id] [bigint] IDENTITY(1,1) NOT NULL,
        [name] [varchar](50) NOT NULL,
        [enabled] [bit] NOT NULL,
        [store_id] [int] NULL
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
ALTER TABLE [dbo].[Machines]
        SET (LOCK_ESCALATION = DISABLE)
GO
--: temp variables
Declare @NumberOfStores int =
        (Select Count(*) From dbo.Stores);
Declare @NumberOfMachines int = 1000;
Declare @i int = 1;
Declare @j int = 1;
Declare @name varchar(50);

While(@NumberOfMachines >= @i)
Begin
        Set @name = 'TMR' +  CONVERT(varchar(50),@i)
        While(@NumberOfStores >= @j)
        Begin
                Insert into dbo.Machines
                        values (@name, 1, @j)
                Set @j = @j + 1;
        End
        Set @j = 1;
    Set @i = @i + 1;
End
```

Listing A.5: Cypher. Creation of 'Country' nodes

```
WITH ['Norway', 'Sweden', 'Denmark', 'Germany',
        'Finland', 'USA', 'Ireland', 'UK',
        'Iceland', 'France']
        as countries
FOREACH (c in countries |
        CREATE (:Country { name:c }) )
```

Listing A.6: Cypher. Creation of 'Chain' nodes

```
WITH ['Bunnpris', 'Kiwi', 'Rema1000', 'Rimi', 'Coop',
        'ICA', 'Meny', 'Spar', 'Netto', '7-eleven']
        as chainnames
FOREACH (c in chainnames |
        CREATE (:Chain { name:c }) )
```

Listing A.7: Cypher. Creation of 'Store' nodes and relationships

```
MATCH (Chains:Chain), (Countries:Country)
WITH Collect (Distinct Chains) as ch,
         Collect (Distinct Countries) as cou,
['Ullevål','Blindern','Centrum','Kringså','Lysaker',
'Sköyen','Fornebu','Stortinget','Nationalteatre',
'KarlJohan','Gamle','Grunerlökka','Grönland',
'Majorstuen','Snaröya','Asker','Sandvika',
'Drammen','Gothenburg','Stockholm'] as addresses

FOREACH(country in cou |
 FOREACH (c in ch |
  FOREACH (a in addresses |
        CREATE (s:Store {name:c.name+"_"+a, address:a})
        CREATE (s-[:BELONGS_TO]->c)
        CREATE (s-[:IN]->country))))
```

Listing A.8: Cypher. Creation of 'Machine' nodes and relationships

```
MATCH (Stores:Store)
WITH Collect(Distinct Stores) as st, "TMR" as name
FOREACH (s in st |
 FOREACH (r in range(1,1000) |
  CREATE (m:Machine {pid:name+"_"+r, enabled:"true"})
  CREATE (m-[:BELONGS_TO]->s) ))
```

Bibliography

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[2] V. Abramova and J. Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 14–22, New York, NY, USA, 2013. ACM.

[3] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[4] E. A. Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[5] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[6] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[7] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. International computer science series. Addison-Wesley, 2005.

[8] Datastax. Nosql in enterprise. White paper, October 2013.

[9] Datastax. Netflix case study. White paper, October 2014.

[10] E. Dede, M. Govindaraju, D. Gunter, R. S. Canon, and L. Ramakrishnan. Performance evaluation of a mongodb and hadoop platform for scientific data analysis. In *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing*, Science Cloud '13, pages 13–20, New York, NY, USA, 2013. ACM.

[11] G. Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, pages 126–130, 2002.

[12] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[13] V. Gudivada, D. Rao, and V. Raghavan. Nosql systems for big data management. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 190–197, June 2014.

[14] J. Han, E. Haihong, G. Le, and J. Du. Survey on nosql database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363–366, Oct 2011.

[15] M. Jackson, S. Crouch, and R. Baxter. Software evaluation: criteria-based assessment. *Software Sustainability Institute, The University of Edinburgh, available at: http://software. ac. uk/sites/default/files/SSI-SoftwareEvaluationCriteria. pdf (accessed 1 October 2012)*, 2011.

[16] B. Kaplan and J. A. Maxwell. Qualitative research methods for evaluating computer information systems. In *Evaluating the Organizational Impact of Healthcare Information Systems*, pages 30–55. Springer, 2005.

[17] I. Konstantinou, E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris. On the elasticity of nosql databases over cloud management platforms. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2385–2388, New York, NY, USA, 2011. ACM.

[18] A. Lith and J. Mattsson. Investigating storage solutions for large data-a comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data. 2010.

[19] I. MongoDB. Forbes overhauls publishing platform: A case study. 2014.

[20] I. MongoDB. Online travel gets personal. White paper, 2014.

[21] M. D. Myers and D. Avison. Qualitative research in information systems. *Management Information Systems Quarterly*, 21:241–242, 1997.

[22] I. Neo Technology. Resource authorization challenge solved with graph database: A case study. 2014.

[23] I. Neo Technology. Snap interactive conquers online dating with neo4j: A case study. 2014.

[24] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. Archware: Architecting evolvable software. In *Software Architecture*, pages 257–271. Springer, 2004.

[25] Z. Parker, S. Poe, and S. V. Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[26] J. Pater and A. Vukotic. Neo4j in action. Manning Publications, 2012.

[27] J. Pokorny. Nosql databases: A step to database scalability in web environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[28] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. " O'Reilly Media, Inc.", 2013.

[29] C. Roe. Acid vs. base: The shifting ph of database transaction processing, 2012.

[30] G. L. Sanders and S. Shin. Denormalization effects on performance of rdbms. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, pages 9–pp. IEEE, 2001.

[31] Tesora. *Data Usage in the Public and Private Cloud*. 2014.

[32] J. Van den Akker. Principles and methods of development research. In *Design approaches and tools in education and training*, pages 1–14. Springer, 1999.

[33] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.