CHALMERS


# Extracting Data from NoSQL Databases

## A Step towards Interactive Visual Analysis of NoSQL Data

*Master of Science Thesis*

PETTER NÄSHOLM

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, January 2012

Extracting Data from NoSQL Databases
A Step towards Interactive Visual Analysis of NoSQL Data

PETTER NÄSHOLM



Examiner: GRAHAM KEMP

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden January 2012

**Abstract**

Businesses and organizations today generate increasing volumes of data. Being able to analyze and visualize this data to find trends that can be used as input when making business decisions is an important factor for competitive advantage. Spotfire is a software platform for doing this. Spotfire uses a tabular data model similar to the relational model used in relational database management systems (RDBMSs), which are commonly used by companies for storing data. Extraction and import of data from RDBMSs to Spotfire is generally a simple task.

In recent years, because of changing application requirements, new types of databases under the general term NoSQL have become popular. NoSQL databases differ from RDBMSs mainly in that they use non-relational data models, lack explicit schemas and scale horizontally. Some of these features cause problems for applications like Spotfire when extracting and importing data.

This thesis investigates how these problems can be solved, thus enabling support for NoSQL databases in Spotfire. The approach and conclusions are valid for any application that interacts with databases in a similar way as Spotfire.

General solutions for supporting NoSQL databases are suggested. Also, two concrete tools for importing data from Cassandra and Neo4j that have been implemented in the Spotfire platform are described. The presented solutions comprise a data model mapping from the NoSQL system to Spotfire tables, sampling and possibly clustering for finding schemas, and an extraction mechanism tailored to the particular system's query interface.

The suggested solutions are not claimed to be complete. Rather, the work in this thesis can serve as a starting point for more thorough investigations or as a basis for something that can be extended.

# Preface

The main stakeholder in this thesis is the company TIBCO Software. The work has been conducted in their office in Göteborg, Sweden between August 2011 and January 2012.

The author would like to thank supervisors Erik Petterson (TIBCO Software) and Graham Kemp (Chalmers) for providing valuable input and guidance throughout the project. Gratitude is also directed towards TIBCO Software for being an excellent host.

# Glossary

| | |
|---|---|
| **ACID** | Atomicity, Consistency, Isolation, Durability |
| **BASE** | Basically Available, Soft-state, Eventual consistency |
| **CD** | Cineasts Dataset |
| **CF** | Column Family |
| **CFS** | Column Family Store |
| **CQL** | Cassandra Query Language |
| **DBMS** | Database Management System |
| **DS** | Document Store |
| **GD** | Graph Database |
| **HDFS** | Hadoop Distributed File System |
| **JAR** | Java Archive |
| **KVS** | Key-Value Store |
| **NoSQL** | Not only SQL |
| **RDBMS** | Relational Database Management System |
| **REST** | Representative State Transfer |
| **SC** | Super Column |
| **SCF** | Super Column Family |
| **SDK** | Software Development Kit |
| **SQL** | Structured Query Language |
| **UUID** | Universally Unique Identifier |

# Contents

# Chapter 1

# Introduction

In this chapter, the main topic of the thesis and the motivation behind it are introduced. This starts off with a presentation of the problem background in Section 1.1. Following this, the problem formulation given in Section 1.2. Then, the main contributions of the thesis are presented in Section 1.3. Finally, an outline of the rest of the thesis is given in Section 1.4.

## 1.1 RDBMSs, NoSQL and Spotfire

Codd presented already in 1970 his relational model for data stores [8]. Ever since, his model has been widely adopted within the IT industry as the default data model for databases. Traditionally, the major database management systems (DBMSs) have been based on this model – resulting in so called relational DBMSs (RDBMSs). Examples of such include MySQL[1], Oracle Database[2] and Microsoft SQL Server[3].

In recent years the requirements for many modern applications have significantly changed – especially since the rise of the Web in the late 90s and early 00s. Large websites must now be able to serve billions of pages every day [9] and web users expect that their data is ubiquitously accessible at the speed of light no matter the time of day. Some of these requirements have made RDBMSs as data stores unsatisfactory in several ways. Issues include that throughput is too low, that they do not scale well and that the relational model simply does not map well to some applications.

As a reaction to this, new types of DBMSs under the umbrella term NoSQL have become popular. The term NoSQL is often interpreted as short for "Not only SQL", where SQL refers to the default data management language in RDBMSs – Structured Query Language. The whole purpose of this movement is to provide alternatives where RDBMSs are a bad fit. The term incorporates a wide range of different systems. In general, NoSQL databases use non-relational data models, lack schema definitions and scale horizontally.

Businesses and organizations today generate increasing volumes of data including information about their customers, suppliers, competitors and opera-

[1] http://www.mysql.com, accessed Jan 25, 2012.
[2] http://www.oracle.com/us/products/database/index.html, accessed Jan 25, 2012.
[3] http://www.microsoft.com/sqlserver/en/us/default.aspx, accessed Jan 25, 2012.

tions. Being able to analyze and visualize this data to find trends and anomalies that subsequently can be acted on to create economic value has become an important factor for competitive advantage – a factor whose importance most likely will increase even more the coming years [23].

Spotfire is a software platform for interactive data analysis and visualization, developed by the company TIBCO Software. Spotfire enables businesses and organizations to understand their data so that they can make informed decisions which in turn create economic value.

While Spotfire includes tools analyzing and visualizing data, the data itself is supplied by the user. The different systems from which one can import data into Spotfire is a key competition point for the product. Today, Spotfire supports importing data from several RDBMSs, spreadsheets and CSV[4]-like formats. However, there is currently no support for any NoSQL alternative.

The Spotfire platform data model is based on the relational model; data is represented by rectangular tables consisting of rows and columns. The relationship to RDBMSs is natural and thus automatic extraction and import from such data sources into Spotfire tables is often a simple task. Because of the non-relational data models and lack of explicitly defined schemas in NoSQL databases, the relationship to Spotfire tables and how to extract data is not always as obvious.

## 1.2 Problem formulation

The thesis aims to investigate how data from NoSQL databases can be extracted and imported into Spotfire tables, so that it can be analyzed and visualized within the application.

Key questions that the thesis attempts to answer include:

- How do different NoSQL data models differ from Spotfire's data model?
- How do these differences affect extraction and import from Spotfire's point of view? How does the lack of explicitly defined schemas affect extraction and import?
- How can these problems be solved – enabling support for NoSQL databases in Spotfire?

To answer these questions, first, a general introduction to NoSQL followed by a survey on NoSQL data models is given. The knowledge presented in these leads to a problem analysis, which ultimately results in suggested solutions. To concretize these, two proof of concept tools that support two particular NoSQL databases are implemented in the Spotfire platform.

While the thesis aims to give an understanding of NoSQL databases in general and how data can be extracted from such into Spotfire, it does not cover underlying implementation details of any NoSQL system. Still, data models, query models and client APIs are thoroughly covered.

[4]Comma-Separated Values. A simple file format where values are separated by commas.

## 1.3 Contribution of thesis

The main contribution of the thesis is its suggestions to TIBCO Software regarding how NoSQL databases can be supported in Spotfire. However, the reasoning and conclusions are valid in a more general sense than that. In fact, developers of any application that interacts with databases in a similar way as Spotfire, that want to support NoSQL alternatives, might benefit from reading the thesis.

Furthermore, Section 3.3 provides a general introduction to NoSQL and contains no Spotfire specifics. This is true for the entire survey in Chapter 4 as well. Thus, these are potentially useful for people in many different organizations and roles; e.g., system architects, developers, database administrators, and perhaps even managers.

## 1.4 Outline of thesis

The rest of the thesis is structured as follows:

- Chapter 2 presents related work. This includes both work on NoSQL in general and work on data analysis on large datasets.
- Chapter 3 gives the theoretical foundation needed for the rest of the thesis. It starts off with a brief recap of concepts and terminology from the relational model. Then, the Spotfire platform is described. After that, a general overview of NoSQL is given. Finally, although it might not at this point be apparent why, some machine learning theory in the form of clustering is given.
- Chapter 4 provides a survey of the four database families within NoSQL. For each family, a general description of characteristics is given, as well as a detailed description of the data model, query model and client APIs for one concrete system.
- Chapter 5 analyzes the problem formulation using the knowledge presented in Chapters 3 and 4, and presents general solutions as a result of this.
- Chapter 6 thoroughly describes the two proof of concept tools that have been implemented to add support for importing data from both Cassandra and Neo4j into Spotfire.
- Chapter 7 provides descriptions and results of the experiments that have been conducted to evaluate the implemented tools.
- Chapter 8 discusses the implemented tools and the overall approach. Also, the points considered most important to investigate further are mentioned.
- Chapter 9 concludes the thesis by considering what has been accomplished and presents possible approaches the company can take.

# Chapter 2

# Related work

As the thesis both provides an introduction to NoSQL in general and deals with how NoSQL systems can be used in a data analysis context, other work on either of these topics are relevant for comparison. Below follows a list of related work and how these differ from the work in this thesis:

- Strauch's paper *NoSQL Databases* [42] provides a thorough introduction to NoSQL. The paper describes the rationales behind the movement, some common techniques and algorithms for solving issues concerning, e.g., consistency and distributed data processing, and also presents a number of concrete systems with implementation details and data models. As the paper is intended to give an overview, no particular focus is put on data models. For example, there are no concrete examples on how the data models can be used or what they imply for, e.g., application development. Also, Graph Databases (see Section 3.3.3) are not covered at all.

- The thesis *No Relation: The Mixed Blessings of Non Relational Databases* [48], written by Varley, gives a good overview over non-relational data models and how they differ from the relational model. Throughout the thesis, Varley tries to determine a winner between the two model paradigms from a data modeling perspective by considering various strengths and weaknesses and comparing them side by side. As the thesis is more than two years old, some parts are already outdated. Like in Strauch's paper, Graph Databases (see Section 3.3.3) are not covered at all.

- Several languages have recently been developed for processing and analysis of large datasets, including Dremel [32], Hive [45], Jaql [4], Pig [36] and Tenzing [7]. Typically, the datasets are spread over several machines in a cluster and are stored either directly in a distributed file system or in some database. The programs are executed in the cluster in a MapReduce[1] fashion. For example, Jaql and Pig programs are translated into Hadoop[2] jobs, and they both read data from HDFS[3]. These languages are mainly targeted to power users with programming experience. Some of them

[1] A software framework developed by Google for distributed data processing on large datasets in clusters [11].

[2] An open source implementation of the MapReduce framework supported by Apache.

[3] Hadoop Distributed File System. The primary storage system for Hadoop applications.

can also be used by less experienced users, e.g., Tenzing, which supports a mostly complete SQL implementation. The approach used in all of these languages requires that there is a cluster that can be used for data processing tasks, and that the structure of the data is known in advance.

- A Spotfire competitor, Tableau[4], recently announced their added support for working with data stored in HDFS [16]. Their approach is to let the user input Hive scripts that subsequently are run on a Hadoop cluster. The output is then imported into Tableau's in-memory engine. The user defines which data from the file system should be processed, and how complex data should be flattened to a format that the application supports. Using a cluster of several machines for these computations enables Tableau to get decent response times with large datasets even though no clever query mechanism is available. Obviously, this approach is targeted at rather advanced users, and it requires analysts to know the structure in which data is stored on beforehand.

[4]http://www.tableausoftware.com, accessed Jan 25, 2012.

# Chapter 3

# Theory

In this chapter the theoretical foundation needed for the rest of the thesis is presented. First, a brief recap of concepts and terminology from the relational model is given in Section 3.1. Then, the Spotfire platform is described in Section 3.2. After that, Section 3.3 gives a general overview of NoSQL. Finally, Section 3.4 presents some machine learning theory in the form of clustering.

## 3.1 Recap of the relational model and ACID

The reader is assumed to be familiar with the relational model and ACID, which is why this section contains only a brief recap of important terminology and concepts.

In the relational model, data is represented as rectangular tables known as *relations*. A relation has a set of named *attributes*. These can be thought of as names of the table columns. Each attribute is associated with a *domain*, i.e., an atomic type such as an integer or a string. A row in a table is called a *tuple*. When a relation has $N$ attributes, each tuple contains $N$ *components* – one for each attribute. Every component must belong to the domain of its corresponding attribute. A *schema* is the name of a relation together with its set of attributes.

Most RDBMSs guarantee so called ACID transactions. ACID is an acronym of four properties. These are, in order [47]:

- Atomicity: Transactions are atomic. That is, a transaction is executed either in its entirety or not at all.
- Consistency: Every transaction takes the database from one valid state to another.
- Isolation: A running transaction will not interfere with another transaction.
- Durability: The effect of a transaction must persist and thus never be lost.

## 3.2 Spotfire

Spotfire is a software platform for interactive data analysis and visualization. It is developed by TIBCO Software[1], a software company based in Palo Alto, CA, US. The first version of Spotfire was launched in 1996.

Spotfire enables businesses and organizations to understand their data so that they can make informed decisions which in turn create economic value. An example of what a business typically wants to analyze is sales data. The business might have stores at several locations and sell a range of different products. Using Spotfire, they can analyze, e.g., how sales differ between the different locations, which products are popular, whether there are some trends in the customers' behaviors, and so on. Findings like these can later be used to make decisions, e.g., when allocating budgets among the locations or when deciding which products to put on the shelves, to maximize the success of the business.

In Spotfire, the basic unit the user works with is the *analysis* file. An analysis file is split into a number of *pages*. Each page contains a number of *visualizations*. A visualization is a visual representation of data in some way. Although not an exhaustive list, common visualization types in Spotfire include the Bar Chart, Line Chart, Combination Chart, Pie Chart, Scatter Plot, Heat Map and Box Plot. An example of an analysis is shown in Figure 3.1.

Before any visualization can be created, data must be imported from external sources into *data tables*. For every analysis file, there is a set of these. A data table consists of *columns* and *rows*. A column has a name and a type. The supported types are: `Blob`, `Boolean`, `Integer`, `LongInteger`, `SingleReal`, `Real`, `Currency`, `Date`, `Time`, `DateTime`, `TimeSpan` and `String`. In a table with $N$ columns, a row is an $N$-tuple where each value belongs to exactly one column and is an instance of that column's type. An example of a Spotfire data table is shown in Figure 3.2.

## 3.3 NoSQL

The term NoSQL dates back to 1998 when it was used for a particular RDBMS that did not support SQL[2]. It was not until 2009 that it was used with approximately the same meaning that it has today [42]. Still, it is not rigorously defined in the literature. As for the word, it is often interpreted as short for "Not only SQL" rather than "No SQL", emphasizing that the movement is about coming up with alternatives for RDBMSs/SQL where these are a bad fit, as opposed to being completely against them [14].

The two seminal papers on Google's Bigtable [6] and Amazon's Dynamo [12] seem to have been the starting point for the NoSQL movement. Many of the design decisions and principles used in these can be found in later works. In general, the pioneers of the movement have been big web companies like the two aforementioned.

NoSQL incorporates a wide range of different systems. Often, these have been created to solve one particular problem for which RDBMSs have not

[1] http://spotfire.tibco.com, accessed Jan 25, 2012.

[2] http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql, accessed Jan 25, 2012.

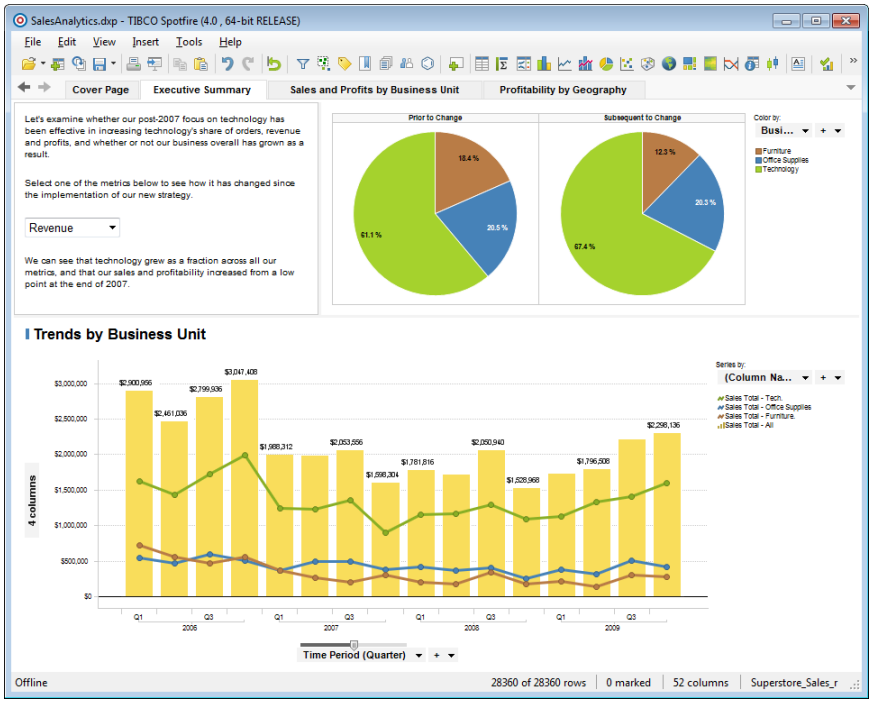


Figure 3.1: *Example of a Spotfire analysis. The current page of the analysis shows a text box and two visualizations; a Pie Chart (containing two separate pies) and a Combination Chart (combining lines and bars).*

been appropriate, for various reasons. A typical NoSQL database uses a non-relational data model, lacks schema definitions and scales horizontally (see Section 3.3.1).

Now that NoSQL is generally introduced, a more thorough background and descriptions will be given in following subsections. First, in Section 3.3.1, the main motives behind the NoSQL movement are given. Then, Section 3.3.2 describes the main characteristics of NoSQL databases. Section 3.3.3 presents a taxonomy of NoSQL databases. Section 3.3.4 gives some dimensions that should be taken into consideration when choosing between using an RDBMS and a NoSQL database. Finally, Section 3.3.5 describes how application development is affected when using a NoSQL system instead of an RDBMS.

### 3.3.1 Motives

While RDBMSs have been the default solution for data management in applications for many years, some modern applications have experienced changing requirements for which RDBMSs have been unsatisfactory in several ways [42]. Below follows a list of some of the most important of these. Naturally, these also form the main motives behind the NoSQL movement:

| Record No | Order Priority | Sales Total | Discount | Ship Method |
|---|---|---|---|---|
| 7081 | 2-HIGH | 227.60 | 0.00 | REGULAR... |
| 7082 | 3-MEDIUM | 99.81 | 0.12 | REGULAR... |
| 7083 | 3-MEDIUM | 51.62 | 0.10 | REGULAR... |
| 7084 | 3-MEDIUM | 32.98 | 0.02 | REGULAR... |
| 7085 | 5-LOW | 83.75 | 0.09 | REGULAR... |
| 7086 | 1-URGENT | 13.02 | 0.07 | REGULAR... |
| 7087 | 1-URGENT | 283.09 | 0.01 | REGULAR... |
| 7088 | 4-NOT SPE... | 7400.85 | 0.05 | REGULAR... |

Figure 3.2: *Example of a Spotfire data table with columns Record No, Order Priority, Sales Total, Discount and Ship Method.*

- Throughput is too low: Some web applications require the underlying data store to process larger volumes than most RDBMSs would be able to handle.
- They were not built to scale horizontally: While *vertical* scaling refers to adding more hardware onto existing machines, *horizontal* scaling refers to adding more machines to a cluster. RDBMSs were originally built to scale vertically [42]. However, vertical scaling has its limitations in that it gives sublinear effects, sometimes requires high-risk operational efforts and that there is an upper bound on how much hardware that can be added and efficiently utilized by the software. Horizontal scaling in general does not have these problems. Today, some RDBMSs offer horizontal scalability, but utilizing this is generally not straightforward.
- Object-Relational mapping is expensive: Today, object-oriented programming is the most prevalent programming paradigm. Persisting application objects in an RDBMS requires a mapping between the application's object model and the database's relational model. Setting this up takes time and effort, and it enforces developers to treat data relationally – even though it might not inherently be.
- The "One Size Fits All"-thinking is flawed: RDBMS development has been characterized by a "One Size Fits All"-thinking, by which one means that RDBMSs have been seen as a general tool that can handle all different requirements applications might have on data management. Being a general tool – trying to be too much at once – some argue that RDBMSs end up excelling at nothing [41]. Since different applications might have different requirements on consistency, performance and so on, this thinking is in some sense inherently flawed.
- ACID is not always needed: Most RDBMSs provide ACID transactions (see Section 3.1). Naturally, when providing these, tradeoffs on other things such as performance must be done. In some applications ACID is not needed, which means some performance gets lost for nothing.

### 3.3.2 Characteristics

Most NoSQL databases share a common set of characteristics [42]. Naturally, since NoSQL is a broad concept, more or less all of these characteristics have exceptions. Still, they can serve to give a general idea about what NoSQL databases are:

- Distributed: NoSQL databases are often distributed systems where several machines cooperate in clusters to provide clients with data. Each piece of data is commonly replicated over several machines for redundancy and high availability.
- Horizontal scalability: Nodes can often be dynamically added to (or removed from) a cluster without any downtime, giving linear effects on storage and overall processing capacities. Usually, there is no (realistic) upper bound on the number of machines that can be added.
- Built for large volumes: Many NoSQL systems were built to be able to store and process enormous amounts of data quickly.
- BASE instead of ACID: Brewer's CAP theorem [5] states that a distributed system can have at most two of the three properties Consistency[3], Availability[4] and Partition tolerance[5]. For a growing number of applications, having the last two are most important. Building a database with these while providing ACID properties is difficult, which is why Consistency and Isolation (see Section 3.1) often are forfeited, resulting in the so called BASE approach [5]. BASE stands for Basically Available, Soft-state, Eventual consistency, with which one means that the application is available basically all the time, is not always consistent, but will eventually be in some known state.
- Non-relational data models: Data models vary, but generally, they are not relational. Usually, they allow for more complex structures and are not as rigid as the relational model.
- No schema definitions: The structure of data is generally not defined through explicit schemas that the database knows of. Instead, clients store data as they desire, without having to adhere to some predefined structure.
- SQL is unsupported: While most RDBMSs support some dialect of SQL, NoSQL variants generally do not. Instead, each individual system has its own query interface. Recently, the Unstructured Data Query Language[6] was proposed as an attempt to unify the query interfaces of NoSQL databases [25].

[3] I.e., that all nodes always agree on the current state.
[4] I.e., that every request gets a response about its success.
[5] I.e., that the system continues to work even though arbitrary messages are lost.
[6] http://www.unqlspec.org, accessed Jan 25, 2012.

### 3.3.3 Taxonomy

There is no universally accepted taxonomy of NoSQL databases. Sources differ in which systems they include and into which categories the systems are classified. Often, however, suggested taxonomies divide systems, at least partly, based on their data models [42]. Since this thesis mainly focuses on properties of data models and client interfaces of these databases, the following data model-based taxonomy of four families is suggested for the purposes here:

- Key-Value Stores
- Document Stores
- Column Family Stores
- Graph Databases

This taxonomy is conceptually consistent with, e.g., Scofield's [39], with some minor naming differences. There has been discussion around the naming of the third family [2]; simply calling it "Column Stores" can be misleading since there are relational column stores too, e.g., Sybase IQ[7] and Vertica[8]. Therefore, the more explicit name "Column Family Stores" is here chosen.

Note that neither object databases nor XML[9] databases are included in any of the families. The reason for this is that these are generally not referred to as being part of the NoSQL movement [42].

Descriptions of the characteristics for each of the aforementioned families, as well as examples of systems that are included in them, are given in the survey in Chapter 4.

### 3.3.4 RDBMSs vs. NoSQL – when to choose what

A general misconception about the NoSQL movement is that it aims to replace RDBMSs. This is not the case. The purpose is rather to provide alternatives in situations where RDBMSs are not a perfect fit. Thus, NoSQL databases and RDBMSs generally satisfy different needs, and are thus not always comparable. When choosing whether to use an RDBMS or a NoSQL alternative for a particular application, at least the following dimensions should be considered:

- Business needs
  - Maturity: RDBMSs are more mature and are thus more proven and more likely to contain fewer bugs.
  - Support: RDBMSs are often professionally supported, whereas NoSQL databases often are open source projects supported by volunteers.
  - Costs: As open source projects, several NoSQL databases can be used for free. Many RDBMS alternatives are in comparison rather expensive. Also, because of less rigid data models and no explicitly defined schemas, application development is often faster using NoSQL databases, resulting in lower development costs.

[7] http://www.sybase.ca/products/datawarehousing/sybaseiq, accessed Jan 25, 2012.
[8] http://www.vertica.com, accessed Jan 25, 2012.
[9] Extensible Markup Language. A markup language often used to store arbitrary data structures.

- Engineering needs
  - Performance: RDBMSs provide sufficient performance for many applications. Whether this is sufficient or not generally has to be determined by evaluating possible alternatives for the particular use case.
  - Consistency: Several NoSQL databases do not offer the ACID properties that RDBMSs generally do. Not all applications need these, and can work sufficiently well with looser guarantees.

When choosing between relational alternatives, there are standardized benchmarks for which these can be compared. A common benchmark for this purpose is TPC-H[10]. Since RDBMSs often are intended to solve similar tasks, benchmarks like this are easy to consult. However, since NoSQL databases often are built to solve one specific problem, benchmarks for these are not as easy to construct. Recently, though, the YCSB benchmark for NoSQL databases was suggested [10].

### 3.3.5 Implications for application development

NoSQL databases generally lack explicitly defined schemas (see Section 3.3.2). This does not, however, imply that applications using NoSQL databases lack schemas and that the stored data lacks structure. Instead, what happens is that the schema moves down to the application level; the schema in a NoSQL database is defined by how applications use it. This has several implications. One is that there is no built-in mechanism enforcing the schema, which potentially could lead to the database being in an invalid state if there is an error in the application[11]. Another implication is that inspecting the database without knowing the schema is much harder since the structure and meaning of the data in general is unknown.

Instead of ACID, some NoSQL databases provide BASE (see Section 3.3.2). Naturally, not all applications can cope with an underlying data store that provides BASE rather than ACID properties. For example, in an online banking application, a user that transfers an amount of money to some account and then finds that the original account's balance is untouched might end up issuing the same transfer multiple times and eventually getting the news that the account has been emptied. Obviously, online banking applications can in general not cope with BASE properties. On the other hand, in applications like Facebook, the consequences of reading an old value are not as severe, and therefore such applications can possibly cope with BASE.

## 3.4 Clustering

Section 5.1 suggests using clustering as part of a general solution. To give a better understanding of that discussion, clustering and methods for solving clustering are here formally introduced. Section 3.4.1 discusses methods for automatically estimating the optimal number of clusters.

[10] http://www.tpc.org/tpch, accessed Jan 25, 2012.

[11] To alleviate these problems, some have built custom tools that check the state of the data store against an external schema on a regular basis [22].

Clustering is the problem of how to assign a set of objects into groups, called clusters, so that objects within the same cluster, according to some measure, are more similar to each other than to objects in other clusters. It is a form of the machine learning problem unsupervised learning, by which one means that the learning (here, how to form the clusters) happens without any labeled training data.

A certain type of clustering, called hard clustering (as opposed to, e.g., soft or hierarchical clustering), refers to the task of assigning each given object to exactly one cluster [31]. Given a method to measure the similarity, or rather, dissimilarity or distance, between objects, this task can be formalized as follows:

> Given a set of objects $X$, the number of clusters $K \in \mathbb{Z}^+$ and a distance function $d \in \mathbb{R}_0^+$ between all pairs of objects in $X$, partition $X$ into $K$ disjoint sets $X_1, X_2, ..., X_K$ such that $\sum_k \sum_{x,x' \in X_k} d(x, x')$ is minimized[12].

One way to express the resulting partitioning is by using a cluster assignment function $C : X \to \{1, 2, ..., K\}$ (see Algorithm 3.1).

This problem is since long known to be NP-hard [17]. With $N = |X|$, the number of distinct cluster assignments possible is [26]:

$$S(N, K) = \frac{1}{K!} \sum_{k=1}^{K} (-1)^{K-k} \binom{K}{k} k^N \tag{3.1}$$

Already $S(19, 4) \approx 10^{10}$, and of course, it is desirable to be able to cluster more than 19 objects into four clusters. Thus, exhaustive search for the optimal solution is in general infeasible. Therefore, suboptimal algorithms must be employed in practice. A common choice for this is the K-means algorithm [20].

K-means is intended for situations where the objects' variables are quantitative and the distance function can be chosen as the squared Euclidean distance. However, this is not always the case for a particular domain. As an alternative in such situations, the K-medoids algorithm has been proposed [20]. This algorithm is suitable when object variables are not quantitative, but rather qualitative attributes.

K-medoids was suggested by Kaufman and Rousseeuw by the name Partitioning Around Medoids [27]. A *medoid* is the center of a cluster, defined as the object in the cluster that minimizes the total distance to all other objects in the same cluster. Algorithm 3.1 shows how to compute K-medoids in detail.

As for time complexity, since the total number of iterations is not known in advance, a general expression cannot be derived; but complexities for individual steps can. Solving Equation 3.2 includes checking the distance from $x$ to each cluster center $m_k$. Since there are $K$ cluster centers and $N$ objects, this step takes $O(KN)$ in total. As for Equation 3.3, for every cluster $X_k = \{x \in X : C(x) = k\}$ this takes $O(|X_k|^2)$ since for every object the distance to all other objects in the same cluster must be summed.

As can be seen in Algorithm 3.1, how to perform step 1 is not explicitly defined. Common suggestions include assigning seeds uniformly at random or

[12] $\frac{1}{2} \sum_k \sum_{x,x' \in X_k} d(x, x')$ is known as the *intra-cluster dissimilarity* (the constant is irrelevant when optimizing). Alternatively, and equivalently, the objective function can be expressed as maximizing the *inter-cluster dissimilarity*, $\frac{1}{2} \sum_k \sum_{x,x' \in X : C(x) \neq C(x')} d(x, x')$.

**Algorithm 3.1** K-medoids

**Input**: A set of objects $X$, the number of clusters $K \in \mathbb{Z}^+$ and a distance function $d \in \mathbb{R}_0^+$ between all pairs of objects in $X$.
**Output**: A cluster assignment $C : X \to \{1, 2, ..., K\}$.

1. Initialize $K$ seed cluster centers.
2. Given current cluster centers $\{m_1, m_2, ..., m_K\}$, assign each object $x \in X$ to the cluster with the closest center, i.e.:

$$C(x) = \underset{k}{\operatorname{argmin}}\, d(x, m_k) \tag{3.2}$$

3. For every cluster $X_k = \{x \in X : C(x) = k\}$, assign as cluster center $m_k$ the object $x \in X_k$ in the cluster that minimizes the total distance to the other objects in that cluster, i.e.:

$$m_k = \underset{x:C(x)=k}{\operatorname{argmin}} \sum_{C(x')=k} d(x, x') \tag{3.3}$$

4. Repeat steps 2 and 3 until no assignments change.

using results from other clustering algorithms [31]. In 2007, the K-means++ method was suggested by Arthur and Vassilvitskii [3]. They found that for K-means, initializing seeds by the K-means++ method outperformed choosing seeds uniformly at random in both speed and the accuracy of the resulting clustering. Naturally, K-means++ seeding is not specific for K-means, but can also be applied to, e.g., K-medoids. The K-means++ seeding is described in Algorithm 3.2.

As shown there, after having chosen the initial seed, seeds are picked with a probability described by Equation 3.4. Since the denominator is constant for a given value of $S$, this equation says that each object is picked with a probability proportional to the square of its closest distance to an already picked seed. Thus, the greater the distance of an object to the already chosen seeds, the more likely it is to be chosen. A straightforward K-means++ seeding implementation requires $O(KN)$ time[13].

An alternative to both uniformly randomized and K-means++ seeding is an algorithm that greedily chooses the next seed to be the object that maximizes the total distance to the already chosen seeds, after having chosen the first seed uniformly at random. This method is here referred to as the Greedy seeding method and it is described in Algorithm 3.3. A straightforward implementation requires $O(KN)$ time. Naturally, this method is not robust against outliers and

[13]To see why, consider how step 2 would be implemented. For all $x' \in X$, $d_{min}(x', S)^2$ can be computed in $O(N)$ in total by storing these values between iterations and updating them for each incremental change in $S$. Given these distances, selecting an object according to prescribed probabilitity can then be implemented in $O(N)$ by uniformly randomizing a number in the range $[0.0, \sum_{x \in X} d_{min}(x, S)^2)$ and running through the stored distances to see which object this value corresponds to. Since step 2 is run $K$ times, this gives a total running time of $O(KN)$.

**Algorithm 3.2** K-means++ seeding

**Input**: A set of objects $X$, the number of clusters $K \in \mathbb{Z}^+$ and a distance function $d \in \mathbb{R}_0^+$ between all pairs of objects in $X$.
**Output**: A set of cluster center seeds $S = \{s_1, s_2, ..., s_K\} \subseteq X$.

1. Choose an initial seed $s_1$ uniformly at random from $X$.
2. Choose the next seed $s_i$, selecting $s_i = x' \in X$ with probability

$$P(x') = \frac{d_{min}(x', S)^2}{\sum_{x \in X} d_{min}(x, S)^2} \tag{3.4}$$

   where

$$d_{min}(x, S) = \min_{s \in S} d(x, s). \tag{3.5}$$

3. Repeat step 2 until $K$ seeds have been chosen.

it is also flawed by the greedy assumption. However, as shall be seen in Section 7.2, it can be very effective.

**Algorithm 3.3** Greedy seeding

**Input**: A set of objects $X$, the number of clusters $K \in \mathbb{Z}^+$ and a distance function $d \in \mathbb{R}_0^+$ between all pairs of objects in $X$.
**Output**: A set of cluster center seeds $S = \{s_1, s_2, ..., s_K\} \subseteq X$.

1. Choose an initial seed $s_1$ uniformly at random from $X$.
2. Choose the next seed $s_i$ to the object which maximizes the total distance to the already chosen seeds, i.e.:

$$s_i = \operatorname*{argmax}_{x \in X \setminus S} \sum_{s \in S} d(x, s) \tag{3.6}$$

3. Repeat step 2 until $K$ seeds have been chosen.

### 3.4.1 Estimating the optimal number of clusters

As can be seen in the discussion above, the number of clusters $K$ is considered an input. In some applications, though, the number of clusters is not always known – it might be the case that there is just a set of objects that potentially could be clustered, and one wants to know what the "optimal" number of clusters is and what the corresponding clustering in that case would be. Of course, it is possible to run the clustering algorithm for different values of $K$ and then manually evaluate the resulting clusterings and pick the best one. But this requires human effort, for a task that potentially is quite tedious. Also, for various reasons, it might not be the case that human evaluation always gives the best results.

Finding the optimal number of clusters is an inherently hard problem; how to define a cluster involves subjectivity and depends on the end goal. Thus, any method is expected to have its flaws and certain cases for which it does not perform well. However, a generally well-performing method can still reduce the amount of human effort required to achieve desired results.

Several methods for automatically estimating an optimal $K$ have been proposed. Among others, these include an information theoretic method [43], the Silhouette method [27], the Gap statistic [46] and Clest [13]. Some of these, e.g., the Silhouette method, do not have the ability to estimate $K = 1$ cluster (only $K \geq 2$).

While Dudoit and Fridlyand concluded in a study that Clest in general is more accurate and robust than several other methods, they also reported good performance of the Gap statistic [13]. In fact, the Gap statistic estimated $K$ more accurately than Clest for four models out of the eight they tried. Furthermore, the Gap statistic has fewer parameters and is thus easier to use in a more general setting.

In general, given a clustering algorithm, the *within-cluster dispersion*[14] $W_K = \sum_k \frac{1}{2|X_k|} \sum_{x,x' \in X_k} d(x, x')$ of a clustering decreases with increasing $K$. Based on this, the overall idea of the Gap statistic is to compute an expected curve of $W_K$ by clustering samples from a reference distribution for $K = 1, 2, ..., K_{\max}$ and compare this to the $W_K$ curve resulting from clustering the observed data. The optimal number of clusters is then the $K$ for which the latter curve falls the farthest below the former. The algorithm for computing the Gap statistic is shown in Algorithm 3.4. For more details of this method, consult Tibshirani et al.'s original paper [46].

As can be seen in Algorithm 3.4, the time complexity of the Gap statistic is dominated by clustering computations; step 2 in particular. There, $KB$ clusterings are computed. Thus, using a clustering algorithm that runs in $O(C)$ time, the Gap statistic can be computed in $O(KBC)$ time.

[14] This is the sum of, over all clusters, the average distances within each cluster.

**Algorithm 3.4** Gap statistic

**Input**: A set of objects $X$, the maximum number of clusters $K_{\max} \in \mathbb{Z}^+$, a distance function $d \in \mathbb{R}_0^+$ between all pairs of objects in $X$, a reference distribution $R$ and the number of reference data sets $B$.

**Output**: An estimation of the optimal number of clusters $K^*$.

1. Cluster $X$ for $K = 1, 2, ..., K_{\max}$, giving a series of $W_K$s.
2. Generate $B$ reference data sets from $R$ and cluster these for $K = 1, 2, ..., K_{\max}$, giving a series of $W^*_{Kb}$ where $b = 1, 2, ..., B$.
3. Compute the estimated Gap statistic:

$$\text{gap}(K) = \frac{\sum_b \log(W^*_{Kb})}{B} - \log(W_K) \tag{3.7}$$

4. Let $\bar{l} = (1/B) \sum_b \log(W^*_{Kb})$ and compute standard deviations:

$$\sigma_K = \sqrt{\frac{\sum_b (\log(W^*_{Kb}) - \bar{l})^2}{B}} \tag{3.8}$$

5. Let $s_K = \sigma_K \sqrt{1 + 1/B}$ and choose the optimal number of clusters:

$$K^* = \text{smallest } K \text{ such that } \text{gap}(K) \geq \text{gap}(K+1) - s_{K+1} \tag{3.9}$$

# Chapter 4

# Survey

In this chapter, a survey of the data models of the four NoSQL families is presented. Sections 4.1 – 4.4 present Key-Value Stores, Document Stores, Column Family Stores and Graph Databases, respectively. For each family, a general description of the general data model within the family is given. To make the discussion more concrete, one system within each family is presented with an overview, its data model, its metadata, its indexing capabilities and its client APIs.

Naturally, since NoSQL databases evolve very rapidly, any comprehensive survey would quickly get outdated. Therefore, it should be noted that this chapter is not intended to serve as a complete guide in any way, but rather an introduction to the families to get a better understanding of them. The choices of the four concrete systems that are described are all motivated by them being among the most popular system within their respective family.

## 4.1 Key-Value Stores

Being one of the NoSQL pioneers in general, Amazon's Dynamo was also the starting point for Key-Value Stores (KVSs). The architecture and data model described in the original Dynamo paper [12] has inspired many of the popular KVSs seen today.

KVSs have the simplest of data models among NoSQL families. The data model is based on the abstract data type Map [19]. Thus, a KVS contains a collection of key-value pairs where all keys are unique. Each pair can be queried by key via a `get` operation. New pairs are added via a `put` operation. What mainly differs between different KVSs' models are the permitted types of the keys and the values.

It can be discussed to which extent the key-value pair model is enough for building applications on top of. For example, Bigtable creators Chang et al. argue that it is too limiting for developers [6], which was why they settled on a more complex model. On the other hand, Dynamo creators DeCandia et al. claim that several services on the Amazon platform require only primary-key access, meaning that KVSs suffice for these [12]. Most likely, however, is that more complex applications call for more complex data models and interfaces than what a KVS can offer.

Dynamo aside, examples of common KVSs include Redis (see Section 4.1.1), Riak[1], Kyoto Cabinet[2], Scalaris[3], LevelDB[4] and MemcacheDB[5]. Section 4.1.1 describes Redis in more detail.

### 4.1.1 Redis

#### Overview

Redis[6] is an open source KVS officially sponsored by VMware. It was initially released in 2009 and is available under the BSD license[7]. Redis has several prominent users, including GitHub [37], Stack Overflow [34] and Flickr [49].

Redis stores all data in-memory, which means it is very fast and that the amount of data stored in a node is limited by its RAM.

#### Data model

Keys in Redis are uninterpreted bit sequences. As for values, there is support for five different types: plain bit sequences and lists, sets, hashes and sorted sets of bit sequences. Lists are ordered bags. Sets are like mathematical sets. Hashes are maps. Sorted sets are like sets with the addition that every element has a floating point number associated with it that determines its rank.

A simple customer database containing two customers might be stored as follows in Redis:

```
(customerIds,            [12, 67])
(customerId:12:name,     Smith)
(customerId:12:location, Seattle)
(customerId:67:name,     Andersson)
(customerId:67:location, Stockholm)
(customerId:67:clubCard, 82029)
```

This is a common way of storing complex objects in KVSs. Redis' data model, however, is a bit more powerful than the average KVS. The same database could be stored using hashes, instead of atomic values, as follows:

```
(customerIds,   [12, 67])
(customerId:12, {name: Smith, location: Seattle})
(customerId:67, {name: Andersson, location: Stockholm,
  clubCard: 82029})
```

Of course, in both of these examples, both keys and values are really bit sequences. For presentation purposes above, these have been written as strings and integers.

[1] http://wiki.basho.com/Riak.html, accessed Jan 25, 2012.
[2] http://fallabs.com/kyotocabinet, accessed Jan 25, 2012.
[3] http://code.google.com/p/scalaris, accessed Jan 25, 2012.
[4] http://code.google.com/p/leveldb, accessed Jan 25, 2012.
[5] http://memcachedb.org, accessed Jan 25, 2012.
[6] http://redis.io, accessed Jan 25, 2012.
[7] http://www.xfree86.org/3.3.6/COPYRIGHT2.html#5, accessed Jan 25, 2012.

#### Metadata

The namespace in Redis is flat and there is no schema support. Therefore, no particular metadata is stored.

#### Indexing

Naturally, all keys are indexed by default, giving $O(1)$ lookups. Values cannot be indexed per se. Instead, to achieve constant lookups on values, custom indexes can be built by adding $K$ to a list at key $V$ for every pair $(K, V)$ that is added.

#### Interface

There are Redis clients for over 20 programming languages that have been created by third party developers [38]. The functionality exposed by different clients varies. Examples of methods in the core API include:

- `SET key value`, which adds the pair $(key, value)$ to the database.
- `GET key`, which returns the value stored at key $key$.
- `LPOP key`, which pops and returns the leftmost value in the list stored at key $key$.
- `SCARD key`, which returns the cardinality of the set stored at key $key$.
- `HSET key field value`, which sets the hash field $field$ stored at key $key$ to $value$.

Redis can be queried for its set of keys matching a certain pattern with the command `KEYS pattern`. For example, `KEYS *` returns all keys in the database, and `KEYS h?llo` returns all keys that start with "h", end with "llo" and have an arbitrary character in between.

## 4.2 Document Stores

From a data model perspective, Document Stores (DSs) are the next logical step from KVSs. The central notion in DSs is the *document*. A document is generally a set of fields, where a field is a key-value pair. Keys are atomic strings or bit sequences, and values are either atomic, e.g., integers or strings, or complex, e.g., lists, maps, and so on. A DS can store several documents or even several collections of documents.

Examples of common DSs include MongoDB (see Section 4.2.1), CouchDB[8], Jackrabbit[9], RavenDB[10] and Terrastore[11]. Section 4.2.1 describes MongoDB in more detail.

[8] http://couchdb.apache.org, accessed Jan 25, 2012.
[9] http://jackrabbit.apache.org, accessed Jan 25, 2012.
[10] http://ravendb.net, accessed Jan 25, 2012.
[11] http://code.google.com/p/terrastore, accessed Jan 25, 2012.

### 4.2.1 MongoDB

MongoDB[12] is an open source DS that was initially released in 2009. It is developed and professionally supported by the company 10gen. Prominent users include Disney [18], craigslist [51] and foursquare [22]. The source is available under AGPLv3[13].

#### Data model

A MongoDB deployment holds a set of *databases*. Each database stores a set of *collections*. A collection holds a set of documents. A document, as described in the introduction, is a set of *fields*, where a field is a key-value pair. In MongoDB, documents are represented in a format known as BSON[14]. Thus, MongoDB supports all types that BSON supports[15]. Keys are always strings. Value types include, but are not limited to: string, integer, boolean, double, null, array, object, date, object id, binary data, regular expression and code. All documents contain an automatically added field called `_id` which assigns the document a unique id.

An example MongoDB document storing information about a customer looks as follows:

```
{
  _id:      4e77bb3b8a3e000000004f7b,
  name:     Müller,
  location: Zürich,
  clubCard: 48551,
  points:   7946
}
```

As can be seen, all values are atomic in the above example. Another example where some values are complex is the following person record:

```
{
  _id:       4b866f08234ae01d21d89605,
  name:      Smith,
  birthYear: 1947,
  address:   {
    city:    San Francisco,
    street:  Mason St.
  },
  hobbies:  [sailing, cooking, chess]
}
```

Notice how the field `address` stores a document within the document. This is called *embedding* in MongoDB. As an alternative to embedding, fields can also *link* to other documents by storing their `_id` values.

[12] http://www.mongodb.org, accessed Jan 25, 2012.
[13] http://www.gnu.org/licenses/agpl-3.0.html, accessed Jan 25, 2012.
[14] Short for Binary JSON; a serialization of JSON-like documents.
http://bsonspec.org, accessed Jan 25, 2012.
[15] Notice, though, that different drivers might implement these types differently.

#### Metadata

While there are no schemas in MongoDB, the names of the databases and the names of their respective collections are stored as metadata. Also, the defined indexes (see **Indexing** below) are stored as metadata.

#### Indexing

Indexes are supported in MongoDB. They work similarly as to how indexes work in RDBMSs, allowing clients to index arbitrary fields within a collection. Even embedded fields can be indexed, e.g., `address.city` from the second example in **Data model** above. Fields that have been indexed support both random access retrieval and range queries. Indexes can be created at any time in the lifetime of a collection. The `_id` field of all documents is automatically indexed.

#### Interface

The original MongoDB creators provide drivers for over twelve programming languages, such as C, C++, Java, PHP, Ruby, Python and .NET. In addition, there are even more drivers that have been developed by the community. Not all drivers completely support the core MongoDB API and they also differ in how support for certain types is implemented. There is no separate query language for MongoDB.

The standard MongoDB distribution also includes an interactive shell called `mongo` through which commands can be issued to an underlying MongoDB instance.

## 4.3 Column Family Stores

The data models in Column Family Stores (CFSs) are inspired by Google's Bigtable [6]. The general theme is a structure called the *column family* (CF). In the original Bigtable paper, a CF is a collection of columns. These exist in a table. Rows are similar to relational rows, except that all rows in the same table do not necessarily have the same structure. A value in a cell, i.e., the intersection of a row and a column, is an uninterpreted bit sequence. Each cell is versioned, meaning that it can contain multiple versions of the same data and that every version has a timestamp attached to it.

Bigtable aside, examples of common CFSs include Cassandra (see Section 4.3.1), HBase[16] and HyperTable[17]. Section 4.3.1 describes Cassandra in more detail.

### 4.3.1 Cassandra

#### Overview

Cassandra was initially developed at Facebook in 2007 to enable their Inbox Search feature, with which users quickly can search through their Facebook

[16] http://hbase.apache.org, accessed Jan 25, 2012.
[17] http://hypertable.org, accessed Jan 25, 2012.

message inbox [30]. In 2008, it was open sourced, and since 2010, it is a top-level Apache project.

Today, Cassandra is actively developed and contributors include both companies and individuals. It has several prominent users; examples include companies such as Netflix [24], Reddit [28], and Twitter [29]. It is also professionally supported by several third parties [44]. Among NoSQL technologies, it is certainly one of the most mature. The source code is available under Apache License 2.0[18].

### Data model

The outermost structure in Cassandra is a *cluster* of machines. Machines in a cluster cooperate to provide clients with data. Within a cluster, data is ordered hierarchically as shown in Figure 4.1. This hierarchy is now further described.

For every cluster, there is a set of *keyspaces*. A keyspace is identified by a unique string within its cluster. Within a keyspace, there is a set of CFs. These come in two different flavors; *standard* and *super*. The former is typically called just CF, whereas the latter usually is called *super CF* (SCF). Both a CF and an SCF are identified by a unique string within their keyspace. For both a CF and an SCF, the next level is *rows*. A row is uniquely identified within its (S)CF by its key, which is an uninterpreted bit sequence.

A row in a CF contains a set of *columns*. A row in an SCF contains a set of *super columns* (SCs), which in turn contain columns. A column is a triplet that contains a *name* (unique within its CF or SC), a *value* and a *timestamp*. The name and value are bit sequences, and the timestamp is a 64-bit integer. The value field is where applications store their actual data[19].

The data model is often referred to as a "multidimensional map" [30]. As can be seen in Figure 4.1, the total number of dimensions required before a column value is reached depends on the type of the enclosing CF. If it is a standard CF, the number of dimensions is four; for an SCF, it is five. Thus, this map is either four- or five-dimensional.

To get a better understanding of the data model, example instances of a CF and an SCF are suggested in Figures 4.2 and 4.3, respectively.

As for types, unless a column is defined in a schema (see **Metadata** below), the type of a value is unknown. Values are, as previously noted, just bit sequences. Cassandra supports types[20], in the sense that developers can use predefined (de)serializers[21] when encoding and decoding data from the database. For schema-defined columns, values are guaranteed to be deserializable by the set validation class.

### Metadata

A cluster stores and provides metadata about its keyspaces and (S)CFs. Among

[18] http://www.apache.org/licenses/LICENSE-2.0, accessed Jan 25, 2012.

[19] Actually, there are other places where application data can be stored. For instance, column names are bit sequences too, and it is certainly possible to store application data directly in these (possibly leaving the value field empty, or storing additional data there). In fact, doing this has even been documented in various design patterns [21].

[20] https://svn.apache.org/repos/asf/cassandra/trunk/src/java/org/apache/cassandra/db/marshal/, accessed Jan 25, 2012.

[21] The bit sequences are really byte arrays, so (de)serializers are methods with signatures `type` → `byte[]` and `byte[]` → `type`.

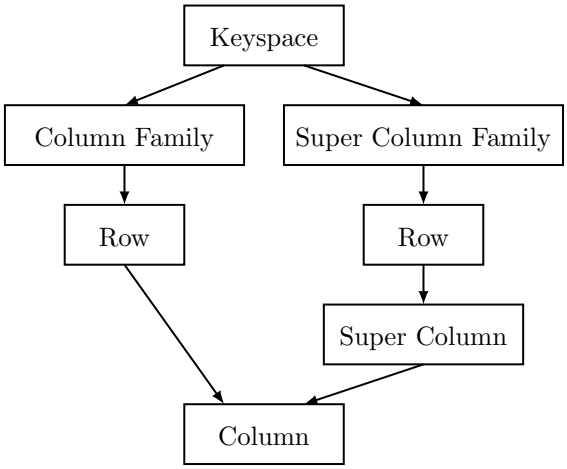


Figure 4.1: *Cassandra's data hierarchy. For all relationships, the cardinality is one-to-many. It can be seen that the number of dimensions required to reach a column, which holds the eventual data value, depends on the enclosing CF's type.*

other things, this always includes the names of these entities. More detailed metadata such as definitions of columns is optional and up to the data modeler. Should it be included, it is done through *schemas*. With a schema, data on all levels in the hierarchy (see Figure 4.1) can be configured. Columns are defined per (S)CF. A column definition includes a name, a validation class and possibly an index configuration. Schema-defined columns "exist" in all rows in the (S)CF. This does not, however, imply that all rows include data for these, but rather that individual rows cannot use already defined columns differently from how they have been configured, i.e., values must pass through the defined validation class.

### Indexing

Row keys are always automatically indexed. This means retrieving data by key is always fast. If a client wants to query on column values rather than row keys, separate indexes for these can be constructed. These are referred to as *secondary indexes*, and they are configured in the schema, as mentioned in **Metadata** above. Secondary indexes can only be built within CFs – not SCFs.

### Interface

Clients can communicate with Cassandra using the Thrift[22] API. The API provides methods through which clients can manipulate and query for data. On top of this, several third party developers have implemented high level client

[22] An Interface Definition Language which enables efficient and seamless communication between services and clients written in different programming languages [40].


Figure 4.2: *Example of a CF storing a small customer database. It can be seen that rows have unique keys that identifies them and that every row can define its own set of columns. Note that this is a simplified picture; timestamps are omitted, and all fields are really bit sequences.*

libraries. The purpose of these is to hide the low level details of the Thrift API, while still providing full functionality. On the Cassandra website, it is advised to write applications using high level clients[23], avoiding direct contact with the Thrift API.

Cassandra also supports its own query language; the Cassandra Query Language (CQL). CQL has many similarities to SQL in both syntax and exposed functionality, but they are not equivalent, nor does one subsume the other. CQL contains many Cassandra specific features and also does not offer as rich query functionality as SQL. The Thrift API provides a method through which CQL queries can be executed. For some programming languages such as Java and Python, there are CQL drivers. Also, the standard Cassandra distribution includes a command-line based CQL client, called `cqlsh`.

## 4.4 Graph Databases

The common theme among Graph Databases (GDs) is that data is structured in a mathematical *graph*. A graph $G = (V, E)$ generally consists of a set of *vertices* $V$ and a set of *edges* $E$. An edge $e \in E$ is a pair of vertices $(v_1, v_2) \in V \times V$. If the graph is *directed*, these pairs are ordered.

The data models between different GDs generally differ in what kind of graphs they support; whether it is directed, whether self-loops (i.e., edges $e = (v_1, v_2)$ where $v_1 = v_2$) are permitted, whether multi-edges (i.e., two or more identical edges between the same vertices) are permitted, and so on.

Examples of common GDs include: Neo4j (see Section 4.4.1), InfiniteGraph[24], HyperGraphDB[25], sones GraphDB[26] and InfoGrid[27]. Section 4.4.1 describes Neo4j in more detail.

[23] http://wiki.apache.org/cassandra/ClientOptions, accessed Jan 25, 2012.
[24] http://www.infinitegraph.com, accessed Jan 25, 2012.
[25] http://www.hypergraphdb.org, accessed Jan 25, 2012.
[26] http://www.sones.de/static-en/, accessed Jan 25, 2012.
[27] http://infogrid.org/wiki/Projects/ig-graphdb, accessed Jan 25, 2012.

**Row keys** **Super columns and columns**

| | general | | loyaltyProgram | |
|---|---|---|---|---|
| | name | location | clubCard | points |
| 0 | Müller | Zürich | 48 551 | 7,946 |
| | name | location | | |
| 12 | Smith | Seattle | | |
| | name | location | clubCard | |
| 67 | Andersson | Stockholm | 82 029 | |

Figure 4.3: *Example of an SCF storing a small customer database. It can be seen that a super column wraps a set of columns for each row. Like in Figure 4.2, this is a simplified picture; timestamps are omitted, and all fields are really bit sequences.*

### 4.4.1 Neo4j

#### Overview

Neo4j[28] is written in Java and is developed by the company Neo Technology[29]. It comes in both an open source and commercial variants, licensed under GPLv3[30] and AGPLv3 (just like MongoDB), respectively. Neo4j is today actively developed, has an active community and a long list of projects in which it is used [35].

#### Data model

For every Neo4j instance, there is exactly one graph. Graph vertices are called *nodes*, and edges are called *relationships*.

Every node contains a set of *properties*. A property is a key-value pair where the key is a Java string, and the value is a Java primitive (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` or `double`), `String` or an array of any of these. Every node also has a unique *id*, which is a nonnegative integer.

Relationships are always directed, i.e., they have a start and end node. Start and end node are not allowed to coincide, i.e., self-loops are not permitted. Just like nodes, relationships have a set of properties. Relationships also always have types. All there is to a relationship type is a defined name, which is a Java string.

An example of a Neo4j graph is shown in Figure 4.4. The graph stores a small customer database containing three customers.

[28] http://neo4j.org, accessed Jan 25, 2012.
[29] http://neotechnology.com, accessed Jan 25, 2012.
[30] http://www.gnu.org/licenses/gpl-3.0.html, accessed Jan 25, 2012.

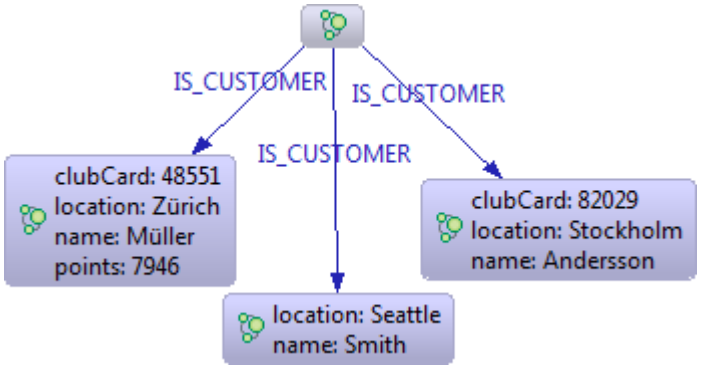


Figure 4.4: *Example of a Neo4j graph storing a customer database with three customers. Here, all customers are reachable from the empty reference node. The image is a screenshot from the graph visualization tool Neoclipse.*

### Metadata

The relevant metadata stored by Neo4j is the following:

- Reference node: A node that exists by default, without having been explicitly added by the user. It is supposed to be the "starting point" of the graph. Often, the whole graph is reachable from the reference node.
- Relationship types: All relationship types available for use in the graph.
- Highest node id: At least, this can be retrieved in constant time for any Neo4j instance. This can be relevant, e.g., when sampling (see Section 6.3.3).
- Indexes: The node and relationship indexes built (see **Indexing** below).

### Indexing

Neo4j supports indexing on both nodes and relationships. An index always has a name, a type (i.e., node or relationship) and a provider. The provider is by default Lucene[31], but it is also possible to use custom providers. A node index entry is a key-value pair pointing to a node. For instance, suppose a graph models a customer database with nodes being customers, an example index entry could then be the pair ("location", "Stockholm") that points to the node of some customer in Stockholm. Similarly, a relationship index entry is a key-value pair pointing to a relationship. Note that there can be several identical key-value pairs in the same index pointing to different nodes/relationships. Also note that the key-value pair does not necessarily have anything to do with the actual data in the indexed node/relationship[32].

[31] An open source information retrieval library. http://lucene.apache.org, accessed Jan 25, 2012.

[32] This differs from how indexes generally work in RDBMSs since in such, values themselves are index keys.

### Interface

It is possible to communicate with Neo4j in a variety of ways. If the communicating application is run on the same machine as the Neo4j instance, embedded mode can be used. This is supported for, e.g., Java and Python applications. To enable communication over network, Neo4j exposes a REST[33] API. Through this, all relevant operations on the graph can be performed.

A Neo4j graph is typically queried by traversals. A traversal is simply a walk in the graph. Typically, a traversal starts in a certain node, follows some specified relationships, and terminates according to some criterion. Neo4j supports its own traversal API methods and the two graph traversal languages Cypher[34] and Gremlin[35].

The Neo4j server is extendible. One way to add custom functionality is to write a server plugin. This is Java code archived into a JAR[36]-file that is deployed on the server. The functionality implemented by the server plugin is then exposed via the REST API.

[33] Representational State Transfer. A style of architecture for web services; runs over HTTP. [15]

[34] http://docs.neo4j.org/chunked/1.5/cypher-query-lang.html, accessed Jan 25, 2012.

[35] https://github.com/tinkerpop/gremlin/wiki, accessed Jan 25, 2012.

[36] Java Archive. An archive file format used to store Java class files, resources and metadata.

# Chapter 5

# Problem analysis and general solutions

The problem formulation (see Section 1.2) gave three bullets with concrete questions that the thesis attempts to answer:

- How do different NoSQL data models differ from Spotfire's data model?
- How do these differences affect extraction and import from Spotfire's point of view? How does the lack of explicitly defined schemas affect extraction and import?
- How can these problems be solved – enabling support for NoSQL databases in Spotfire?

Chapter 4 answered the question in the first bullet by describing the general data models of the four NoSQL families, together with a particular data model of one specific system per family. It could be seen that none of the models exactly matches Spotfire's tabular model.

In the rest of this chapter, the questions in the other two bullets are answered. Section 5.1 gives general answers, while Sections 5.2 – 5.5 give more specialized answers for each of the four NoSQL families.

## 5.1 General

Before attempting to answer the questions in the two last bullets in the chapter introduction, Spotfire's data model and the rationale behind it needs to be analyzed. As has been noted earlier, Spotfire's tabular data model closely resembles the relational model. But why is data in a data visualization and analysis tool represented by tables? First, representing data by tables makes importing data from RDBMSs a straightforward task – after all, RDBMSs have been the default solution for data management for decades. Second, structuring data in this fashion makes it easy to understand and work with.

A table is an instance of something more abstract; it is really a collection of comparable instances of some concept. Although this concept might be either an entity or a relationship, it is for the remainder of the thesis thought of as an

entity to simplify the discussion. Rows in the same table are intended to always model the same entity – whether the entity is a Person, a Flight, a Sale, or something else. The columns of the table represent the attributes of the entity instances, i.e., the dimensions along which the instances can be described, and thus compared. In the relational model, these dimensions are always atomic, i.e., they cannot be complex structures such as lists, maps, and so on. However, allowing only atomic attributes is not really a limitation of the model, since complex attributes can be modeled in another set of tables together with a definition of the relationship between these tables and the attributes.

Regarding the first question from bullet two above, an obvious problem that arises when importing data from NoSQL systems into Spotfire concerns how to convert the NoSQL data models into the Spotfire data model. As argued above, a table is a representation of a collection of comparable entity instances. Thus, enabling support for NoSQL data stores in Spotfire is all about enabling extraction and import of comparable entity instances into Spotfire tables[1]. The problems that must be solved in order to do this include:

a) how to let the user specify the comparable entity and its attributes,

b) how, and from where, to extract this exactly, and

c) how to convert attribute values into something that Spotfire permits.

Note that a) and b) are related. If a user specifies that an entity shall be extracted from structure $A$, then the data design in $A$ naturally describes the entity and its attributes. Of course, this rests on the assumption that structure $A$ stores exactly one entity and that each instance is designed in roughly the same way. Exactly this is the case in RDBMSs, and solving a) and b) is then done simply by asking the database for its tables and corresponding schemas, and subsequently letting the user choose one of these and the desired columns to import.

Because of the lack of explicit schemas, this cannot be done in NoSQL databases. However, given that there is a structure storing one entity, one way to solve this would be to let the user "blindly" give the full path to this structure and an entity definition textually. For example, assuming a Cassandra CF stores exactly one entity and that CF columns can be mapped to table columns, it would be feasible to let the user give the CF path, `<host>:<port>:<keyspace>:<CF>`, and columns `c1`, `c2`, ..., `cn`.

However, this is generally not preferable because the user might not know the exact structure of the database and also, typing errors that could lead to undesired results are easily made. Instead, a more guided approach is suggested: A schema can be inferred by sampling the database. This inferred schema can then be presented to the user who proceeds by selecting which attributes to import from the presented ones – just like in the RDBMS case. Of course, it should also be possible for the user to edit the presented schemas since the results may not be perfect because of the sampling. This general approach is from now on referred to as *schema inference*.

Naturally, sampling entire databases is rather costly. Instead, schemas should be inferred over smaller structures. In general, it is suggested to infer

[1]It is also conceivable to redesign Spotfire's data model into something that more closely resembles the NoSQL data models. However, in this thesis, Spotfire's data model is considered to be a given, and not something that can be changed.

schemas over structures that store exactly one entity. Of course, all databases do not necessarily have such structures. In such cases, structures storing multiple entities can be sampled, and the sample can subsequently be *clustered*, i.e., the different entity schemas can be separated into clusters so as to find one schema per entity (see Sections 3.4 and 6.3.6). Structures, e.g., Cassandra CFs or MongoDB collections, are usually part of the underlying NoSQL database's metadata and can thus be straightforwardly asked for, and the user can choose from these which one to sample and import from. An extension of this approach would be to allow sampling and extraction over several structures, but this is generally not needed since data is not modeled that way.

As for b) specifically, given what to extract, querying and retrieving data from an RDBMS is straightforward because of SQL. NoSQL systems generally all have different query interfaces supporting different kinds of queries. Thus, no general solution to this problem is suggested. Instead, an investigation of each particular system's interface must be conducted in order to solve this. Concerning where to extract from, it is, naturally, suggested that the same structure that was sampled also should be extracted from, since the inferred schema is valid for precisely that structure.

Solving c) is in general a simple task. If the underlying NoSQL system supports types (as in, e.g., Neo4j) then it is only a matter of type conversion. If it does not (as in, e.g., Cassandra) either some observed values can be analyzed and types can be inferred accordingly, or the user can be forced to explicitly give a conversion procedure.

## 5.2 Key-Value Stores

If one uses a KVS to store data with more than one dimension, the typical pattern is to put all dimensions into the key, separated by some special token, and store the values as usual [1]. For example, in an application storing a collection of the entity `User`, entries similar to the following can be expected:

```
(userid:56:username,  bar)
(userid:56:password,  782a30df3b2d8f0ed078016388cb50b6)
(userid:492:username, foo)
(userid:492:password, cc6c6102174b3050bc3397c724f00f63)
```

A sample of key-value pairs could be analyzed to find the general form of the keys. A result of such an analysis of the above set of pairs would be something similar to `userid:<int>:username` and `userid:<int>:password`. This could then be presented to the user who can choose to import any subset of the found structures. However, the structure of the keys can be very different in different applications. Coming up with an inference algorithm powerful enough to be generally useful is likely rather challenging. Instead of sampling and trying to infer a schema, perhaps it is more appropriate to let the user give only regular expressions for keys of pairs that should be extracted. Each expression could then be mapped to a table column. For example, extracting all `Users`' names and passwords in the running example would then be done when given the regular expressions `userid:[0-9]*:username` and `userid:[0-9]*:password`.

Since KVSs usually contain just one flat namespace with entries, the appropriate entity-storing structure to sample and extract from is simply the key-value

collection.

## 5.3 Document Stores

In DSs, documents conceptually correspond to relational table rows [33]; each document has a set of fields that closely resembles a table's set of columns. Thus, the suggested structure to sample and extract from is the structure that most closely wraps documents[2]. A document field can then be mapped to a table column.

Some DSs allow fields to store complex data such as lists or even other documents. Naturally, these fields cannot appropriately be converted into atomic Spotfire values. When the user wants to import a complex field into Spotfire, it should be compulsory to specify what atomic part of that structure that should be extracted, so as to flatten the structure so that it matches Spotfire's model. For example, if a document has a field storing a list of atomic values, the user should be forced to give an index in the list, or possibly several if more than one value in this list is desired to import.

## 5.4 Column Family Stores

In CFSs, a CF is conceptually similar to a table in the relational world [21]. The main difference is that tables always are rectangular, whereas each row in a CF can contain an arbitrary set of columns, and these do not have to be shared by other rows in the same CF. However, using a CF to store an entity is a common pattern [21], and thus the suggested entity-storing structure to sample and extract from is the CF. A CF column can then be mapped to a table column.

## 5.5 Graph Databases

In GDs, there is usually no finer structure than the graph itself. Thus, the suggested entity-storing structure to sample and extract from is simply the graph. The graph typically contains several different entities connected to each other in some fashion. For example, a movie database probably stores both `Movie`s and `Actor`s. Thus, GDs are examples where clustering, mentioned in Section 5.1, is relevant.

Entities are often stored as vertices, having some attached attributes describe them, similarly as a table's columns. Therefore, it is most important to support importing vertices and their attributes. However, edges could also have attributes that are interesting to analyze. Thus, two different import modes could be offered; one importing vertices and the other edges. In both cases, a vertex'/edge's attributes can be mapped to columns.

[2]For example, in CouchDB, the database is a flat collection of documents, whereas in MongoDB, there is a hierarchy and the closest structure storing documents is the collection (see Section 4.2.1).

# Chapter 6

# Implementation

Two proof of concept tools that can import data from Cassandra and Neo4j into Spotfire tables have been implemented. This chapter describes these in detail. First, in Section 6.1, an overview of required resources and concepts is given. Also, the tools' general architecture is described. Then, Sections 6.2 and 6.3 respectively deal with tool implementation details for the two aforementioned systems.

## 6.1 Overview

Here, a general overview of the two implemented tools is given. In Section 6.1.1, the Spotfire SDK[1] and API are introduced. Then follows a definition of the concept of On-Demand in Section 6.1.2. Lastly, in Section 6.1.3, the general tool architecture is given.

### 6.1.1 Spotfire SDK and API

Spotfire has a public SDK that provides developers with resources to extend the platform[2]. The platform allows for several extension points; examples include:

- *Tools* that perform analytic actions.
- *Transformations* that preprocess data prior to analysis.
- *Visualizations* that display data.

Custom extensions are written against the public Spotfire API. To help developers get started with this, TIBCO Software has published a set of extension tutorials and examples[3]. There is also the API documentation[4] that can be used for reference. The extensions can be written in any .NET language.

[1] Software Development Kit.
[2] http://stn.spotfire.com/stn/Extend/SDKOverview.aspx, accessed Jan 25, 2012.
[3] http://stn.spotfire.com/stn/Extend/Extending.aspx, accessed Jan 25, 2012.
[4] http://stn.spotfire.com/stn/API.aspx, accessed Jan 25, 2012.

### 6.1.2 On-Demand

The term *On-Demand* is present in various different contexts (e.g., *Video* and *Print On-Demand*) with slightly different meanings. In the context here, On-Demand means that data from a data source is fetched only when it is explicitly demanded, and that no other data than what is demanded is fetched. This stands in contrast to fetching data before it is demanded and/or fetching more data than is asked for.

The concept originates from a need to enable analysis of big data. With On-Demand, analyses are conducted in a top-down fashion where the user starts out with an aggregated view of a data set and then drills down to explore details as desired. The initial aggregated view can originate from, for instance, various ETL[5] processes.

### 6.1.3 General architecture

Both tools have been written in C# .NET. They are mainly based on the idea of fetching data On-Demand. This can be motivated by the fact that NoSQL databases are designed to contain large amounts of data, so extracting and presenting a structure in its entirety is often infeasible. This is, though, not always the case, which is why both tools also support exhaustive imports. These two import modes are referred to as On-Demand and Exhaustive mode, respectively.

Using Spotfire terminology, the two implementations are *tools* in an application context (i.e., subclasses of API class `CustomTool<AnalysisApplication>`) that both execute a *data function* (i.e., subclasses of API class `CustomDataFunctionExecutor`) for importing data from the underlying data source.

The two generic steps in which the user interacts with the tools are as follows:

1. Configuration of import, i.e., giving where data is located, what data to include in the resulting Spotfire table, how to import it and other data source specific parameters.

2. If On-Demand mode is chosen: Configuration of how On-Demand is controlled, i.e., giving definitions of rules for how to determine which values certain columns are limited to.

Column values can be limited to, for example, a marking in another table or an arbitrary expression in the Spotfire expression language[6]. In On-Demand mode, the user can choose to refresh the imported data manually or automatically when the value of a parameter controlling the On-Demand has changed. Step 1 above is tailored for each of the two tools because the underlying data sources differ. Step 2 uses existing Spotfire dialogs, and is thus identical for both tools.

When these two steps are done – and if in On-Demand mode, initial values for the On-Demand parameters have been set – data will be imported from the underlying data source into a Spotfire table and the user can analyze and visualize it.

[5]Extract, Transform, Load. A term common in data warehousing, referring to the process in which data is *extracted* from a source, *transformed* according to one's particular needs, and then *loaded* into another data source.

[6]http://stn.spotfire.com/spotfire_client_help/ncfe/ncfe_custom_expressions_introduction.htm, accessed Jan 25, 2012.

The extraction is executed in the data function. Depending on whether the mode is Exhaustive or On-Demand, the specific role of the data function differs.

In Exhaustive mode, the extraction happens once. The only input required is the import configuration. Using information from this, the data function communicates with the underlying data source and retrieves data accordingly. Finally, a Spotfire table is output.

In On-Demand mode, every time an On-Demand parameter value is changed and a refresh is issued, the table is updated with newly imported data. The data function therefore keeps track of the initial import configuration across all refreshes. For every import, the current On-Demand parameter values are passed to the data function. This is, for every column in the Spotfire table, a set of values – possibly the empty set, meaning no filter. The data function communicates with the underlying data source and fetches data from the configured mapping that matches the On-Demand parameter values. Finally, a Spotfire table is output. This procedure is illustrated in Figure 6.1.

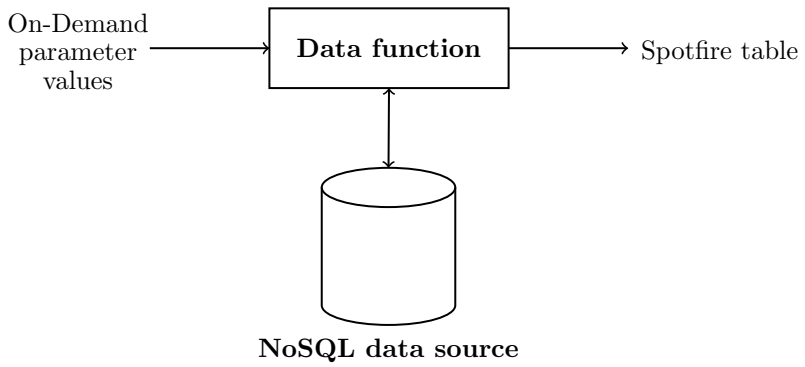


Figure 6.1: *The role of the data function in On-Demand mode. For every extraction, the current On-Demand parameter values are input. The data function communicates with the underlying data source, and finally outputs a Spotfire table.*

To consolidate the understanding of the data function's role in the two import modes, analogies to the relational world can be made: Assuming the Spotfire table has a 1:1 mapping to a data source table `t` and that columns `c_1`, `c_2`, ..., `c_N` are desired to import, in Exhaustive mode, the task of the data function is to respond to the following type of SQL query:

```
SELECT c_1, c_2, ..., c_N
FROM   t
```

In On-Demand mode, given that column `c_i` is limited to the set of values `v_i` = (`v_i_1`, `v_i_2`, ..., `v_i_Mi`) according to the On-Demand parameters, the data function's task is to respond to the following type of SQL query:

```
SELECT c_1, c_2, ..., c_N
FROM   t
```

```
WHERE  c_1 IN (v_1_1, v_1_2, ..., v_1_M1) AND
       c_2 IN (v_2_1, v_2_2, ..., v_2_M2) AND
       ...                                AND
       c_N IN (v_N_1, v_N_2, ..., v_N_MN)
```

It should be clarified that when all `v_i`'s are empty, no data should be returned.

Of course, the data functions cannot simply pass queries like these to the underlying NoSQL data sources since they do not support SQL. How this actually is handled in the implemented data functions is described in Sections 6.2.4 and 6.3.4.

## 6.2 Cassandra tool

Section 4.3.1 introduced Cassandra and its data model. In this section, it is described how extraction of tabular data from a Cassandra cluster has been implemented in the Cassandra tool. Section 6.2.1 motivates the use of Cassandra, and also describes choices made in the tool. Section 6.2.2 gives the architecture of the tool. Section 6.2.3 goes into depth on how an import is configured. Section 6.2.4 explains how the tool's data function has been implemented. Lastly, Section 6.2.5 describes the implemented pagination support.

### 6.2.1 Motivation and choices

As described in Section 4.3.1 Cassandra is actively developed, has several prominent users, is professionally supported by third parties and is one of the most mature systems within the NoSQL movement. These facts together with the host company's expressed interest in Cassandra were the reasons for choosing to implement support for it.

As Cassandra is actively developed, new versions are continuously released; and changes between versions can be quite substantial. Therefore, it should be pointed out that the implementation described here is based on version 1.0.0.

As described in Section 4.3.1, an application developer can either choose to communicate with Cassandra directly via the Thrift API or high level clients that add an abstraction layer on top of that. In the implementation described here, all communication is done directly via Thrift. Although, as mentioned in Section 4.3.1, it is advised that developers use high level clients, the choice for using the Thrift API directly was made for two main reasons:

- The high level clients are even less mature than Cassandra itself. This means the risk for stumbling into blocking bugs is rather high. Figuring out that something is a bug can be very time consuming, not to mention the time needed to solve or find a way to work around it.
- Whenever there is new functionality released for Cassandra, the high level client developers must first implement support for it on their end before it can be used in an application using such.

Concerning how to map Cassandra's data model to tables, the procedures suggested in Chapter 5 were chosen. That is, a CF is assumed to store an entity, and CF columns are mapped to Spotfire columns. The reasoning in the aforementioned chapter motivates this choice. As described in Section 4.3.1,

Cassandra also allows SCFs. Basically, SCFs just add another dimension on top of standard CFs. Therefore, if an SCF is sampled, a specific SC must subsequently be chosen to extract from.

### 6.2.2 Architecture

The general architecture of the implemented tools is described in Section 6.1.3. It is mentioned there that there is both a configuration of the import, and a data function that the tools execute. For both of these parts, the Cassandra tool must communicate with the underlying Cassandra cluster. To clarify how this is done and what entities are involved, Figure 6.2 gives an architectural overview of the tool and its surroundings.

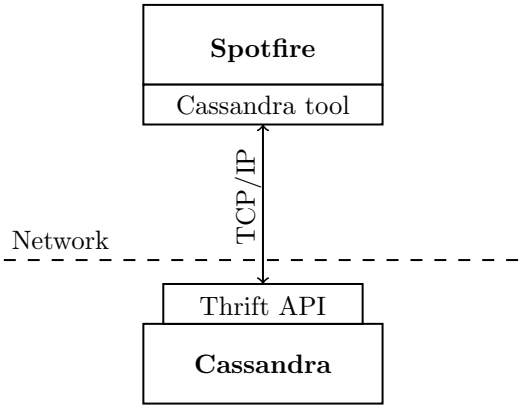


Figure 6.2: *Architectural overview of the Cassandra tool and its surrounding entities. It can be seen that communication between the tool and Cassandra is done using the Thrift API over TCP/IP.*

Section 6.2.1 mentioned that all communication between the Cassandra tool and the underlying Cassandra cluster is done via the Thrift API. Thrift allows for several different transport protocols. Typically, it is used on top of the TCP/IP stack using streaming sockets. This is how it is done here as well, as can be seen in Figure 6.2. The figure also shows where the network boundary is.

### 6.2.3 Configuration of import

Referring to the steps defined in Section 6.1.3, step 1 for Cassandra includes determining values for the following parameters:

- Data location and selection
  - Host and port.
  - Keyspace.

  - CF, or SC and related SCF. (Remember: a CF and SC are the two structures mapped to a table.)
  - Columns and their types.

- Other parameters
  - Consistency level: When reading from Cassandra, consistency level must always be given. The consistency level dictates how many replicas in the cluster that must to be read before a value is returned to the caller. Examples of permitted values are `ONE`, `QUORUM` and `ALL`. A full listing of permitted values and their meanings can be found in the Cassandra documentation[7].
  - Page size: When fetching data from Cassandra, depending on the size of the result set, it is not always desirable to query for and retrieve the results in one piece. It might be more efficient to query for and fetch the results in several smaller pieces. Therefore, support for paginating results has been implemented. This parameter determines the number of rows returned per page. (See Section 6.2.5 for how this is implemented, and Section 7.1 for related experiments.)

It is now described how the values of these are determined, in the same order as they are presented above.

Host and port are given by the user as a string. Once this has been entered, the user can ask for that particular cluster's keyspaces and (S)CFs. These are stored as metadata in Cassandra, so retrieving them is straightforward (`describe_keyspaces`[8]). The user proceeds by selecting one of the presented (S)CFs, and consequently corresponding keyspace.

Now, it is desirable to present the union of columns over all rows in the chosen (S)CF, so the user can import an arbitrary subset of these. Although Cassandra lets developers define schemas, this is optional. This means, when no schema is defined, Cassandra cannot be directly queried for the columns (or SCs) of a certain (S)CF. As indicated in Section 5.1, what instead can be done is to sample a number of rows and then the column set can be estimated based on that. This is how it is done here. As for schema-defined columns, these are always added to the column set based on information from stored metadata.

As described in Section 4.3.1, hierarchically, for CFs, columns follow rows. For SCFs, SCs follow rows, and then come columns. This means that if an SCF is chosen in the previous step, the user first has to select an SC after sampling before individual columns can be selected.

Consistency level is given as one of the permitted values mentioned above. Sample size and page size are given as integers.

When these values are given, the user proceeds by asking to sample the chosen (S)CF, and the tool queries Cassandra for the given amount of rows. In Cassandra, rows are always returned sorted. Sorting order is determined by the cluster's *partitioner*. In version 1.0.0, Cassandra provides `ByteOrderedPartitioner` and `RandomPartitioner` out-of-the-box. Users can also define custom partitioners. With `ByteOrderedPartitioner`, rows are sorted lexically

[7] http://wiki.apache.org/cassandra/API#ConsistencyLevel, accessed Jan 25, 2012.
[8] http://wiki.apache.org/cassandra/API#describe_keyspaces, accessed Jan 25, 2012.

by their keys. With `RandomPartitioner`, rows are sorted lexically by the MD5[9] hashes of their keys.

When asking Cassandra for a set of rows (`get_range_slices`[10]), one of the parameters is a *key range*[11]. This consists of a start and an end key for the desired rows. If no row keys are known, empty byte arrays can be given. For start and end key, respectively, this means the first and last row of the collection, as defined by the row sort order. Thus, Cassandra can be sampled by giving empty byte arrays as key range, and at the same time limiting the result size to the given sample size.

Once the cluster has been sampled, observed columns, and possibly SCs, are presented to the user. Since the sampling might not find everything, it is also possible to give columns and SCs explicitly – the user might know how the data is modeled.

In Spotfire, every column has a type (see Section 3.2). In Cassandra, if a column is defined in a schema, the `validation_class`[12] of that column is used as type. The `validation_class` is a Java class deployed on the Cassandra cluster which is used for validating all values before they are put into the column. Normally, this is any one of Cassandras supported types, but can also be a custom type.

If a column is not defined in a schema, the tool tries to infer the column type by analyzing observed values during the sampling. This is done in a straightforward fashion by, for the observed values of a certain column, trying to parse these to every supported type. Among the types for which all values could be parsed, the most specific is chosen. If this means no type, the column type is marked as unsupported.

In the tool, the supported types are `BooleanType`, `LongType` and `UTF8Type`, referring to their names in the Cassandra implementation. They all have obvious equivalents in Spotfire[13]. The user can change the inferred types to any of the tool-supported types. If the actual type is not among the supported ones, the column cannot be imported.

When all columns are present with types, the user selects a subset of them to import. Finally, the import mode, Exhaustive or On-Demand, is chosen, and step 1 is finished.

### 6.2.4 Data function

Section 6.1.3 described, among other things, the task of the data function. In Exhaustive mode, this boils down to retrieving a set of columns for all rows in a given CF or SC. This has straightforwardly been implemented by using the Thrift API method `get_range_slices`, with pagination added on top of it (see Section 6.2.5).

As for On-Demand mode, it is as an analogy in Section 6.1.3 concluded that the task boils down to responding to a certain type of SQL query. Cassandra,

[9] A cryptographic hash function.
http://tools.ietf.org/html/rfc1321, accessed Jan 25, 2012.
[10] http://wiki.apache.org/cassandra/API#get_range_slices, accessed Jan 25, 2012.
[11] http://wiki.apache.org/cassandra/API#KeyRange, accessed Jan 25, 2012.
[12] http://wiki.apache.org/cassandra/API#ColumnDef, accessed Jan 25, 2012.
[13] `BooleanType` ↦ `Boolean`; `LongType` ↦ `LongInteger`; `UTF8Type` ↦ `String`.

of course, does not support SQL. As mentioned in Section 4.3.1, Cassandra supports the Thrift API, including the method for executing CQL queries.

Referring to the generic SQL query in Section 6.1.3, using either regular methods in the Thrift API, or the one executing CQL queries, Cassandra supports equivalents of everything there except for the `IN` operator. Something similar, though, is supported: equivalents of `WHERE` clauses where every column is filtered to one value, rather than a set of them (i.e., a simple equality operator; `WHERE c = v`). This, however, is only true when at least one of the filtered columns is indexed (then, `get_indexed_slices`[14] can be used).

After some contemplation, one realizes that this query functionality is sufficient to implement the equivalent of the `IN` operator on top of it. For example, the query

```
SELECT director, genre, title
FROM   movies
WHERE  director IN ('Andersson', 'Tarantino')     AND
       genre    IN ('crime', 'drama', 'thriller')
```

can be translated into six different queries on the form

```
SELECT director, genre, title
FROM   movies
WHERE  director = v1 AND
       genre    = v2
```

where (`v1`, `v2`) take on the values { (`'Andersson'`, `'crime'`), (`'Andersson'`, `'drama'`), (`'Andersson'`, `'thriller'`), (`'Tarantino'`, `'crime'`), (`'Tarantino'`, `'drama'`), (`'Tarantino'`, `'thriller'`) }.

Formally, a query from Spotfire for a set of columns $C = \{c_1, c_2, ..., c_n\}$ with corresponding filters $S = \{S_1, S_2, ..., S_n\}$ can be translated into $\prod_i |S_i|$ Cassandra-supported queries where each query limits the column values to one combination of the filter values. As for pagination (see Section 6.2.5), this is done per Cassandra query, thus possibly increasing the actual number of calls to Cassandra. This is how it is done in the implemented data function.

### 6.2.5 Results pagination

It is mentioned in Section 6.2.3 that result sets can be paginated. This is relevant both when sampling and performing bulk imports, i.e., when using Thrift API methods `get_range_slices` and `get_indexed_slices`. It should be noted that pagination is not explicitly implemented in either the Cassandra back end, or the Thrift API – there is no `get_next_page` method or anything equivalent; `get_range_slices` and `get_indexed_slices` only return rows in a list whose size can be limited.

However, by utilizing the key range parameter, pagination can be implemented on the client-side. Suppose there is a certain query $Q$ that should be issued to Cassandra. Suppose further that the page size is set to $K$. Pagination can then be implemented as follows:

[14] http://wiki.apache.org/cassandra/API#get_indexed_slices, accessed Jan 25, 2012.

1. Issue $Q$, with the key range set to include all keys and the limit on the size of the results set to $K$. Cassandra will then return the first $K$ (at most) rows matching $Q$.
2. Present this (possibly empty) page of returned rows to the application.
3. If the returned number of rows is less than $K$, there are no more rows to fetch for $Q$ and the algorithm terminates.
4. Otherwise, issue $Q$ again, with the key range set to include all keys starting at the key of the last row on the previously fetched page. Again, limit the result size to $K$. Repeat from step 2.

Note that, because the semantics of the start key are inclusive, rows preceding page breaks are retrieved twice. This whole procedure is illustrated with an example in Figure 6.3.

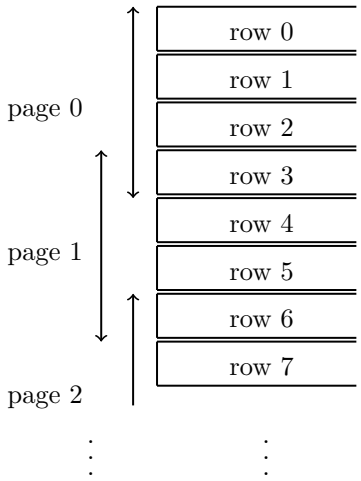


Figure 6.3: *Illustration of how pagination works. In the figure, the page size is set to 4. As can be seen, both page 0 and page 1 contain row 3. This is because row 3's key is used as start key when querying for page 1.*

For an original query $Q$, a general formula for the number of issued queries $M$ to Cassandra using pagination is now derived. Let $K$ denote the page size, and $N$ the size of the entire result set. When $N < K$, only one query is needed, because everything fits into one page. When $N = K$, two queries are needed, because even though everything fits into one page, that page is filled so another query must be issued to determine whether the result set has been depleted or not. When $N > K$, first an initial query is needed after which there are $N - K$ more rows to fetch. For subsequent queries, because of the page overlap described above, at most $K - 1$ new rows are fetched per page. Thus, the following formula can be derived:

$$M = 1 + \left\lceil \frac{\max(N - K + 1, 0)}{K - 1} \right\rceil \tag{6.1}$$

The max function is used to avoid negative numerators when $K > N + 1$. The added 1 in the first argument to max is required when $N = K$ to get the numerator greater than the denominator so that the fraction gets rounded up (remember: there will be two queries when $N = K$). Obviously, $K$ must be greater than 1.

It is apparent that this formula asymptotically grows as fast as $N/K$, which comes as no surprise since paginating $N$ results with a page size of $K$ roughly means splitting them into $N/K$ pages, and one query is used per page.

## 6.3 Neo4j tool

Section 4.4.1 introduced Neo4j and its data model. In this section, it is described how extraction of tabular data from Neo4j has been implemented in the Neo4j tool. Section 6.3.1 motivates the use of Neo4j, and also describes choices made in the tool. Section 6.3.2 gives the architecture of the tool. Section 6.3.3 goes into depth on how an import is configured. Section 6.3.4 explains how the tool's data function has been implemented. Section 6.3.5 describes the implemented pagination support. Lastly, Section 6.3.6 goes through the implemented clustering algorithms.

### 6.3.1 Motivation and choices

As described in Section 4.4.1 Neo4j is actively developed, has an active community and is widely used. These facts together with the host company's expressed interest in Neo4j were the reasons for choosing to implement support for it. The implementation described here is based on version 1.5.

As described in Section 4.4.1, the main way to communicate with a remote Neo4j instance is via a REST API over HTTP. Most relevant functionality for accessing data is exposed through this API. However, since the REST API runs over HTTP, and in some situations returns more data than is needed, communicating via the out-of-the-box REST API from Spotfire can be inefficient. Therefore, the choice was to create a server plugin (see Section 4.4.1) that implements Spotfire specific functionality server-side. Communicating with the server plugin is also done via a REST API – simply an extension of the original – but in this way, it can be assured that the returned responses are minimal and that the functionality exactly meets what Spotfire requires. Thus, efficiency is optimized for, with the drawback of having to deploy the server plugin on the server in the first place.

Concerning how to map Neo4j's data model to tables, the procedures suggested in Chapter 5 were chosen, with the exception that only one of the two modes is supported; importing nodes. That is, nodes are assumed to correspond to table rows, and the properties a node stores are thought of as table columns. The reasoning in the aforementioned chapter motivates this choice.

### 6.3.2 Architecture

The general architecture of the implemented tools is described in Section 6.1.3. It is mentioned there that there is both a configuration of the import, and a data function that the tools execute. For both of these parts, the Neo4j tool

communicates with the underlying Neo4j instance. To clarify how this is done and what entities are involved, Figure 6.4 gives an architectural overview of the tool and its surroundings.

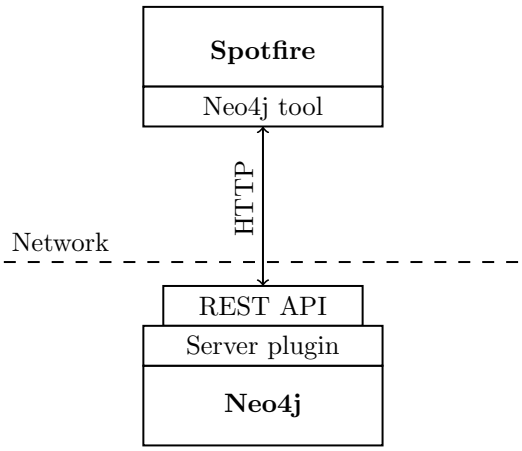


Figure 6.4: *Architectural overview of the Neo4j tool and its surrounding entities. It can be seen that communication between the tool and Neo4j is done over HTTP using a REST API.*

As can be seen in the figure, in addition to the two general tool parts, there is here a third part, namely the server plugin. The server plugin is written in Java and is deployed on the server as a JAR-file.

It is mentioned in Section 6.3.1 that all communication between the Neo4j tool and the underlying Neo4j instance is done via a REST API that exposes both out-of-the-box functionality and the functionality implemented by the server plugin. Figure 6.4 further illustrates this fact, and by showing where the network boundary is, the importance of minimizing the REST API responses is clarified.

### 6.3.3 Configuration of import

Referring to the steps defined in Section 6.1.3, step 1 for Neo4j includes determining values for the following parameters:

- Data location and selection
  - Host and port.
  - Node properties and their types. (Remember: a node and its properties are mapped to a table row.)
- Other parameters
  - Retrieval method: As described in Section 4.4.1 Neo4j's query model is mainly based on graph traversals. Thus, one way to retrieve data is by traversing the graph; starting at a given node, following edges of

certain types and terminating when a certain criterion has been met. Other ways include using indexes or performing exhaustive searches in the entire graph.

– Page size: As in the Cassandra tool, support for paginating results has been implemented. Here, this parameter determines the number of nodes returned per page. (See Section 6.3.5 for how this is implemented, and Section 7.1 for related experiments.)

It is now described how the values of these are determined, in the same order as they are presented above.

Host and port are given by the user as a string. For the user to be able to give the node properties with corresponding types that should be included in the import, it is desirable to know the schemas of the Neo4j instance. Here, this means the union of node properties for each entity in the underlying graph. The user can then select an arbitrary subset of these node properties for an entity, without having to enter them manually.

Neo4j does not keep metadata for this (see Section 4.4.1). As indicated in Section 5.1, what instead can be done is to sample a number of nodes and then infer schemas based on that. This is how it is done here.

To sample the graph, values for two parameters are required; sample size $N$ and sampling method. $N$ is given by the user as an integer. As for sampling method, the user can choose from three different methods that have been implemented:

- *Naïve*: Neo4j provides a method to iterate through all nodes in the graph (`GraphDatabaseService.getAllNodes`[15]). The resulting sample from the Naïve sampling method consists of the $N$ first nodes according to this iterator.

- *Naïve Traversal*: Starting in the reference node, this sampling method traverses the graph using a Breadth-First Search, following all edges in both directions, and returns a sample consisting of the $N$ first nodes observed in this traversal.

- *Random*: It is possible to get the highest id, $id_{high}$, of all nodes in the graph, in constant time. Using this, this sampling method repeatedly randomizes an integer $id \in [0, id_{high}]$ and checks whether a node with an id equal to $id$ exists (since nodes can be deleted, it is not guaranteed that there are nodes for all ids below $id_{high}$). If so, it adds that node to the sample. A new $id$ is then randomized and the procedure is repeated until $N$ nodes have been found. To avoid having this sampling method run forever, there is a parameter determining the maximum number of misses before it terminates. Thus, it is possible that it returns a sample with fewer than $N$ observations, even though the graph contains $N$, or more, nodes.

To determine how schemas are inferred, the user gets to choose how to cluster the results of the sampling – three choices are available:

[15] http://api.neo4j.org/1.5/org/neo4j/graphdb/GraphDatabaseService.html, accessed Jan 25, 2012.

- Do not cluster at all, i.e., treat each unique node structure as an entity. The node structure of node $X$ is defined as X's set of properties $P_X$, set of incoming relationships $I_X$ and set of outgoing relationships[16] $O_X$. Thus, two nodes $A$ and $B$ are considered structurally equal iff $P_A = P_B \wedge I_A = I_B \wedge O_A = O_B$. This method always returns the greatest number of entities, since the sample is not clustered at all.
- Cluster using a given number of clusters $K \geq 1$, i.e., compute an optimal clustering given that there must be $K$ clusters.
- Cluster using the "optimal" number of clusters, i.e., let the underlying algorithms determine how many distinct entities there are and cluster accordingly.

How the clustering is implemented is described in Section 6.3.6. For now, it suffices to think about the clustering as a grouping of observed nodes into entities, which relieves the user from work when setting up the import, plus it actually is a form of analysis in itself.

The structures of the entities resulting from the sampling and subsequent clustering are presented to the user. The structure of a cluster (i.e., a collection of observed nodes) is defined as the union of the cluster's node structures, for each of the three node structure constituents $P$, $I$ and $O$ (see above). The user proceeds by choosing one of the entities, adding or removing properties that should be imported, setting property types, and possibly also setting required properties and relationships that a node must include to be included in the import.

Neo4j properties are always typed (see Section 4.4.1). To the user, the proposed types are computed by a majority vote among the observed instances (ties are broken arbitrarily). All primitive types and strings are supported by the Neo4j tool, while arrays are not. For each of these, there is a corresponding Spotfire type that the property gets imported as[17].

After having chosen what to import, the user proceeds by choosing one of four retrieval methods. All methods return only properties for nodes that satisfy the conditions that have been set up. They differ in which nodes they consider, how they iterate through these and whether data is imported in Exhaustive or On-Demand mode. The retrieval methods that have been implemented are:

- *Exhaustive*: Data is imported exhaustively at once and all nodes in the entire graph are considered.
- *Exhaustive By Traversal*: Data is imported exhaustively at once, and the set of nodes that are considered is determined by a specified traversal definition. This includes the id of the start node, the maximum traversal depth and which relationships and corresponding directions to follow during the traversal.
- *On-Demand By Index*: Data is imported On-Demand, and the set of nodes that are considered is determined by what is found in a given node index (see Section 4.4.1). It is assumed that the underlying index provider is Lucene and that key-value pairs are property name and value, respectively.

[16] That is, property *names* and relationship *type names*.

[17] `bool` $\mapsto$ `Boolean`; `byte`, `short`, `int` $\mapsto$ `Integer`; `long` $\mapsto$ `LongInteger`; `Float` $\mapsto$ `SingleReal`; `Double` $\mapsto$ `Real`; `char`, `String` $\mapsto$ `String`.

- *On-Demand By Traversal*: Data is imported On-Demand and the set of nodes considered is determined just like in Exhaustive By Traversal above.

Finally, page size is given as an integer and step 1 is finished.

### 6.3.4 Data function

Section 6.1.3 described, among other things, the general task of the data function. Section 6.3.3 mentioned that there are four different retrieval modes to choose from. For all these, the data function begins by issuing a query for the desired data to the server plugin. The server plugin creates an iterator over the nodes that are considered for that particular retrieval mode and query. For each node in the iterator, it is checked that it contains the required properties and relationships (if any), and also that it satisfies given On-Demand filters (if any). Values of desired properties in the resulting set of nodes are then retrieved by the data function page by page (see Section 6.3.5) and are finally presented in Spotfire as a table.

Which nodes that are considered and how these are fetched depends on the retrieval mode. For Exhaustive, `GraphDatabaseService.getAllNodes` is called. For Exhaustive By Traversal and On-Demand By Traversal, `Node.traverse`[18] is called. For On-Demand By Index, `ReadableIndex.query`[19] is called with a Lucene query string as an argument. Note that the Lucene query language supports equivalents of all operators in the SQL query presented in Section 6.1.3, which is why no special tricks are needed for this (which is not the case in the Cassandra tool – see Section 6.2.4).

### 6.3.5 Results pagination

It is mentioned in Section 6.3.3 that result sets can be paginated. This is an important feature since for each message that the Neo4j server sends, heap space corresponding to the message's size must be allocated. Thus, to keep the server's memory footprint low while working with large result sets, pagination is required.

The implemented pagination works as follows: Each time the Neo4j tool wants data, a query method in the server plugin is invoked. This query method creates and stores a lazy iterator over the results[20] and returns a UUID[21], which is a unique identifier of that particular query throughout its lifetime. When the Neo4j tool wants the actual results from this query, it repeatedly asks the server plugin for the next page, using the aforementioned UUID as an argument, until the result set is depleted.

As can be understood from the discussion above, the server plugin is stateful. To avoid performance degradation over time because of stored iterators that will never be used (because of, e.g., canceled reads), queries that have not been completely fetched within a parameter of $T$ seconds are removed. Depending

[18] http://api.neo4j.org/1.5/org/neo4j/graphdb/Node.html, accessed Jan 25, 2012.

[19] http://api.neo4j.org/1.5/org/neo4j/graphdb/index/ReadableIndex.html, accessed Jan 25, 2012.

[20] That is, the results per se are not stored, only a description of how they shall be retrieved once they are explicitly asked for is. This makes the memory footprint of each query very low.

[21] Universally Unique Identifier. A key randomly generated according to an algorithm with the property that the probability of having two generated UUIDs coincide is practically zero.

on the size of the result sets and how fast they can be depleted, a reasonable value for $T$ is on the order of minutes.

For an original query $Q$, a general formula for the number of issued queries $M$ to the server plugin using pagination is

$$M = 1 + \left\lceil \frac{\max(N - K + 1, 0)}{K} \right\rceil \tag{6.2}$$

where $N$ is the size of the entire result set, and $K$ the page size. The derivation of $M$ is identical to the derivation of corresponding $M$ in the Cassandra tool (see Section 6.2.5), except that here, there are always $K$ fresh values per page, so the denominator here is $K$ instead of $K - 1$. As in the Cassandra case, this formula grows as fast as $N/K$.

### 6.3.6 Clustering

Section 6.3.3 explained that a node sample can be clustered. The goal of the clustering is to distinguish different entities within the sample, which in turn helps an analyst analyze a certain Neo4j instance. Clustering in general was introduced in Section 3.4.

The clustering algorithm implemented is K-medoids. This can be motivated by it being a general algorithm that can be used when clustering objects with qualitative variables. An input to K-medoids is a distance function between objects – here, nodes. As Hastie et al. mention, there is no generic method for finding a distance function – rather, careful thought with consideration to the actual domain is required for good results [20]. Here, it is desirable to find a distance measure which is general for all graphs, and at the same time sufficiently powerful to be able to distinguish entities in every particular instance.

As Section 4.4.1 explained, every Neo4j node has a set of properties and a set of directed relationships to other nodes; this is all that characterizes a node. It is reasonable to assume that if two nodes have identical sets of properties and relationships, then they model the same entity. The more these sets differ, the more likely it is that they model different entities. A general measure of the degree of overlap between two sets $A$ and $B$ is the Jaccard coefficient [31]:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{6.3}$$

This can be extended to measure the degree of difference, through a measure known as the Soergel distance[22] [50]:

$$S(A, B) = 1 - J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{6.4}$$

Since the direction of a relationship generally has meaning[23], the following general node distance function $d$ between two nodes $A$ and $B$ is therefore suggested:

$$d(A, B) = w_P S(P_A, P_B) + w_I S(I_A, I_B) + w_O S(O_A, O_B) \tag{6.5}$$

[22] When $|A \cup B| = 0$, $S(A, B)$ can be defined to zero.

[23] For instance, if a customer node $C$ has a relationship *bought* to a product $P$, this most likely means that $C$ bought $P$, and not vice versa. In fact, a customer probably never has an incoming *bought* relationship, but certainly outgoing such. Thus, relationships' directions are discriminative when it comes to separating entities.

$P_X$ denotes node $X$'s set of properties. $I_X$ and $O_X$ denote node $X$'s set of incoming and outgoing relationships[24], respectively. $w_P$, $w_I$ and $w_O$ are non-negative real-valued weights of the individual Soergel distances between the properties, incoming and outgoing relationships, respectively. These can be configured as desired, but generally, it is expected that a node's properties discriminate more than its relationships, and that incoming relationships discriminate equally much as outgoing such, which is why a configuration where $w_P > w_I = w_O$ is recommended. As can be seen by inspecting Equation 6.5, $d$ is maximized when the individual Soergel distances are maximized, i.e., when all intersections are empty. Conversely, $d$ is minimized when all intersections are equal to corresponding unions.

This distance function $d$ is the one used in the implementation. The weights have been configured to $w_P = 1.0$ and $w_I = w_O = 0.5$. As for seeding method, uniformly randomized, K-means++ and Greedy seeding have been implemented. The last one mentioned is the one used in the actual tool. This can be motivated by the experimental results described in Section 7.2. The presented clustering is the one with smallest intra-cluster dissimilarity over 30 runs.

Section 6.3.3 mentioned that a sample can be clustered using an "optimal" number of clusters. The algorithm used to determine this in the implemented tool is the Gap statistic. The discussion in Section 3.4 motivates this choice. The Gap statistic requires the number of reference datasets $B$ and a reference distribution $R$ as input. Tibshirani et al. do not give recommendations for how to choose $B$ [46]. However, in Dudoit and Fridlyand's study, $B = 10$ was used – a value that is used in this implementation too [13]. As for $R$, none of the two choices considered by Tibshirani et al. is applicable here. Instead, a reference node is constructed as follows:

1. Given the set of observed property set sizes, generate a reference property set size $N_P$ uniformly at random from these.

2. Given the set of observed properties $P$, generate a reference property set by picking $N_P$ properties uniformly at random from $P$.

3. Repeat steps 1 and 2 in corresponding versions for incoming and outgoing relationships.

Since the Gap statistic is not deterministic, the presented value is decided by a majority vote over 30 runs (ties are broken arbitrarily).

[24]That is, property *names* and relationship *type names*.

# Chapter 7

# Experiments

This chapter describes the methods and results of the experiments conducted to evaluate some of the behavior in the two implemented tools (see Chapter 6). First, Section 7.1 explains the experiments concerning the results pagination for both the Cassandra and the Neo4j tool. Then, Section 7.2 gives a detailed description of how the implemented clustering algorithms in the Neo4j tool perform on a real world dataset.

## 7.1 Results pagination

Although the general read performance of both Cassandra and Neo4j is given by their implementations, the way in which clients query for data affects retrieval speed. In both implemented tools, result sets can be paginated (see Sections 6.2.5 and 6.3.5). The only tunable parameter when paginating results is the page size. Naturally, many more parameters than page size affect retrieval speed, e.g., the number of network hops to the server, network load, server load and how much of the requested data the server keeps in RAM already. During the experiments, efforts were made to keep these factors fixed. Also, note that the actual retrieval times measured are not the interesting parts of the experiments; rather, the shape of the retrieval time curve as a function of the page size is.

In the Cassandra tool experiment, a one-machine cluster using the default Cassandra configuration held all data. The machine was a modern desktop computer with a 3 GHz dual-core processor and 4 GB RAM running Windows 7. It was located one network hop away from the Spotfire client on a Gigabit Ethernet. In the Neo4j tool experiment, the same machine was used. The default Neo4j configuration was used, except that the maximum allowed Java heap size was changed from 64 to 256 MB to enable the use of large page sizes.

In the Cassandra tool, setting the page size to $P$ means that each query sent through the Thrift API asks for at most $P$ rows (columns cannot be paginated). The data used for the Cassandra tool experiment was the first 100,000 rows of the Airline On-Time dataset from 1987[1]. These were all inserted into a single CF. All `NA`-valued columns were omitted (remember: in Cassandra, there is no need to enter dummy values for columns whose values really are missing).

[1] http://stat-computing.org/dataexpo/2009/1987.csv.bz2, accessed Jan 25, 2012.

Numerical values were stored as 64-bit integers. String values were stored as UTF-8[2] strings. In the experiment, all of these 100,000 rows were imported into a Spotfire table, varying the page size between 10, 100, 1,000, 10,000 and 100,000. The total retrieval time from starting the import until there was a table in Spotfire was measured and averaged over three runs. As a comparison, the same experiment was repeated for a CF containing 100,000 rows where each row contained one big column whose value consisted of as many bytes as any Airline On-Time row. Thus, the same amount of data was imported in the two cases. The results of these experiments are shown in Figure 7.1. Retrieval times are shown in seconds; and note that the horizontal axis is logarithmic.

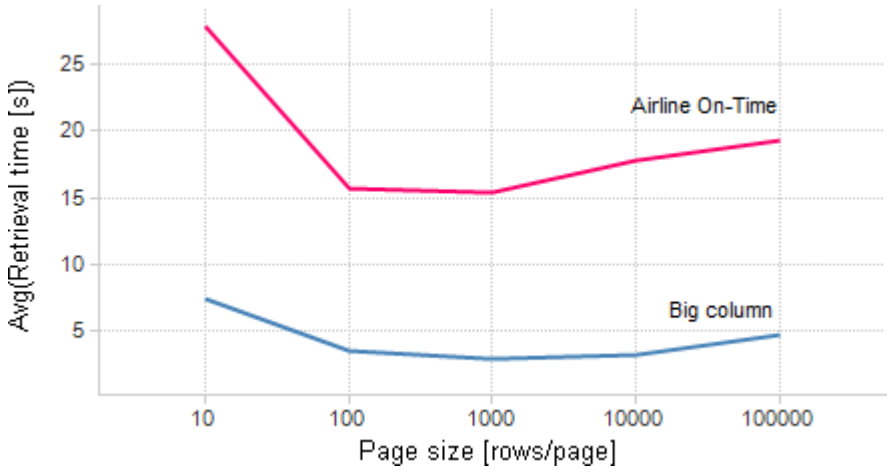


Figure 7.1: *Average number of seconds needed to fetch 100,000 rows from a one-machine Cassandra cluster for two different datasets, as a function of the page size. The graph shows that the retrieval time curve has a similar shape in the two cases.*

In the Neo4j tool, setting the page size to $P$ means that each query sent through the REST API asks for at most $P$ nodes (properties cannot be paginated). The dataset used for the Neo4j tool experiment was the entire Cineasts Dataset described in detail in Section 7.2. All properties for all nodes were fetched. The experiment was conducted in a similar fashion to the Cassandra experiment, but page sizes 10 and 100 were not used since they would have resulted in excessively long retrieval times. Figure 7.2 shows the results. Again, retrieval times are in seconds, and the horizontal axis is logarithmic.

In the two figures, it can be seen that the retrieval time curve is always convex. This means that optimizing the retrieval time is not as easy as picking the smallest or greatest page size available, but rather searching for something in between. Naturally, these experiments are far from exhaustive. The findings can, nevertheless, serve as a starting point for a more thorough investigation for a potential production implementation.

[2]UCS Transformation Format, 8 bits.

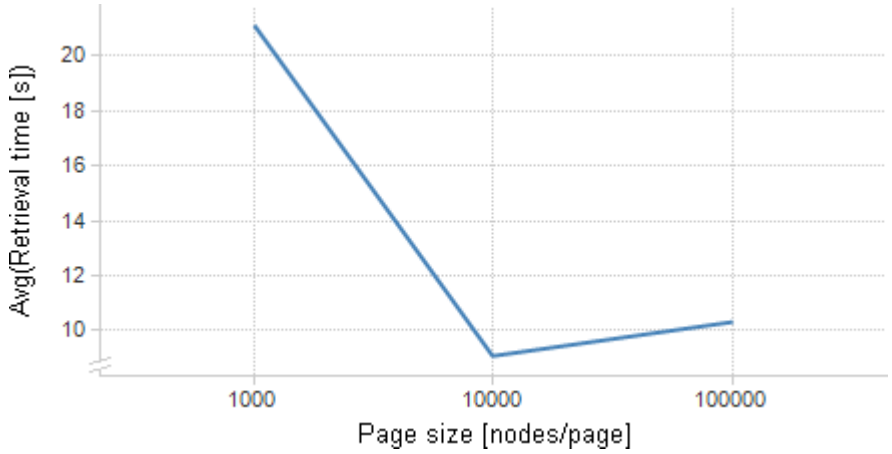


Figure 7.2: *Average number of seconds needed to fetch all properties from 63,045 Neo4j nodes as a function of the page size.*

## 7.2 Clustering

To evaluate the performance of the implemented clustering algorithms in the Neo4j tool, three experiments have been conducted. The dataset that has been used is the Cineasts Dataset (CD) provided on the Neo4j website[3].

CD originates from a Neo4j application example. It contains real world data about movies, actors and directors, mined from themoviedb.org[4], plus some application-specific user data (the application added a social layer on top of a movie database). CD contains in total 63,045 nodes, 106,665 relationships, 719,318 properties and five relationship types. From the domain, there are three distinct entities: `Movie`, `Person` and `User`. `Movie`s and `Person`s respectively model movies and persons directly related to movies, i.e., actors and directors. `User`s model users in the aforementioned social layer of the database. The node counts of these respective entities are 12,862, 50,134 and 48. There is also the reference node, which models nothing else than itself.

The five relationship types are `ACTS_IN`, `DIRECTED`, `FRIEND`, `RATED` and `ROOT`. `ACTS_IN` is a relationship from a `Person` to a `Movie` indicating that the former is an actor in the latter. `DIRECTED` works similarly but for directors. `FRIEND` denotes a friendship from one `User` to another. `RATED` indicates that a `User` rated a `Movie`. Lastly, `ROOT` is the relationship type used from the reference node to connect it to the rest of the graph.

Before the experiments are described, note that the algorithms only consider the set of structurally distinct nodes (see Section 6.3.3) as input. In CD, although there are 63,044 nodes in total, there are only 140 distinct node structures; 115, 18 and 7 `Movie`s, `Person`s and `User`s, respectively. Thus, it is possible to use the entire dataset as input in the experiments. This is also the case in

[3]http://example-data.neo4j.org/files/cineasts_movies_actors.zip, accessed Jan 25, 2012.
[4]http://www.themoviedb.org, accessed Jan 25, 2012.

the first experiment, but as shall be seen, something that will be varied in experiments two and three.

In the first experiment, the number of iterations before K-medoids converged was measured for uniformly randomized, K-means++ and Greedy seeding (see Section 3.4). An iteration was considered to be computing steps 2 and 3 in Algorithm 3.1. The dataset was clustered 500 times each for $K = 1, 2, ..., 10$, and the number of iterations was then averaged and plotted against $K$. The resulting graph can be seen in Figure 7.3. In the figure, mainly three things can be observed. First, clusterings converge very quickly in general. In practice, this means very fast execution times. Second, K-means++ seeding is consistently superior to uniformly randomized seeding. Third, Greedy seeding gives the fastest convergence for $K \leq 3$ but ends up performing worst of the three for $K \geq 7$.

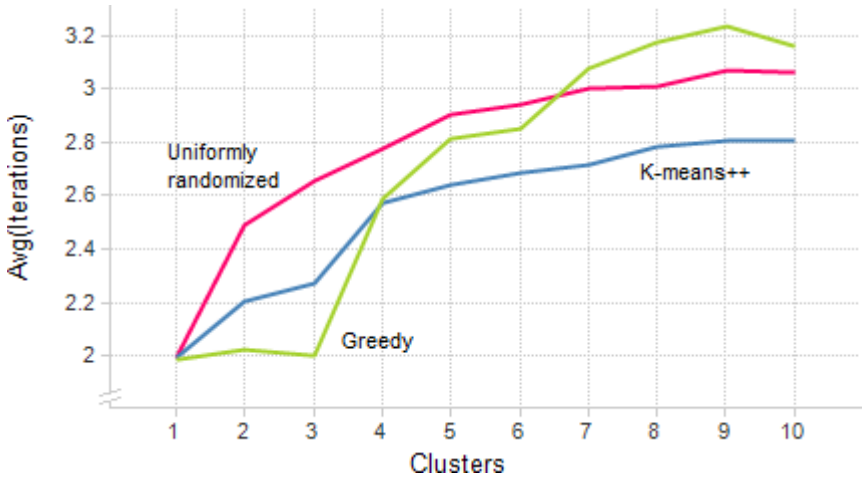


Figure 7.3: *The average number of iterations before K-medoids converged. It can be seen that, in general, clusterings converge very quickly, that K-means++ seeding consistently outperforms uniformly randomized seeding, and that Greedy seeding is sometimes better and sometimes worse than the other two methods.*

Given that the clustering algorithms converge sufficiently fast, the most interesting part is to see what the resulting clusters look like and whether the algorithms are able to distinguish between `Movie`s, `Person`s and `User`s in a sensible way when given the "correct" $K$. Therefore, the following second experiment was conducted: For a given input, K-medoids with $K = 3$ was run 50 times, and from the resulting clusterings, the one with the smallest intra-cluster dissimilarity was saved. In this clustering, clusters were labeled `Movie`, `Person` or `User` according to a majority vote among each cluster's members. Using this labeling, *purity* was then computed. Purity is defined as the portion of all objects that are assigned correctly, according to the computed cluster labeling [31]. These numbers were averaged over 500 runs for uniformly randomized, K-means++ and Greedy seeding.

The experiment was run on three different input configurations; "115/18/7",

| | Number of Movies/Persons/Users | | | | | |
|---|---|---|---|---|---|---|
| | 115/18/7 | | 40/18/7 | | 7/7/7 | |
| | Freq. | Purity | Freq. | Purity | Freq. | Purity |
| Movie/Movie/Movie | 39 | 82 | 0 | 62 | 0 | - |
| Movie/Movie/Person | 43 | 95 | 58 | 89 | 11 | 67 |
| Movie/Movie/User | 10 | 86 | 0 | 71 | 0 | - |
| Movie/Person/Person | 3 | 95 | 13 | 89 | 4 | 67 |
| Movie/Person/User* | 5 | 100 | 25 | 99 | 71 | 98 |
| Movie/User/User | 0 | 87 | 3 | 72 | 12 | 66 |
| Person/Person/User | 0 | - | 0 | - | 2 | 67 |
| Person/User/User | 0 | - | 0 | - | 0 | 67 |

Table 7.1: *Cluster label and purity percentages over three different data sets when using uniformly randomized seeding. As shown, the algorithms have some trouble giving the desired output (asterisk marked)* `Movie/Person/User` *when the input is skewed towards* `Movies`*.*

| | Number of Movies/Persons/Users | | | | | |
|---|---|---|---|---|---|---|
| | 115/18/7 | | 40/18/7 | | 7/7/7 | |
| | Freq. | Purity | Freq. | Purity | Freq. | Purity |
| Movie/Movie/Movie | 5 | 82 | 0 | - | 0 | - |
| Movie/Movie/Person | 47 | 95 | 30 | 89 | 3 | 67 |
| Movie/Movie/User | 16 | 86 | 0 | 71 | 0 | - |
| Movie/Person/Person | 3 | 95 | 6 | 89 | 0 | 67 |
| Movie/Person/User* | 29 | 100 | 63 | 100 | 92 | 99 |
| Movie/User/User | 0 | - | 1 | 72 | 4 | 66 |
| Person/Person/User | 0 | - | 0 | - | 0 | 67 |
| Person/User/User | 0 | - | 0 | - | 1 | 67 |

Table 7.2: *Cluster label and purity percentages over three different data sets when using K-means++ seeding. As shown, the algorithms have some trouble giving the desired output (asterisk marked)* `Movie/Person/User` *when the input is skewed towards* `Movies`*.*

"40/18/7" and "7/7/7". These differ in how many instances of each entity they contain. The notation "$x/y/z$" denotes $x$ `Movie`s, $y$ `Person`s and $z$ `User`s. Thus, "115/18/7" denotes the entire CD. For the other two input configurations, the given number of distinct node structures for every entity were picked uniformly at random from CD for each run.

The results are presented as rounded percentages in Tables 7.1, 7.2 and 7.3 for uniformly randomized, K-means++ and Greedy seeding, respectively. In the tables, a relative frequency ("Freq.") value $F$ for a given combination of entities $C$ means that $C$ was the resulting cluster labeling in $F$ % of the cases. A purity value $P$ means that the average purity was $P$ % for $C$. The desired output `Movie/Person/User` is highlighted with an asterisk in all tables.

As can be noted by inspecting the tables, K-medoids together with uniformly randomized and K-means++ seeding has a hard time producing the desired output `Movie/Person/User` for the entire CD ("115/18/7"). This is because the `Movies` simply outnumber the `Persons` and the `Users`, so cluster initializations are not always well separated. When the distribution of entities is more equal

| | Number of Movies/Persons/Users | | | | | |
|---|---|---|---|---|---|---|
| | 115/18/7 | | 40/18/7 | | 7/7/7 | |
| | Freq. | Purity | Freq. | Purity | Freq. | Purity |
| Movie/Person/User* | 100 | 100 | 100 | 100 | 100 | 99 |

Table 7.3: *Cluster label and purity percentages over three different data sets when using Greedy seeding. As shown, the algorithms always produce the desired output (asterisk marked)* `Movie/Person/User` *and very close to completely pure clusters.*

in the input, the algorithms much more often produce a desired clustering with high purity. It can also be seen that K-means++ seeding readily improves the results as compared to the uniformly randomized seeding, and, lastly, that Greedy seeding appears to be clearly outstanding in this regard.

In the third experiment, the performance of the Gap statistic was evaluated. For an algorithm estimating the optimal number of clusters, in particular two things are interesting to measure: how consistent the result is (since the algorithm is randomized) and how close the result is to the "correct" number. Therefore, the Gap statistic was run 500 times and the number of times each specific $K$ was output was counted. Again, this was done using uniformly randomized, K-means++ and Greedy seeding when clustering. Also, the same input configurations as in the second experiment were used. Figures 7.4, 7.5 and 7.6 show the results for "115/18/7", "40/18/7" and "7/7/7" respectively. In the figures, for a given number of clusters $K$, the first, second and third bar respectively show how often that particular $K$ was output using uniformly randomized, K-means++ and Greedy seeding. The number of clusters ranges from 1-7 since this covers all the outputs.


Figure 7.4: *Estimates of the optimal number of clusters by the Gap statistic for "115/18/7". The bars within each group, from left to right, respectively represent uniformly randomized, K-means++ and Greedy seeding.*


Figure 7.5: *Estimates of the optimal number of clusters by the Gap statistic for "40/18/7". The bars within each group, from left to right, respectively represent uniformly randomized, K-means++ and Greedy seeding.*

As can be observed, none of the seeding methods produce $K = 3$ most often for "115/18/7". For "40/18/7", they all do. Finally, for "7/7/7", both uniformly randomized and Greedy seeding do. It can also be noted that Greedy seeding produces $K = 3$ in a large majority of the cases for both "40/18/7" and "7/7/7".

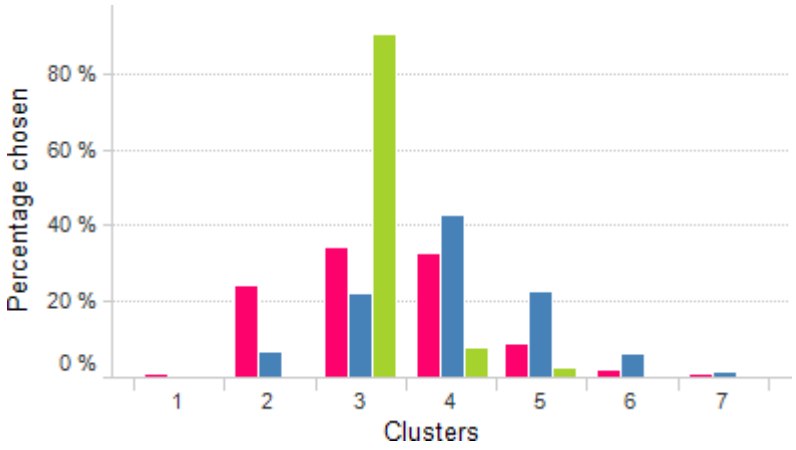


Figure 7.6: *Estimates of the optimal number of clusters by the Gap statistic for "7/7/7". The bars within each group, from left to right, respectively represent uniformly randomized, K-means++ and Greedy seeding.*

# Chapter 8

# Discussion and future work

In this chapter the implemented tools the overall approach are discussed. Also, the points considered most important to investigate further are mentioned. Sections 8.1 and 8.2 discuss the Cassandra and Neo4j tools, respectively. Suggestions for future work are presented for both of these. Section 8.3 discusses whether the tools' sample and page size should be determined automatically by the application. In Section 8.4, the architectural differences in terms of client and server execution in the two implemented tools are discussed. In Section 8.5, Spotfire's inherent fit for NoSQL data is discussed. Section 8.6 finally discusses which role the user's level of expertise has for solving the problem of supporting NoSQL systems.

## 8.1 Cassandra tool

In the implemented Cassandra tool, only one sampling method is mentioned. This method's resulting sample is based on the first $N$ rows in the sampled (S)CF (see Section 6.2.3). There is one issue with this method that deserves to be mentioned. All the returned rows are consecutive in terms of the sort order, which, depending on how the data is modeled, might have some actual meaning, and could therefore limit the representativeness of the sample. This, however, is inherent to Cassandra's query model and is hard to get away from. Although not obvious exactly how it could be implemented, it would be useful to have alternatives to this sampling method to avoid this issue. Offering multiple sampling methods are, though, not as important here as in the Neo4j tool since it is assumed that the designs of the rows within the sampled (S)CF are similar, whereas in Neo4j different nodes could model different entities. It should be noted that if the cluster is configured to use the `RandomPartitioner`, rows are returned semantically randomly ordered; at least as random as the MD5 algorithm allows for.

As for types, it would be possible to allow users to write their own type converters between Cassandra bit sequences and the Spotfire types. These could then be used when importing values from Cassandra into Spotfire. One way to enable this is by letting users input DLLs[1] that implement some type converter interface.

[1] Dynamic-Link Libraries. Files containing functions that can be shared between programs.

In Section 6.2.4, it is described how the data function is implemented in On-Demand mode by using the API method `get_index_slices`. This approach is feasible only when at least one of the filtered columns is indexed. However, in CFs, column indexes might not have been built. And even worse, in SCFs, column indexes can *never* be built (see Section 4.3.1). In such cases, there is no selection mechanism in Cassandra. In the implementation, relevant columns for *all* rows in the underlying CF or SC are then fetched, and filtering is performed client-side. Naturally, this is prohibitively expensive, and in practise probably an infeasible way of working. However, this is a limitation (from this point of view) of Cassandra that has to be dealt with. To avoid these problems, it could, for example, be enforced that On-Demand imports are only allowed when columns are indexed, or suggested to the user to build indexes when configuring the import.

The most relevant future work for the Cassandra tool is considered to be adding support for more types and exploring whether it is possible for a more sophisticated type inference algorithm than the one currently used.

## 8.2 Neo4j tool

The Cassandra and Neo4j tools conceptually differ in that in the former, it is assumed that there is exactly one entity per sampled structure (i.e., the (S)CF), whereas in the latter, it is assumed that there are possibly several entities per sampled structure (i.e., the graph). This was the reason for adding clustering support in the Neo4j tool.

When trying to infer a schema by sampling and clustering, there are two error sources. First, the sampling might not give a representative sample. This could either be caused by a too small sample size, or an inappropriate sampling method. Second, the clustering might not give the "real" clusters. Thus, to improve the overall schema inference, both of these parts can be improved on. Naturally, the clustering deserves more focus, as it is harder to get right.

Looking back on the results from the clustering experiments in Section 7.2, although it is not an exhaustive evaluation, it can be seen that convergence speed is not a problem. Another observation is that the choice of the seeding method obviously has a big impact on the end result. As it seems, the Greedy seeding is superior to both uniformly randomized and K-means++ seeding, at least for this particular dataset. When $K$ is the real number of entities and entities are truly separable, this makes sense. As long as each cluster gets initialized with an instance that is rather representable for its entity, the other instances will be clustered correctly, given a good distance measure. If similar results can be achieved for other datasets, the clustering is probably very useful to the user when Greedy seeding is used. This cannot be said about the uniformly randomized and K-means++ seeding, as the results tend to be very sensitive to skewed input.

As for the Gap statistic, its usefulness is hard to determine from only these experiments. For Greedy seeding, it is correct for two out of three datasets. When it is wrong, it is wrong only by one. Even if the Gap statistic does not determine $K$ perfectly, perhaps it is useful to the user anyway, because at least it provides a hint in the right direction – especially when $K$ is determined from a majority vote over multiple runs.

When extracting by traversal, a starting node id has to be given. If the user knows nothing about the graph, a reasonable starting node id might be hard to come up with. However, for Neo4j, there is a graph visualization tool called Neoclipse that can be useful in such situations. With Neoclipse, the user gets to see the overall structure of the graph. If some node that is interesting to start traversing from is found, the user can simply note that node's id, and later use this in a Spotfire analysis. Something similar to Neoclipse within Spotfire could be interesting to have for general graphs.

For the Neo4j tool, the following items, in no particular order, are considered most interesting for future work:

- The sampling methods could be further developed. This does not necessarily mean exposing more methods to the user; perhaps it is possible to mix some existing methods into a powerful hybrid.
- A more accurate distance measure could be developed, taking more parameters into consideration, e.g., properties on found edges and types of found node properties.
- Existing seeding methods could be evaluated on more datasets, and perhaps alternative seeding methods could be implemented and compared to the existing ones.
- More experimentation on the Gap statistic and a comparison to other methods for estimating the optimal number of clusters. Depending on how this turns out, perhaps the feature is not useful to the user at all.

## 8.3 Sample and page size parameters

When sampling a database in the implemented tools, sample size has to be given. When data is retrieved, it is paginated. For this, page size has to be given. Naturally, the user's burden should be minimized, so the fewer parameters, the better. Both sample size and page size are parameters that can be discussed whether they should have to be given by the user or if the application should be able to automatically assign reasonable values for them.

There are inherent tradeoffs involved with both. The greater the sample size, the more representative the sample is, but the more time and resources are needed to collect it. For the page size, the tradeoff can be noted from the convex shape of the retrieval time curve presented in Section 7.1.

For the sample size, defining the sweet spot involves subjectivity. There is, though, guidance in the statistics literature to how large a sample size has to be before it is representative. Presenting a sensible value based on statistical theory could be useful to guide the user, but this value could also be different from what the user desires.

In contrast to the sample size, defining the optimal page size is not subjective. It does, though, depend on the specific setting in terms of number of network hops to the servers, network load, and so on. Although these factors cannot easily be taken into consideration dynamically, some reasonable value could be computed and presented as a starting point for the user. For various reasons, though, the user might want to override this.

As a natural conclusion of the discussion above, perhaps it makes most sense to suggest values for both sample size and page size, but that they both can be edited by the user.

## 8.4 Client vs. server execution

In the Cassandra tool, when a database is sampled, the resulting sample is sent over the network to the Spotfire client. The sample analysis is then done client-side. In contrast to this, when a Neo4j instance is sampled in the Neo4j tool, the sample is kept on the server and is also analyzed server-side. Only the resulting schema is sent to the Spotfire client. This illustrates two different architectural approaches.

Naturally, if the server can cope with the extra load of taking care of the entire schema inference procedure, this is advantageous, since less data has to be sent over the network and the Spotfire client avoids the extra load. The only reason the Cassandra tool does not run the schema inference server-side is because there is no built-in support for things like server plugins, like in Neo4j. Notice, though, that for schema inference this is not necessarily a major problem since a schema does not have to be inferred for every import – the schema could be inferred only once (or periodically, e.g., during nights) and then saved in Spotfire somewhere for reuse.

Client vs. server execution is not only relevant when inferring schemas, but in general for functionality that the underlying database does not provide. Another example of this is the filtering that must be performed in the Cassandra tool when there are no indexes (see Section 8.1). Running this filtering server-side instead of client-side could mean substantial performance gains on the overall retrieval time. To enable working with truly large data sets, server, or rather cluster, execution is probably necessary (cf. approaches in Dremel/Hive/Jaql/Pig/Tenzing mentioned in Chapter 2).

## 8.5 Retrofitting

Section 5.1 mentions that one of the effects of using a tabular model in Spotfire is that importing data from tabular sources is straightforward modelwise. Since NoSQL systems generally do not use the relational model, bridging the model differences is not as easy. Of course, this is the general problem in the thesis – but it could be discussed whether the relational model is appropriate at all in applications like Spotfire that wish to support NoSQL data sources. While feasible, retrofitting NoSQL data into a tabular representation requires assumptions and heuristics and certainly does not always end up elegant. Perhaps a less rigid model is more appropriate. On the other hand, at least in Spotfire's case, this would require the entire platform to be redesigned, to adapt to something that might anyway change tomorrow again.

Furthermore, Spotfire is built to have all data in-memory. For most desktop computers and workstations, this limits the size of the dataset to at most a few gigabytes. Many datasets stored in NoSQL systems are much bigger than that, which hints that the in-memory model of Spotfire perhaps is not that well-suited for NoSQL data in general. This problem can, though, be alleviated by setting

up sensible On-Demand analyses. This has the disadvantage that constantly working with subsets of the dataset does not give a good overview.

## 8.6 User level of expertise

Depending on the user's level of expertise and knowledge about the dataset that should be analyzed, different functionality could be exposed. For example, a user who already knows the schema of the dataset does not gain any insight by sampling and subsequently having a schema inferred for it. The only advantage with this approach for such a user is that the schema does not have to be entered manually, and thus potential typing errors are avoided. If all users were like that, less effort should be put on schema inference – and perhaps clustering can be scrapped completely. If users in general are more novice in this respect, schema inference is an important part of the analysis, as the user gets to know the dataset through this procedure.

It would also be possible to expose a simpler text-based interface that lets users programmatically define how to query for data and which structure to map to certain columns. For example, the Spotfire import dialog could provide a text box in which users are given programmatic access to some parts of the underlying data store. In this text box, they can code their own programs extracting data from the NoSQL system. Less effort has then to be put on data model differences, and Spotfire merely has to provide a connection to the underlying data store. This approach is conceptually similar to Tableau's approach mentioned in Chapter 2, except that the query mechanisms of the underlying system can be utilized and that the cluster load gets smaller as less data has to be processed. Naturally, this approach would be targeted at expert users with programming abilities. However, this functionality does not have to be available only to these users. If there is a query language that is sufficiently easy to use, non-programmers can probably also use it. For example, Google reports that their SQL supporting query engine Tenzing has been widely adopted by the non-engineering community within the company for analyzing data [7].

# Chapter 9

# Conclusion

The overall purpose of the thesis has been to investigate how NoSQL databases can be supported in the Spotfire platform (see Section 1.2). As shown by the NoSQL introduction and survey (see Section 3.3 and Chapter 4), data model differences and lack of explicit schemas and a general query interface in NoSQL systems are the reasons for making this a nontrivial problem. However, by using some assumptions and heuristics, the suggested solutions (see Chapter 5) together with the implemented tools (see Chapter 6) showed how this problem and its related issues can be solved.

The solutions are based on first finding a data structure in the underlying NoSQL system that stores a collection of entity instances. Modelwise, these instances are easily mapped to Spotfire's tabular data model. Schemas for such structures can be found by sampling the database, possibly followed by a clustering. As for data extraction, an in-depth analysis of the particular system's query functionality is required, and parts of it can be exposed to the user to enable efficient extraction.

The suggested solutions are not claimed to be complete. Rather, the work in this thesis can serve as a starting point for more thorough investigations or as a basis for something that can be extended.

Other approaches are conceivable too. One suggestion is to only offer the programmatic approach discussed in Section 8.6. Obviously, this has the disadvantage that only experienced programmers can use it. On the other hand, perhaps it is reasonable to require that an organization using Spotfire has at least one experienced programmer who can set up ad hoc analyses – or TIBCO Software could offer this as a consulting service.

As briefly discussed in Section 8.5, a completely different approach would be to redesign Spotfire to support a less rigid model, allowing for, e.g., complex data. In this way, many of the model differences between NoSQL systems and Spotfire could be avoided. However, because there is no general NoSQL query interface, extracting data would still have to be tailored for each supported system. Also, changing the data model in Spotfire likely requires a substantial amount of work. Naturally, the final choice of how, or even if, to support NoSQL databases in Spotfire depends not only on technical details, but also on business considerations and what customers ask for.

Finally, it is concluded that if support for another NoSQL system shall be developed, this will require its own tailored solution. The designs of the two

implemented tools support this claim. Most likely, not even systems within the same NoSQL family can be handled generically. At least, however, the overall approach used in the thesis can be reused.

# Bibliography


[1] *A case study: Design and implementation of a simple Twitter clone using only the Redis key-value store as database and PHP*. Accessed January 25, 2012. URL: http://redis.io/topics/twitter-clone.

[2] D. Abadi. *Distinguishing Two Major Types of Column-Stores*. Accessed January 25, 2012. Mar. 2010. URL: http://dbmsmusings.blogspot.com/2010/03/distinguishing-two-major-types-of_29.html.

[3] D. Arthur and S. Vassilvitskii. "k-means++: The Advantages of Careful Seeding". In: *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 2007, pp. 1027–1035.

[4] K.S. Beyer et al. "Jaql: A Scripting Language for Large Scale Semistructured Data Analysis". In: *Proceedings of the VLDB Endowment* 4.12 (2011).

[5] E. Brewer. *Towards Robust Distributed Systems*. Accessed January 25, 2012. July 2000. URL: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[6] F. Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26.

[7] B. Chattopadhyay et al. "Tenzing – A SQL Implementation On The MapReduce Framework". In: *Proceedings of the VLDB Endowment* 4.12 (2011), pp. 1318–1327.

[8] E.F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[9] *comScore Reports Global Search Market Growth of 46 Percent in 2009*. Accessed January 25, 2012. Jan. 2010. URL: http://www.comscore.com/Press_Events/Press_Releases/2010/1/Global_Search_Market_Grows_46_Percent_in_2009.

[10] B.F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM. 2010, pp. 143–154.

[11] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[12] G. DeCandia et al. “Dynamo: Amazon’s Highly Available Key-value Store”. In: *ACM SIGOPS Operating Systems Review* 41.6 (2007), pp. 205–220.

[13] S. Dudoit and J. Fridlyand. “A prediction-based resampling method for estimating the number of clusters in a dataset”. In: *Genome biology* 3.7 (2002), research0036.

[14] E. Evans. *NoSQL: What’s in a name?* Accessed January 25, 2012. Oct. 2009. URL: `http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html`.

[15] R.T. Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. University of California, 2000.

[16] E. Fields. *Tableau 6.1 Now Supports Hadoop*. Accessed January 25, 2012. Nov. 2011. URL: `http://www.tableausoftware.com/about/blog/2011/10/tableau-61-now-supports-hadoop-13921`.

[17] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Vol. 174. Freeman San Francisco, CA, 1979.

[18] R. Giudici. *Disney Central Services Storage: Leveraging Knowledge and skillsets*. Accessed January 25, 2012. May 2011. URL: `http://www.10gen.com/presentations/mongosf2011/disney`.

[19] M.T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc., 2006.

[20] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2009.

[21] E. Hewitt. *Cassandra: The Definitive Guide*. O’Reilly Media, Inc., 2010.

[22] H. Heymann. *Practical Data Storage: MongoDB at foursquare*. Accessed January 25, 2012. June 2011. URL: `http://www.10gen.com/presentations/mongonyc-2011/foursquare`.

[23] McKinsey Global Institute. *Big data: The next frontier for innovation, competition, and productivity*. May 2011.

[24] Y. Izrailevsky. *NoSQL at Netflix*. Accessed January 25, 2012. Jan. 2011. URL: `http://techblog.netflix.com/2011/01/nosql-at-netflix.html`.

[25] J. Jackson. *CouchBase, SQLite launch unified NoSQL query language*. Accessed January 25, 2012. July 2011. URL: `http://www.arnnet.com.au/article/395469/couchbase_sqlite_launch_unified_nosql_query_language`.

[26] A.K. Jain and R.C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[27] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Vol. 39. Wiley Online Library, 1990.

[28] D. King. *She who entangles men*. Accessed January 25, 2012. Mar. 2010. URL: `http://blog.reddit.com/2010/03/she-who-entangles-men.html`.

[29] R. King. *Cassandra at Twitter Today*. Accessed January 25, 2012. July 2010. URL: http://engineering.twitter.com/2010/07/cassandra-at-twitter-today.html.

[30] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[31] C.D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Vol. 1. Cambridge University Press Cambridge, 2008.

[32] S. Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 330–339.

[33] D. Merriman. *SQL to Mongo Mapping Chart*. Accessed January 25, 2012. Sept. 2011. URL: http://www.mongodb.org/display/DOCS/SQL+to+Mongo+Mapping+Chart.

[34] K. Montrose. *Does StackOverflow use caching and if so, how?* Accessed January 25, 2012. Nov. 2010. URL: http://meta.stackoverflow.com/questions/69164/does-stackoverflow-use-caching-and-if-so-how/69172.

[35] *Neo4j Projects*. Accessed January 25, 2012. URL: http://www.delicious.com/neo4j/projects.

[36] C. Olston et al. "Pig Latin: A Not-So-Foreign Language For Data Processing". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM. 2008, pp. 1099–1110.

[37] T. Preston-Werner. *How We Made GitHub Fast*. Accessed January 25, 2012. Oct. 2009. URL: https://github.com/blog/530-how-we-made-github-fast.

[38] *Redis Clients*. Accessed January 25, 2012. URL: http://redis.io/clients.

[39] B. Scofield. *NoSQL – Death to Relational Databases(?)* Accessed January 25, 2012. Jan. 2010. URL: http://www.slideshare.net/bscofield/nosql-codemash-2010.

[40] M. Slee, A. Agarwal, and M. Kwiatkowski. *Thrift: Scalable Cross-Language Services Implementation*. Accessed January 25, 2012. Apr. 2007. URL: http://thrift.apache.org/static/thrift-20070401.pdf.

[41] M. Stonebraker et al. "The End of an Architectural Era (It's Time for a Complete Rewrite)". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment. 2007, pp. 1150–1160.

[42] C. Strauch. *NoSQL Databases*. Accessed January 25, 2012. Feb. 2011. URL: http://www.christof-strauch.de/nosqldbs.pdf.

[43] C.A. Sugar and G.M. James. "Finding the number of clusters in a dataset: An information theoretic approach". In: *Journal of the American Statistical Association* 98.463 (2003), pp. 750–763.

[44] *Third Party Support*. Accessed January 25, 2012. Nov. 2011. URL: http://wiki.apache.org/cassandra/ThirdPartySupport.

[45] A. Thusoo et al. “Hive – A Petabyte Scale Data Warehouse Using Hadoop”. In: *2010 IEEE 26th International Conference on Data Engineering (ICDE)*. IEEE. 2010, pp. 996–1005.

[46] R. Tibshirani, G. Walther, and T. Hastie. “Estimating the number of clusters in a data set via the gap statistic”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63.2 (2001), pp. 411–423.

[47] J.D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2008.

[48] I.T. Varley. “No Relation: The Mixed Blessings of Non-Relational Databases”. MA thesis. The University of Texas at Austin, 2009.

[49] N. Walker and N. Caudill. *Talk: Real-time Updates on the Cheap for Fun and Profit*. Accessed January 25, 2012. Oct. 2011. URL: `http://code.flickr.com/blog/2011/10/11/talk-real-time-updates-on-the-cheap-for-fun-and-profit`.

[50] P. Willett, J.M. Barnard, and G.M. Downs. “Chemical Similarity Searching”. In: *Journal of Chemical Information and Computer Sciences* 38.6 (1998), pp. 983–996.

[51] J. Zawodny. *Lessons Learned from Migrating 2+ Billion Documents at Craigslist*. Accessed January 25, 2012. May 2011. URL: `http://www.10gen.com/presentations/mongosf2011/craigslist`.