Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis No. 3460

Extending a Methodology for Migration of the Database Layer to the Cloud Considering Relational Database Schema Migration to NoSQL

Rilinda Lamllari

Course of Study: Master Informatik

Examiner: Prof. Dr. Frank Leymann
Supervisor: Steve Strauch
Commenced: January 28, 2013
Completed: June 27, 2013

CR-Classification: C.2.4, D.2.11, D.2.12, E.2, H.2.0, H.2.1, H.2.4, H.2.5, H.4.2

Abstract

The advances in Cloud computing and in modern Web applications have raised the need for highly available and scalable distributed databases to accommodate the big data being created and consumed. Along with the explosion in data growth comes the necessity to rapidly evolve databases and schemas to meet user demands for new functionality. A special attention is being paid to the vast amounts of semi-structured and un-structured data, and the data management tools should reflect the support for these needs. This has lead to the development of new Cloud serving systems such as 'Not Only' SQL (NoSQL) databases.
NoSQL databases were driven by the scalability needs of the big companies, such as Google, Facebook, Amazon, and Yahoo. While the demands of these key players are different from those of small and medium enterprises in terms of scalability, the core problem is the same — storage arrays are not scalable and force you into expensive, forklift upgrades.
These facts combined with changes in how IT resources are delivered and consumed through the Cloud computing paradigm, projects adopting NoSQL solutions are not a hype anymore. NoSQL databases are being offered as a service by the big Cloud providers, such as Google, Amazon, Microsoft, but by smaller vendors as well. In this master thesis we investigate the possibilities and limitations of mapping relational database schemas to NoSQL schemas when migrating the database layer to the Cloud. Based on literature research we provide recommendations and guidelines with regard to schema transformation and discuss the implications at other application architecture layers, such as business logic and data access layer.
We extend an existing data migration tool and methodology for incorporating the migration guidelines and hints. Moreover, we validate our work based on a chosen sub-set of relational and NoSQL databases by using example data from the established TPC-H benchmark.

Contents

List of Figures

List of Tables

1 Introduction

With the advent of Web 2.0 sites where millions of users may both read and write data, scalability for simple database operations has become more important. The increasing volume and detail of information captured by enterprises and the rise of multimedia, as well as the social media is putting companies in front of large datasets. Examples of these large datasets are links, social networks, activity in Web or transactions logs, manufacturing data from assembly line devices, scientific data collections, etc. Websites have started tracking activity in a very detailed way in order to have more accurate and detailed performance information to deal with the performance requirements posed by their customers.
Big data, a very controversial name for large datasets, according to IBM spans three basic dimensions[1]: velocity, variety, and volume.

- Volume of data of different types is overgrowing in terabytes and even petabytes.
- Variety depicts the structured level of data: un-structured data (such as documents, e-mail, multimedia and social media), semi-structured data (CSV files, log files, form, reports), and structured data [IBM].
- Velocity is the speed with which these data are produced (e.g. real time data from social networks) and the speed with which these data should be processed (e.g. for time-sensitive processes such as fraud detection).

Companies are being faced with big data [Ins] and they are exploring ways on how to unlock and create value out of these data, being innovative and gaining competitive advantage. According to Gartner [Sta], in 2013, big data is forecast to drive $34 billion of IT spending which shows the worldwide interest in this explosion. Even though there is no universal agreement on what big data is, in this thesis we use it in the context of "data that exceed the processing capacity of traditional database systems". According to EMC[2], big data is more characterized by a scale-out architectural style then by specific data sizes or processing speeds [Gro].
Another definition given by National Institute of Standards and Technology (NIST)[3]: "where the data volume, acquisition velocity, or data representation limits the ability to perform effective analysis using traditional relational approaches or requires the use of significant horizontal scaling for efficient processing" [CM]. Relational Database Management System (RDBMS) will be challenged to scale up or out to meet the demands posed by this exponential data growth [CM],[Gro].
While big data often gets associated with data analytics, in particular with Hadoop[4], actually

[1]What is big data? http://www-01.ibm.com/software/data/bigdata/
[2]EMC: http://www.emc.com/index.htm
[3]US Agency of Department of Commerce which is responsible for developing standards and guidelines
[4]Apache Hadoop: http://hadoop.apache.org/

it is much broader than that, and is far more concerned with data storage than analytics. Technologies like Apache Hadoop are concerned with large scale data analysis and have been widely recognized and studied, while NoSQL systems are developed for Cloud data serving [CST$^+$10]. Apache Hadoop, and its underlying file system Hadoop Distributed File System (HDFS)[5], store the bulk of big data today, the need was for a more structured way to manage and expose data, stored in Hadoop or not – without the strong constraints that relational databases impose. Thus, laid the groundwork for the rapid growth of NoSQL systems.

NoSQL technologies, most commonly referred to as 'Not Only SQL', is a term that describes a broad class of technologies that provide a different approach to data storage compared with traditional RDBMSs. Some refer to them as 'No relational' or 'No RDBMS', some simple prefer to call them Distributed Database Management Systems. Whatever the literal meaning, NoSQL is used today as an umbrella term for all databases and data stores that don't follow the traditional and well-established RDBMS principles and often relate to large data sets accessed and manipulated on a Web scale [Tiw11]. They feature elasticity and scalability in combination with a capability to store big data and work with Cloud computing systems, all of which make them extremely popular at the time being. The reality is that one size doesn't fit it all. NoSQL databases are targeted at a particular set of applications scenarios; they will not and cannot replace RDBMSs, they can address certain limitations and are a complement to deal with issues such as complexity, performance, and scalability.

Looking at the interest shown for NoSQL databases, some RDBMSs have started introducing NoSQL features into their products. For example, with the release of SQL Server 2008, a number of features were added to the product under the moniker "beyond relational" to store large binary objects in the file system with full transactional integrity [Lob]. It introduced a set of features that can be classified as NoSQL-like in nature. Oracle announced the introduction of a NoSQL-type interface into its MySQL Cluster product that is key/value in design. With SQL and NoSQL access through a new Memcached API, MySQL Cluster represents a "best of both worlds" solution allowing key value operations and complex SQL queries within the same database [Orab]. Also, PostgreSQL now supports schema-less (document key-value store) data as well, without losing the traditional database features, the *hstore* data type is used for storing sets of key/value pairs within a single PostgreSQL value [Posb].
Not only the industry, with the most popular examples of [Ale], [Ana], [Pat], but also the research domain is showing a big interest in exploring NoSQL technologies. EPIC, a large research project on how do people use social media in times of crisis, has transitioned a large-scale data collection infrastructure from the relational database MySQL to a hybrid persistence architecture that makes use of both relational and NoSQL technologies [SA12].

1.1 Motivating Scenarios

In an era of high competition, having 'always-on' capabilities is fundamental for established and new startup companies to survive. Being close to the customers and understanding their

[5]HDFS is a distributed file system designed to run on commodity hardware

preferences is the motto of today's companies. According to McKensey survey [Ins], a retailer could increase its operating margin by more than 60 percent using big data to the full. And the adoption of NoSQL solutions by retailers seem to prove that.

DynamoDB[6], or simply Dynamo, the database offering by Amazon is the result of 15 years of learning in the areas of large scale non-relational databases and Cloud services. It is the system being used by Amazon e-commerce itself. Riak database[7] - a Dynamo clone with commercial support by Basho - is based on Amazon's architectural principles and has been adopted successfully by retail and e-commerce platforms [Basc], offering architectural, operational and development benefits for the retailers. Top retailers in US are using Riak for applications such as shopping carts, product catalogs, API platforms, and mobile applications.

As retailers grow and have to store more and more data, traditional relational databases aren't always the best option. They want to scale easily, without the operational burden of manual partitioning of data. Meanwhile, business requirements demand their data is always available for reads and writes. For them failure to accept additions to a shopping cart, or serve product information quickly, has a direct and negative impact on revenue. Many retailers now operate online with an API or data services platform. They included Riak as an integral part in their transformation push to re-platform the eCommerce platform as it offers high availability and scale, so retailers can always serve customers, even under failure conditions, and rapidly grow to meet peak loads.

Netflix,.inc [Ana], a movie and TV shows streaming provider, migrated from Oracle to Amazon Web Services (AWS) SimpleDB[8] and Simple Storage Service (S3)[9] because AWS promised better availability and scalability in a relatively short amount of time. Also, Alexa.com [Ale] which offers web-scale services built on top of AWS, chose Amazon SimpleDB over MySQL to store intermediate status and log data, and Amazon S3 for put and retrieves datasets.

Taking these use cases into consideration, we see that consumer facing applications benefit the most from NoSQL solutions as they are more concerned about low latency and high availability. This is the case for Amazon-like scenarios, where dealing with a huge traffic and where availability is crucial. But for the transaction data in a normal scenario you might not want to sacrifice the reliability of the RDBMS when dealing with transactions. This is the reason why e-commerce has often been a domain exclusive to RDBMSs. Another reason is that domains that require rich data models and sophisticated queries have been assumed to fit best within the realm of the RDBMS [Ban11]. Instead, product catalogue and shopping cart might be ideal use cases for migrating to NoSQL. Given these facts, we will migrate partially a relational database schema for an e-commerce into the NoSQL datastores. We use TPC-H benchmark as basis for example database schema, because it is business oriented and is targeting systems examining large volumes of data and executing queries with a high degree of complexity [Tra].

[6]Amazon DynamoDB: http://aws.amazon.com/dynamodb/
[7]Riak: http://basho.com/riak/
[8]SimpleDB: http://aws.amazon.com/simpledb/
[9]S3: http://aws.amazon.com/s3/

1.2 Problem Statement

RDBMS technology can no longer keep pace with the velocity, volume, and variety of data being created and consumed. These systems are based on relations and data normalization. Accommodating new big data into these systems causes the relational normalized schema to become too complex. Normalizing data requires more tables, which requires more table joins, thus requiring more keys and indexes. As databases size starts to grow into terabytes, performance starts to significantly degrade. The main reason why Cloud computing is being utilized by many organizations is to avoid high upfront costs and to avoid dealing with issues such as scalability in which Cloud providers have an extensive experience. The decision to start using NoSQL databases comprises a set of trade-offs, better said it's all about trade-offs. Migrating to the Cloud NoSQL databases, requires some preparation for the enterprise IT that grew up with RDBMSs. The transition from legacy RDBMS to NoSQL requires careful planning. It is a completely different mindset needed by the developers and architects in order to get the most out of this new technology. Distributing data across multiple nodes in a data center results in a linear performance improvement. However, understanding how to store data in ways that allow scalability requires a rethinking of your overall approach to systems design. Thus, designing the tailored data model for your application, and not being only driven by the simple APIs these databases offer, should be at the core of the planning. To make the right decision when choosing the NoSQL store for your business problem, requires a very good low-level understanding of the target store. The plethora of NoSQL products with many different feature sets, makes it almost impossible to recommend only by the data model that they offer. Moreover, how will this transition impact the other application layers is another very important aspect of this process. Therefore, we investigate the potential and limitations when mapping relational schemas to NoSQL with the goal of providing guideline and support.

1.3 Scope of Work

This master thesis has the goal to specify, design, and implement a methodology for migration of the relational database schemas to a NoSQL database schema taking into consideration lessons learned from companies that have already migrated their applications and from the literature review. The relational databases we consider for the the validation of our approach are MySQL version 5.6 and PostgreSQL version 9.2. There are four basic categories of NoSQL databases: key-value stores, document, columnar, and graph databases. This classification is arguable, and many prefer to classify also Object Oriented Database Management Systems (OODBMS) and XML databases as NoSQL systems. In this thesis we focus on the three typical data models, key-value, document, and columnar databases. The graph databases are meant for a specific usage, i.e. highly interconnected data in social networking applications, this is the reason this is left out of scope of this work. In this thesis we propose and implement the extension of a multiple criteria decision support system for application migration from RDBMS to NoSQL databases in the Cloud. Users will obtain hints and recommendations on selecting the appropriate database based on their functional and non-functional properties.

Even though the right tool can be found by starting using and testing it, we provide a basis for the relevant features to be considered before migration. We leave outside of scope the efforts on estimating all the factors before migration including the pricing aspects of the solution and the scalability costs. The focus of this master's thesis is the schema transformation. The main goal is to investigate the possibilities, limitations, and shortcomings when migrating from relational to NoSQL databases. The efforts involved in this transition with respect to changes at the application layer and the query transformation are not part of this work. Although for these two aspects where appropriate we provide hints and guidelines when the schema change has direct impact on them.

1.4 Research Questions

In the following we refine the major goal of investigating the potential and limitations when migrating relational schemas to NoSQL in order to provide guidelines and support into concrete research questions that are addressed in this thesis:

- What are the characteristics of NoSQL databases, and what are the characteristics of the applications making use of this new technology?
- What relational schema elements can be migrated to NoSQL? Which ones can be mapped directly, which cannot, which can be emulated differently?
- What are the trade-offs involved when choosing a NoSQL database, i.e. what trade-offs make these systems in order to support specific application needs? What are the trade-offs the application owner itself has to make related to: transactional behavior, complex and ad-hoc queries, data integrity, data types, etc.? And for those features that can't be traded off, how can they (if possible) be emulated differently?
- What is the impact on other application layers, i.e. data acess layer and business logic layer, in case of missing functionality or different non-functional properties of the NoSQL target datastore, e.g. consistency, ACID transactions, etc.?
- How to support the decision process, migration, and refactoring when migrating from a relational database to a NoSQL store?

1.5 Research Design

For the first two research questions we performed a systematic review with the focus on identifying and appraising the functional and non-functional properties of the NoSQL databases. Based on the different application migration use cases of the database layer to NoSQL in the Cloud, we abstracted the state of the art on use cases including types of applications. We performed a literature research on NoSQL stores and the different categories that fall under this umbrella from online resources, official websites, books and different industry reports. After the analysis, we abstracted the features to include in the methodology and

the questionnaire in order to support the decision process. Based on the analysis we also provide answers for the other research questions. Based on provider offerings we performed an analysis on differences of relational and NoSQL and NoSQL categories, and the results were incorporated to the Cloud Data Hosting Solution taxonomy.
Based on literature analysis and provider offerings, the outcome about application use cases for NoSQL is derived. From best practices, use cases, experience reports from industry and analysis of current state of tool, we model the extension of the methodology and the requirements for prototypical extension. For the validation of our work we use example data from established benchmark.

1.6 Outline

The following six chapters cover different phases of work that have been pursued to accomplish the given goals and to conceive future tasks on the topic.

- **Fundamentals, Chapter 2**–At the beginning, relevant literature that covers the fundamentals of this master thesis was surveyed. The chapter gives an explanation of NoSQL database categories, Data modeling, CAP Theorem, Software-as-a-Service (SaaS).

- **Related Work, Chapter 3**–This chapter presents the related work in the following domains relevant for this thesis: application scenarios suitable for using NoSQL databases, data modeling considerations and best practices when migrating from relational to NoSQL schemas based on the industry reports of successful use cases migration.

- **Analysis, Concept and Specification, Chapter 4**–an analysis of relational modeling and non-relational modeling, followed by an analysis of two relational databases and three NoSQL stores. Lessons learned in the first chapters are considered when formalizing the functional and non-functional requirements for the system. This includes a conceptual overview, a set of best and worst modeling practices, a decision support system analysis, and a taxonomy extension.

- **Design, Chapter 5**–An architectural overview devises components and their relations that together fulfill the described system requirements. The design of the extended architecture of the system developed by Bachman is shown. All the requirements abstracted from the analysis are mapped into categories of the questionnaire.

- **Implementation, Chapter 6**– In this chapter we describe the tools, libraries, etc. used for the implementation of the extensions.

- **Outcome and Future Work, Chapter 7**–This last chapter summarizes the outcomes of this work, the findings and suggests future extensions to the system.

1.7 Definitions and Conventions

The following definitions and abbreviations should be inspected for understanding the descriptions in this work. They are used throughout the document.

Definitions

List of Abbreviations

The following list contains abbreviations used in this document.

BSON Binary JavaScript Object Notation

CAP Consistency Availability Partition tolerance

CLI Command Line Interface

CQL Cassandra Query Language

DAL Data Access Layer

DBMS Database Management System

DDD Domain Driven Design

DDL Data Definition Language

ETL Extract Transform Load

IaaS Infrastructure-as-a-Service

JSON JavaScript Object Notation

LDBC Liberty Database Connectivity

LOB Line of Business

NoSQL 'Not Only' SQL

OLTP Online Transaction Processing

OODBMS Object Oriented Database Management Systems

ORM Object-Relational Mapping

PaaS Platform-as-a-Service

RDBMS Relational Database Management System

SaaS Software-as-a-Service

SLA Service Level Agreement

SQL Structured Query Language

UnQL	Unstructured Query Language
UUID	Universally Unique Identifier
YCSB	Yahoo Cloud Serving Benchmark

2 Fundamentals

This master thesis relies on distinct conceptual and technological fundamentals that are clarified in this chapter. Together they form the context for the outcomes of this work. As the goal is to investigate the possibilities and limitations of mapping relational schemas to NoSQL schemas when moving the database layer to the Cloud, initially an introduction to Cloud computing is given in Section 2.1. We provide an overview on current trends in the database market in Section 2.2, a generic overview of non-functional properties of NoSQL data systems is given in Section 2.3, followed by a summary of NoSQL categories in Section 2.4.

2.1 Cloud Computing

The idea behind Cloud computing is to offer the computational power on demand in the same way the public utility services like electricity, water, gas, and telephony are provided. We as consumers are not aware of the whole infrastructure behind the electrical wall socket which hides the power generation stations and a huge distribution grid [VBB11]. Cloud computing is an evolving paradigm. It is the result of the convergence of a set of technologies which were considered as hype in their early stages. Because of specification and standardization processes, these technologies matured and form the foundation for Cloud computing. Such advancement in technologies include [VBB11]:

- Hardware: Virtualization, Multi-core chips
- Internet technologies: Web Services, Service-Oriented Architectures, Web 2.0, Mashups
- Distributed computing : Utility and Grid Computing (clusters and grids)
- Systems management : Autonomic Computing, Data Center Automation

NIST defines Cloud computing as "...a pay-per-use model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [MG09]. There are five essential characteristics for the Cloud model deriving from this definition. Firstly, the customer can provision computing resources without further intervention of the service provider. Moreover, the computing resources are accessible by various types of client platforms via standard protocols. In addition to raw computing and storage, Cloud computing providers usually offer a broad range of software services. They also include APIs and development tools that allow developers to build scalable applications upon their services. Below are given three basic service models that compose a Cloud model: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

- **Software-as-a-Service (SaaS)** is a service model where service providers offer applications running on a Cloud infrastructure to their customers. The underlying Cloud infrastructure enables Cloud characteristics like elasticity and accessibility. In this model the customer can only control individual configurations that are applied while the customer uses the application.

- **Platform-as-a-Service (PaaS)** allows the customer to deploy and configure his own artifacts to the Cloud infrastructure, such as applications, services, or libraries. These artifacts are defined using programming languages, description languages, or modeling languages predetermined by the service provider.

- **Infrastructure-as-a-Service (IaaS)** is the most flexible service model providing the customer with control at the operating systems and virtual machines level. As a consequence, the customer can deploy any software that runs on the available operating systems.

There are many other flavors of these basic models, such as Database as a Service (DBaaS) which is a form of PaaS, Integration as a Service which can be a form of PaaS or SaaS, etc. The DBaaS is currently found in the public marketplace in three broad capabilities [PC]:

- Online relational databases such as Amazon RDS[1] (MySQL offering), Microsoft SQL Azure (SQL-server like product)[2].

- Non-relational databases or NoSQL databases: Google AppEngine Datastore, Amazon SimpleDB and DynamoDB, Microsoft Azure Table storage, etc. Apart of the big Cloud providers, many other companies are offering NoSQL as a Service, such as Datastax, MongoHQ, Cloudera, Cloudant, Basho, etc.

- The ability to operate virtual machine images loaded with common open source databases such as MySQL or similar commercial databases.

Regarding the ownership of the physical resources, four deployment models can be distinguished: A *Private Cloud* is used exclusively by one organization, possibly comprising many divisions. A *Community Cloud* is shared by multiple organizations. Both deployment models do not pretend the Cloud infrastructure to be off premise. On the contrary, a *Public Cloud* is not restricted to a group of organizations, and therefore, must be provided by a separate service provider. A *Hybrid Cloud* combines at least two different of the previous deployment models [MG09]. This approach allows an organization to use a public Cloud for its Business Continuity Plan (BCP), if its own computing capabilities can not handle with the current workload.

[1]Amazon Relational Database Service: http://aws.amazon.com/rds/

[2]Introducing Windows Azure SQL Database: http://msdn.microsoft.com/en-us/library/windowsazure/ee336230.aspx

2.1.1 Migration to the Cloud

Organizations migrate to the Cloud primarily for business and economic reasons, but also for technological reasons. The economic reasons driving the migration to the Cloud are the cost reduction of both IT capital expenditures (CAPEX) as well as the operational expenses (OPEX) [VBB11]. Many applications take advantage of storing data in the Cloud to build different kinds of clients — Web, mobile, desktop — that access the same data. The migration can happen at one of the five levels of application, code, design, architecture, and usage. Combined with the Cloud service models - IaaS, PaaS or SaaS model - it results in many migration use cases. Applications are typically built using a three layer architecture pattern consisting of a presentation layer, a business logic layer, and a data layer (Fowler et al., 2002). The presentation layer is responsible for application-users interactions, the business layer realizes the business logic and the data layer is responsible for application data storage. The data layer is in turn subdivided into the Data Access Layer (DAL) and the Database Layer (DBL). The DAL encapsulates the data access functionality, while the DBL is responsible for data persistence and data manipulation. In order to take advantage of Cloud computing, the applications either are designed to be run in the Cloud (Cloud-native application) or moved to the Cloud (Cloud-enabling them) [SABL13]. Each application layer can be hosted in different Cloud deployment models. Typically migration projects into the Cloud are implemented using a phased approach. T.S.Mohan et al. propose a generic iterative process based on a seven-step model for migration into a Cloud environment [VBB11]. By encapsulation your application functionalities into services, it supports changing the data storage technologies as the needs and technology evolve. Separating parts of applications into services also allows you to introduce NoSQL technologies into an existing application [SF12].Decisions to migrate existing systems to public IaaS clouds can be complicated as evaluating the benefits, risks and costs of using Cloud computing is not a straightforward process. In this thesis we do not consider such decisions, we assume that the decision to migrate to the Cloud has already been taken.

2.2 SQL, NoSQL and NewSQL

Different DBMSs are emerging to address the different applications requirements that came with the changes of the Web. Large enterprises have many Online Transaction Processing (OLTP) systems for different transactions such as order, withdraw money, cash a check, etc [Mica]. Extract Transform Load (ETL) products are used to extract data from these systems and put them in a common format in a data warehouse for business analysis. These activities were supported in large by traditional RDBMSs. The new Web-based workloads (new OLTP), such as online games, location based services, social networking, etc., pose new requirements for higher throughput and greater schema flexibility. This has led to the emergence of new database management systems like NoSQL and NewSQL. These new emerging workloads of modern Web applications are defined as *serving workload* and the systems supporting them as Cloud *serving* systems. Such systems support online read and write access to data [CST$^+$10].

2.2.1 RDBMS and SQL

A RDBMS is a Database Management System (DBMS) that is based on the relational model introduced by Edgar F. Codd in 1969. The relational model is solidly based on two parts of mathematics: first-order predicate logic and the theory of relations. A mathematical relation is a set with special properties. All of the relation's elements are tuples, all of the same type and unordered. Relation is the mathematical term for a table. Each row represents a tuple of the relation, see figure 2.1. The ordering of rows is immaterial and all rows are distinct from one another in content. There are twelve rules of the relational model that were adopted by many open source and commercial DBMS implementations. The relational model provides precise specification for what should be built by a DBMS, not how should it be built. He defined twelve rules for relational management system, and rule number five says: "The DBMS must support a clearly defined language that includes functionality for data definition, data manipulation, data integrity, and database transaction control" [Cod90]. The following

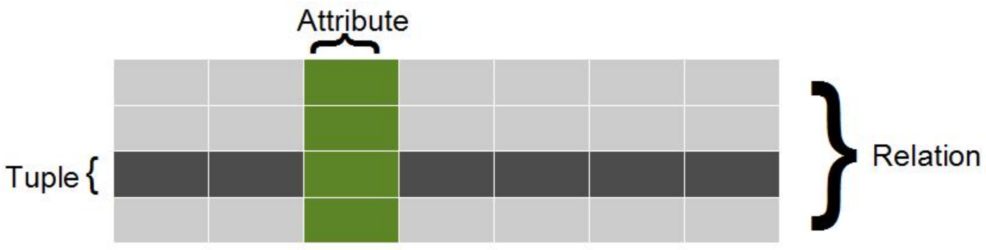


Figure 2.1: Codd's Relational Model

are the most important data integrity features in the relational model and include primary key, foreign key and constraints. Together with relations (tables), tuples (rows) and attributes (columns) they form the basic blocks of the relational model.
A primary key identifies uniquely a row and may consist of a simple column or a combination of columns. When it consists of a combination of columns, the key is said to be *composite*. A composite primary key enables you to specify two attributes in a table that collectively form a unique primary index. Each column participating in a composite primary key may be a foreign key, but not necessarily [Cod90].

Codd defines the two following types of integrity:
Entity integrity: the Primary key of a table must contain a unique, non-null value for each row.
Referential integrity: if the Foreign key contains a value, that value must refer to an existing row in the parent table. You want to make sure that no one can insert rows in the Orders table that do not have a matching entry in the Customer table. This is called maintaining the referential integrity of your data.
Integrity constraints imposed by the relational model that are supported by RDBMSs are:

- *Uniqueness* constraints to ensure that a given column is unique
- *Foreign key* or referential constraints to ensure that two keys share a primary key to foreign key relationship

- *Not Null* constraint to ensure that no null values are allowed
- *Check* constraint: defines a business rule on a column, enforce domain integrity by limiting the values that are accepted by one or more columns (for example, a customer's *age* should always be > 18)

Indexing - a mechanism for providing faster access to data. Indexes are special data structure to optimize the query for a specific set of columns (a.k.a performance-oriented objects). They are built to avoid full table scans during queries. They can be created on any combination of attributes on a relation.

Joins are read operations that take two separate tables and combining them in some way to return a single table.

The relational DBMSs use Structured Query Language (SQL) as the industry standard language for implementing the functionalities of Codd's rules and for querying the data. American National Standards Institute (ANSI) passed the SQL as a standard in 1986. The International Standards Organization (ISO) adopted it in 1987. It has seen many changes since the first time it was published as a standard because most commercial vendors offered the features that were missing in it. The 1989 standards were more complete, but still left many important elements undefined. SQL is also interchangeably used to refer to databases that use SQL as their query language.

RDBMS ensure the transaction control via the well known ACID guarantees. The ACID model is one of the oldest and most important concepts of database theory. In the context of transaction processing it refers to the four following properties. They constitute the basic building blocks of any database transaction model. All four properties have to be met in order for database transactions to be processed reliably.

- Atomicity - an operation is atomic if it either completes entirely or not at all
- Consistency - all operations will bring the database from one consistent state to another, i.e. they will preserve the database rules.
- Isolation - ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other.
- Durability - once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

RDBMSs are in the market since three decades and are being offered by the big companies such as Oracle, IBM, and Sybase, Microsoft and open source software products such as MySQL and PostgreSQL. They generally guarantee ACID properties for transaction-oriented applications, i.e. OLTP. They are mature and offer many advanced features such as transactions and logging to ensure data integrity and query optimization features. These SQL databases work fine as long as the data comes in at a reasonable speed and is of a normal variety. The problem is that being relational and guaranteeing ACID is not necessary for some use cases and can add unnecessary overhead, which popular heavily trafficked website don't want. We will describe such use cases later in this work.

PostgreSQL[3] is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX, and Windows. It is fully ACID compliant and it has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). Its SQL implementation strongly conforms to the ANSI-SQL:2008 standard. There are active PostgreSQL systems in production environments that manage in excess of 4 terabytes of data [Posa].

MySQL[4] is the most used open source RDBMS. Since April 2009, and since version 5.1, MySQL is in two different variants: the open source MySQL Community Server and the commercial Enterprise Server. MySQL is modular in design, provides pluggable storage engines, and allows pluggable modules to support additional features as desired. It offers ACID compliance when using transaction capable storage engines (InnoDB and Cluster).

2.2.2 NoSQL Stores

The NoSQL movement began as an effort to accommodate the big data phenomenon. The Dynamo technology developed at Amazon [DHJ+07] and the BigTable distributed storage system developed at Google [CDG+08], have inspired many of today's NoSQL applications and databases. The motivation behind BigTable was the need to store results from the WebCrawlers which extracted HTML pages, images, sounds, videos and other media from the Internet. The resulting data set was so large that it could not fit into a single relational database, so Google built their own storage system.
There are about 150 NoSQL databases [NoS] in the world today, and new ones are being worked on. This variety makes it difficult to select the best tool for a particular case. The plethora of NoSQL databases can be grouped into the four basic categories based on the data models that they use: key-value, document, columnar and graph databases which will be explained in the next section.
Driven by people who thought that tabular structure is too limiting, NoSQL databases are not a hype anymore but are being adopted by many applications. The most commonly accepted interpretation of NoSQL is 'Not Only SQL' and thus conveying the belief that NoSQL databases will not replace the relational databases, they will just be part of a hybrid architecture addressing needs such as scalability, high-performance, and high availability. Rather than conforming to SQL standards and providing relational data modeling, NoSQL databases have different design points compared to RDBMSs. They typically offer fewer transactional guarantees in exchange for greater flexibility and scalability [PC]. NoSQL movement is seen by the old-school database world as an heretical movement because historically, databases almost always have tried to implement the relational model and be fully ACID-compliant. For the NoSQL proponents, for the top-tier Web sites it is important to enable massive scalability, low latency, and an easier programming model.

One of the great benefits of having a rich SQL implementation is that you hook into all of the development, reporting, ETL and backup tools, etc., that are used in the enterprise. Old

[3]PostgreSQL: http://www.postgresql.org/
[4]MySQL: http://www.mysql.com/

school Database Administrators (DBA) and relational system designers believe that NoSQL supporters are going to stumble over the problems the veterans have already fixed [Moh13]. One of these arguments is about the querying features.

The programmer has limited control over the execution of the SQL statements; it is the database engine that takes care of the statements execution optimization. A *query optimizer* which is a very important architectural component of RDBMSs takes care of the execution by deciding which query plans to execute to most quickly answer a query. The query optimizers can leverage all the information and statistics that are available about the various tables and allow the developers to focus on the "what", in NoSQL world the query optimization responsibility is shifted to the application layer, i.e. choosing the right data model design.

There is no common query language for NoSQL stores, like the SQL for relational databases. Even though there are some data stores like Cassandra and Hive that support a subset of SQL standard. Unstructured Query Language (UnQL)[5] aims to be a language specifically for unstructured data that can be used across the NoSQL landscape. With Couchbase as an industry-leader, the approach aims to create an open query language for JSON, semi-structured and document databases. It is much harder for a query optimizer to perform a global optimization taking into account latency, errors, etc. Hence, most NoSQL databases rely on explicit programmatic queries of a certain model such as MapReduce that can be executed on a distributed architecture.

MapReduce is a programming model for analytic and aggregation tasks for large scale data intensive applications. Actually the MapReduce framework is patented by Google, but ideas are adopted in a number of open-source implementations. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The *map* function processes a key/value pair to generate a set of intermediate key/value pairs, and the *reduce* function merges all intermediate values associated with the same intermediate key [DG08]. It is the equivalent of a relational table-scan. For the same purpose RDBMS databases use SUM, AVG and GROUP BY. The main purpose of MapReduce is to process large amounts of unstructured data and generate meaningful structured data as output. There are certain scenarios where you cannot use the MapReduce model, for example if the computation of a value depends on previously computed values.

2.2.3 NewSQL

As an alternative to NoSQL and custom deployments, a new class of parallel DBMSs, called *NewSQL* is emerging. They were driven by the need to preserve ACID guarantees while achieving scalability. This is based on using the ability of partitioning of OLTP workloads [PCZ12]. The same way the performance of applications using NoSQL stores depends on the data model design, the scalability of OLTP applications on many of these newer DBMSs also depends on the existence of an optimal database design. There are three approaches to achieve the scalability and still preserve the ACID guarantees and SQL interface [Pra]:

[5]UnQL: http://unqlspec.org

- New databases - designed from scratch to achieve scalability and performance. These solutions can be software-only (VoltDB, NuoDB and Drizzle) or supported as an appliance (Clustrix, Translattice). Examples of commercial offerings are: Clustrix, NuoDB and Translattice, and open source offerings: VoltDB[6], Drizzle[7], etc,.
- New MySQL storage engines - to overcome MySQL's scalability problems, a set of storage engines are being developed, which include Xeround, Akiban, MySQL NDB cluster, GenieDB, Tokutek, etc. The still use the MySQL interface, but the data migration from other databases (including old MySQL) is not supported. Examples of offerings are, commercial: Xeround, GenieDB and TokuTek; open source: Akiban, MySQL NDB Cluster and others.
- Transparent clustering - these solutions retain the databases in their original format, but provide a pluggable feature to cluster or shard transparently in order to improve scalability.

Note that this class of databases should not be confused with the new SQL database language. NewSQL is also a new SQL database access language which is still in the phase of collecting ideas and has partial functionality implemented. The idea is to have a standardized new query language to access a SQL database from Java because SQL is outdated and too complex. More importantly there is no real SQL standard, every product database has its own specifics, e.g. data types, keywords, size restrictions etc. In order to support the existing applications which are making use of SQL, to run with different databases, SQL to NewSQL and NewSQL to SQL converters are needed. Based on the Liberty Database Connectivity (LDBC) - a JDBC driver that provides vendor-independent database access - the SQL syntax is translated to each database. Applications can start using another database with no changes required. LDBC does not define a new database API, but it documents and enforces the expected behavior. With LDBC, applications will just work on all major databases and there is no need to change any source code [8].

2.2.4 NoSQL Adoption Drivers

The NoSQL databases were endorsed by the big companies, Google and Amazon, but their adoption spread in other companies as well, of a comparable size such as Facebook, Yahoo, EBay and many smaller ones. The drivers for NoSQL adoption are categorized as follows:

- **Schema Flexibility**
  Managing the continuously evolving schema and metadata for semi-structured and un-structured data, generated by diverse sources, is a convoluted problem [Tiw11]. Schema flexibility can be translated as the ability to handle semi-structured and un-structured data. Frequent changes to the schema are needed to react to frequent market changes and product innovations. Hence, the schema flexibility - also referred to as being schema-free or schema-less - is inducing organizations towards NoSQL solutions. According

[6]VoltDB: http://voltdb.com/
[7]Drizzle: http://www.drizzle.org/
[8]LDBC Liberty Database Connectivity: http://ldbc.sourceforge.net/

to Couchbase's[9] survey [Cou] this is the main reason, together with scalability, why businesses are moving to NoSQL solutions. The RDBMSs data model is not flexible enough to handle big data that contain a mixture of structured, semi-structured, and unstructured data. In NoSQL it is easier to deal with non-uniform data as most of NoSQL databases do not impose schema restrictions. This feature will avoid the insertion of many columns with null values or with naming such as CustomField in the relational model when dealing with non-uniform rows.
The term *schema-less* is a bit misleading and thus it requires some explanation. The lack of meaningful Data Definition Language (DDL) is conflated with the lack of a schema. In the NoSQL world the schema has to be defined by the application, because the data stream has to be parsed by the application when reading the data from the database. This schema is a set of assumptions about the data's structure in the code that manipulates these data. As the data has to be parsed by the application, in case of a schema mismatch is the application that should catch and throw these errors, not the database as was the case with RDBMSs. Thus, even in schema-less databases, the schema of the data has to be taken into consideration when refactoring the application [SF12].
For applications using document-based stores, in fact the application has schema which is implicit to the documents. Nevertheless, you still have to think about modeling aspects, such as: the type of relationship with graph databases, the names and order of columns in column-family stores or how is the key assigned and what is the value structure in key-value stores. The database doesn't care if different documents don't follow the same schema or if different rows have different number of columns. Also, if you find you don't need some data anymore, you can just stop storing them, without worrying about losing old data as you would if you delete columns in a relational schema. They offer a flexible schema design that can be changed without downtime or service disruption. This does not imply that you cannot do this in SQL, you can change with SQL statements the schema at anytime.

- **High Performance**
  The new Cloud OLTP workloads require higher throughput than traditional ones. The in-memory features of these databases can offer higher performance for applications that are ready to trade away some functionality, such as durability or consistency. NoSQL stores are designed to run well on clusters. This is supported by the concept of *Aggregate*. The first three NoSQL database categories have a big similarity: all have a fundamental unit of storage which is a rich structure of closely related data: for key-value stores it's the value, for document stores it's the document, and for column-family stores it's the column family. In Domain Driven Design (DDD) terms, this group of data is an **aggregate**. You create small domains of ownership so that you don't have to deal with the whole domain model. An aggregate is a collection of related objects that you wish to treat as a unit. The updates and the communication with the database is done in terms of aggregates. They also make a natural unit for sharding and replication, by enabling running of databases in clusters [SF12].

- **Application Development Agility**

[9] Couchbase: http://www.couchbase.com/

RDBMS schema should be changed before the applications changes, limiting this way the adding/updating of application features. In NoSQL, the focus shifts to *domain design* by empowering agile development of applications. Furthermore, NoSQL databases are, in many cases, easier to get up and running than are relational databases. NoSQL databases better fit modern application development. There is a shorter time needed to go from concept to implementation, thus making NoSQL databases appealing for growing companies looking for agility. This is the case when applications are designed from the beginning to use Cloud technologies (Cloud-native application). The situation is more complex when the application migrates from a local relational database to a NoSQL in the Cloud (Cloud-enabling it) as it will impact the business layer and the data access layer where further adaptations are needed. Going back at the case of [Ale], the team highlights the ease of using Amazon Web Services by noting the time saved using the pre-built libraries provided in Amazon Web Services' resource center enabling Alexa to kick start any new project with the comfort of building on top of existing libraries.

2.2.5 Polyglot Persistence

The kind of information that needs to be represented and stored has significantly changed over time. Features such as performance and availability are becoming paramount for the consumer facing applications, but the most critical enterprise applications still favor consistency over the availability. Line of Business (LOB) applications is the set of critical computer applications that are vital to run an enterprise, such as accounting, supply chain and enterprise resource planning. These applications will keep using the power of relational databases. The database consistency, query optimization, and set-based declarative query capability that relational databases have provided for decades is still required by most LOB applications; this has not changed [BBBI].
This situation has led to the term *Polyglot Persistence* which means that enterprises will use different databases that have different persistence approaches to solve different problems within one application or across the enterprise. For example, in an e-commerce scenario, transactional data storage and session information storage have different performance requirements (consistency, availability, and backup). You can live without a backup of the session information, it can be reconstructed in case something goes wrong, but you cannot afford to lose customer orders. Or the payment information may be stored in a transactional relational database to ensure consistency, while product catalogs and shopping carts might achieve better performance by using NoSQL stores [SF12]. One can also mix the use of NoSQL inside SQL. One benefit of this approach is that the same data store can be used for both the SQL and the NoSQL data. For example, PostgreSQL implements the *hstore* data type for storing sets of key/value pairs (simply text strings) within a single PostgreSQL value. This can be useful in various scenarios, such as rows with many attributes that are rarely queried, or when dealing with semi-structured data - the schema-less benefit of using NoSQL databases.

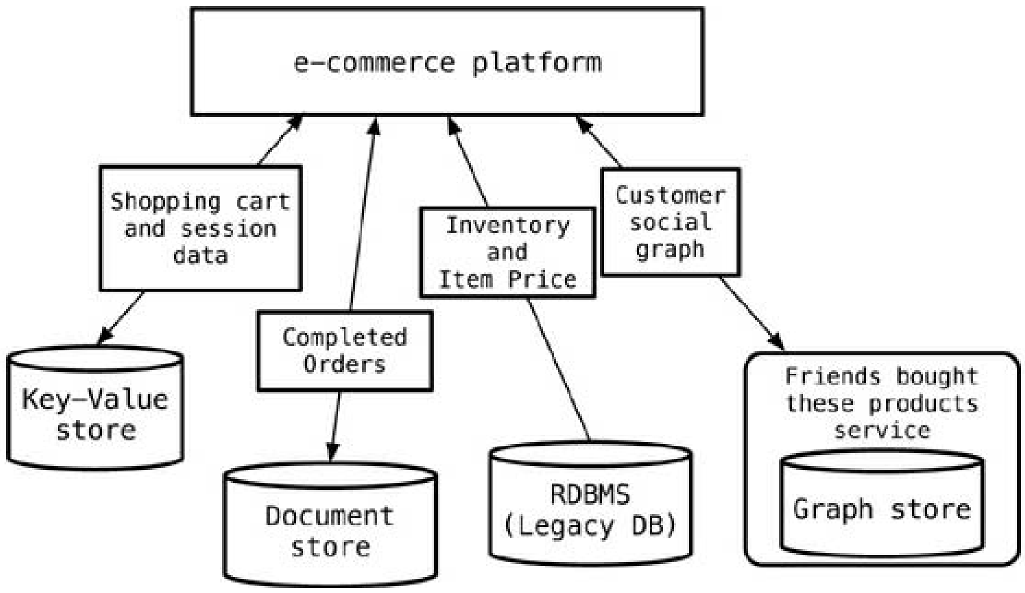


Figure 2.2: Polyglot Persistence in an e-commerce scenario [SF12]

2.3 Non-Functional Properties of NoSQL Systems

Features such as horizontal scalability, high availability, and low latency are considered the main drivers for NoSQL databases adoption. For this reason, in this section we will summarize the non-functional properties that are the actual adoption drivers of the NoSQL systems. Scalability is the most discussed feature of NoSQL databases. For many Web applications, scalability involves the elimination of latency in rather simple operations, such as pulling up an individual note, writing out a status message, bringing up account settings or the profile for a specific customer, and so forth [BBBI]. In order to keep up with the transaction growth of modern applications, having systems able to scale is very important. A system can scale either vertically, moving the application to a more powerful server (i.e. one that has more RAM, CPU), or horizontally by adding more servers to the infrastructure. Both approaches are expensive, but the first one is also limiting as there is an upper limit to how large a system can be. Horizontal scaling is more flexible but brings also additional complexity. It can be achieved in two dimensions when considering data storage:

- Functional partitioning - any good database design decomposes the schema into functionality-related groups, also known as tables. For example in an e-commerce scenario customers and orders are functionally distinct and typically are put into separate tables. Functional partitioning is important for achieving high degree of scalability. The database constraints such as foreign keys ensure the consistency between these functional areas [Pri08].
- Sharding is splitting the data within one functional area across multiple databases. It is also known as horizontal partitioning. Data can be distributed across multiple servers, so each server acts as the single source for a subset of data. For example: customer data of customers in different geographical regions can be put on different machines.

If one node responsible for a subset of data goes down, the whole data becomes unavailable. In order to avoid data loss and achieve high availability, the replication mechanism is applied - copying the dataset in more than one node (replica). There are two approaches to replication:

- **Master-slave** When updates made in one master node are propagated to other replica nodes. This kind of replication helps with read scalability as more than one node is serving read requests but doesn't help with the scalability of writes. Read resilience is provided only against slaves, while the master is a single point of failure for writes. In read-heavy workloads, should the master fail, the slaves can still handle read requests. However, you are still limited by the ability of the master to process and propagate those updates. So, it is not a good choice for datasets that are heavy write traffic.

- **The master-master** or peer-to-peer (or master-less) replication addresses the problem of the single point of failure. In this architecture there is no master, all nodes are functionally similar; each one is used to handle read and write requests. In case of a node failure, the access to the data store is not affected and adding nodes to the cluster in order to improve performance is easier.

The drawback of replication is the impact in consistency: you face the risk that different clients, reading data from different nodes, will see different values because the changes haven't all been replicated to the nodes. In distributed-storage systems that need to provide high performance and high availability, like NoSQL systems, the number of replicas is in general higher than two [Vogb]. Different NoSQL databases in order to have read consistency, ask more than one replica before sending the reply to the client. Read performance is improved further if replication is combined with caching techniques.

Relying on the database constraints to ensure consistency across functional groups creates a coupling of the schema to a database deployment strategy. In order to be able to apply the constraints, the tables must reside on a single database server, thus preventing horizontal scaling as transaction rates grow. Schemas that can scale to a high degree will place functionally distinct data on different database servers. This shifts the implementation of the data constraints to the application layer, not at the database level anymore [Pri08].

Partitions occur when some nodes in a system cannot reach other nodes, but both data sets are reachable by different groups of clients. Partitions can happen within and across data centers. A system that is not tolerant to network partitions can achieve data consistency and availability, and often does so by using transaction protocol such as 2PC [BANE09]. The 2PC protocol was invented to enable running transactions on multiple database instances. But to send a message over a WAN significantly increases the latency of a transaction (on the order of hundreds of milliseconds), a cost too large for many Web applications that serve heavy write requests. Horizontal scalability is easier to achieve without using the 2PC protocol [CST$^+$10].

The *Consistency Availability Partition tolerance (CAP) theorem* which lies at the heart of distributed systems has been used by the NoSQL movement as one of the arguments against

traditional relational databases. Formulated by Eric Brewer in year 2000 in a keynote presentation, the theorem states that it is possible to optimize for any two of Consistency, Availability, and Network Partition Tolerance, but not all three at the same time [Bre]. Said that, in case of a network partition a distributed system that is partition tolerant, it can either provide availability or consistency.
The rise of large Web applications and distributed data systems, makes the partition-tolerance property inevitable, thus imposing compromise on either consistency or availability. Dynamo was the pioneer of eventual consistency as a way to achieve high availability and scalability for its shopping cart application; data reads are not guaranteed to be up-to-date but all the nodes will see the updates eventually.

The relationship between CAP and ACID is more complex and often misunderstood, in part because the "C" and "A" in ACID represent different concepts than the same letters in CAP and in part because choosing availability affects only some of the ACID guarantees [Bre12].(In case of a partition, there is no ACID "I" because concurrent operations may occur on both sides of the partition. And there is no ACID "C" because you have only local integrity checks). The "C" in CAP - Consistency - means that there is the same copy of data in all the nodes where it is replicated, also referred to as *single-copy consistency*. When some nodes crash or some communication links fail, it is important that the service still perform as expected. This explains the "A" in CAP, and it has a different meaning than the classic one we have for availability. A failed, unresponsive node doesn't infer a lack of CAP availability. The availability in CAP means that a request should always be responded by a non-failing node [SF12].

This has introduced a new data semantics approach - **BASE** - a design philosophy created in the late 1990s to capture the emerging design approaches for Internet services for which the primary value was high availability of data. Compared to ACID which adopts a pessimistic approach and forces consistency at the end of every operation, BASE follows an optimistic approach that data will be available after a short period of time [10]. It stands for Basically Available, Soft state, Eventually consistent. The three techniques that achieve the BASE semantics are [FGC+97]:

- **Stale data** - services can tolerate stale data knowing that all copies will be eventually consistent after a short time.
- **Soft state** - in order to achieve high performance data is not written persistently; in case of failure it can be regenerated at the expense of additional computation.
- **Approximate answers** (based on stale data or incomplete soft state) delivered quickly may be more valuable than exact answers delivered slowly.

The systems that favor availability on consistency have a BASE approach. Another trade-off that is not part of CAP and which exists even when there are no network partitions, is the one between consistency and latency. Deciding for an asynchronous replication strategy versus a synchronous one comes to the point of choosing between consistency and latency. The synchronous approach wait until all updates have made it to the replica(s) before returning to

[10]This time during which the client can see outdated data is called inconsistency window

the user. It ensures strict consistency but as communication (especially over WAN) between independent entities is involved, it brings latency. The asynchronous approach doesn't introduce latency, requests can be processed without waiting for the propagation of updates to other nodes. But, there is the risk that read request will be served by outdated nodes.

This was the situation for year 2000. The modern view of CAP in 2012 is that the choice between availability and consistency is much more fine grained and complex. Perfect availability and consistency in the presence of partitions is rare, but there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. The decision can be made more than once, and at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved [Bre12].
Vogel, the CTO of Amazon distinguishes between client-side and server-side consistency. The client-side deals with how consistency is perceived by the client, while server-side deals with how updates are propagated through the system and that the developer of the system can experience [Vogb].

- Client side: Strong, Weak, and Eventual (a specific form of weak consistency). The eventual consistency has the following variation forms:
  - Casual consistency (read your write consistency, i.e. not reading in between a write)
  - Read your write: if a process has updated a data item, it will always access the updated value and not the old one.
  - Session consistency: if a process has updated a data item, it will always access the updated value and not the old one.
  - Monotonic read consistency: if a process has seen an updated value, any subsequent access will never return an older value of that data item.
  - Monotonic write consistency: the system guarantees to serialize the writes by the same process.
- Server-side has to do with the configuration of the replicas. The following settings:
  - N- total number of replicas for the dataset
  - W- the number of replicas that need to acknowledge the receipt of the update before the update completes
  - R - the number of replicas that are queried during a read operation

  depend on what is the common case and which performance path needs to be optimized. For example, In R=1 and N=W (read only from one node) you optimize for the read case, and in W=1 and R=N (write to only one node) you optimize for a very fast write. In the latter case, durability is not guaranteed in the presence of failures, and if W < (N+1)/2, there is the possibility of conflicting writes when the write sets do not overlap.

W and R are respectively Write and Read quorums. The eventual consistency rises when having W+R<=N configuration, i.e. having no overlapping of nodes.

2.3.1 CAP Systems

Given the different behaviors of database system when they have to deal with partitions, the following is a popular classification of systems through the lens of CAP.
CP systems: database systems that adhere to ACID properties, focus on CAP consistency

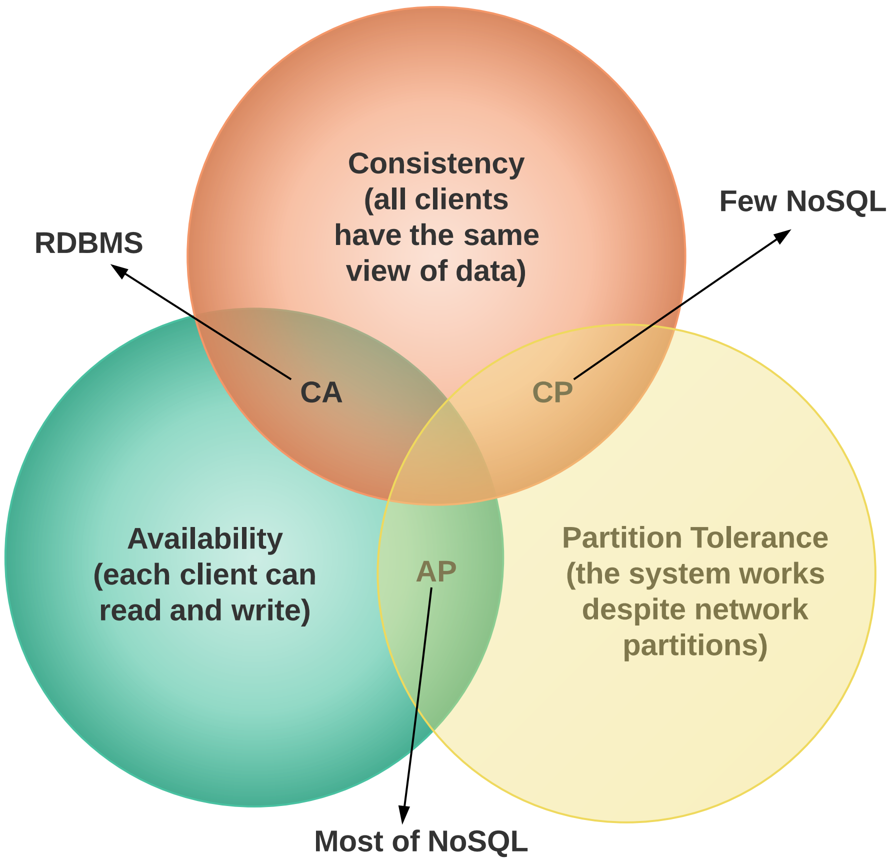

Consistency (all clients have the same view of data)
RDBMS
Few NoSQL
CA
CP
Availability (each client can read and write)
AP
Partition Tolerance (the system works despite network partitions)
Most of NoSQL

Figure 2.3: CAP Systems

first and then availability. They forfeit availability for consistency in case of partitions.
AP systems: NoSQL systems designed to support applications that need to be highly available, in case of partitions they forfeit consistency, thus falling into the AP category. The case of PNUTS, the NoSQL system from Yahoo, seem not to fit into this definition. PNUTS relaxes consistency by only guaranteeing "timeline consistency" where replicas may not be consistent with each other but updates are guaranteed to be applied in the same order at all

Table 2.1: SQL vs NoSQL vs NewSQL Summary

Features	RDBMS	NoSQL	NewSQL
Data model	Relational model	Domain driven	Relational model
Transactions	ACID (almost all)	BASE	ACID
Querying	SQL	REST, Client libraries, Protocol buffers	SQL
CAP classification	CA	AP/CP	mainly CP

replicas [CRS+]. It also gives up availability - if the master replica for a particular data item is unreachable, that item becomes unavailable for updates.
CA systems: refer to systems that are not tolerant to network partitions, traditional RDBMS fall into this category. But what if a partition happens? It means that they loose availability, thus falling into the same group as CP systems.
Tunable Consistency: allows the user to decide the level of consistency he wants. The consistency level is a setting that clients must specify on every operation (insert, update, read) and that allows the user to decide how many replicas in the cluster must acknowledge a write operation or respond to a read operation in order to be considered successful. As many of the NoSQL can tune the consistency level in case of partitions, this is the reason some NoSQL systems fall into AP and CP categories.
Table 2.1 gives a summary at a high level of the features these systems provide.

2.4 NoSQL Database Categories

A comprehensive summary of NoSQL databases and their features is given by Pramod and Martin in their "NoSQL Distilled" [SF12] book and a very up to date list of the databases is available online at [NoS]. In this thesis, we give a summary of NoSQL databases focusing on the application functionalities and non-functional properties that each category support. This summary is not claimed to be exhaustive as this is an immature and fast-growing market, and adoption drivers are still evolving.

It is important to understand each category, because they serve certain scenarios (applications and workloads) best. Comparison of data models is easy, while comparing them in terms of performance is a challenging task. Attempting to evaluate NoSQL systems all together conflates too many different needs with too many different kinds of required optimizations, and prevents meaningful comparison [NE]. Yahoo Cloud Serving Benchmark (YCSB) is an extensible framework from Yahoo Research that has become a standard for evaluating the performance of NoSQL stores. Currently they have developed two benchmark tiers: scalability and performance, while future addition on availability and replication have been announced. Performance of these systems depend on the partitioning, replication, and transactional consistency that they have chosen. To understand the performance implications of these decisions for a given type of application is challenging [CST$^+$10].
Enumerations of NoSQL categories tend to vary, and there are cases when a database can fall in more than one group. In the following sections we will briefly introduce the four main categories of NoSQL stores.

2.4.1 Key-Value Stores

Key-value stores are the simplest NoSQL data stores and can bee considered as the mother of all NoSQL databases. Even though they are not the same, they have many things in common. A HashMap[11] is the simplest data structure that can hold a set of key-value pairs and they all store data as maps [Tiw11]. They were inspired by Amazon DynamoDB storage model. The APIs the databases of this category offer the "match query" options. A match query extracts the value associated with a certain key.

- Get(key) - extracts the value given a key
- Put(key, value) - creates or updates the value given its key
- Delete(key) - removes the key and its associated value

They are easy to use from an API perspective. - the client can either get the value for the key, put a value for a key, or delete a key from the data store; that is the reason why they are easy to scale and generally have great performance. The aggregates can be stored into one single bucket which is a namespace for keys. But storing all different objects in one single bucket increases the chance of key conflicts. An alternate approach is to append the name of the

[11]HashMap is an associative array where the index is called the "key"

object to the key, for example 288790b8a421_userProfile so that can be accessed as needed. The value can be a blob, text, JSON, XML, and so on. For the data store (not for all) the value is opaque, it is the application that should understand what is stored. The query characteristics make key-value stores likely candidates for storing session data (with the session ID as the key), shopping cart data, and user profiles [SF12].

Some key-value stores get around the "opaque" nature of the value by providing the ability to search inside the value, such as Riak. *Riak* [Ria] - a Riak cluster is masterless, automatically redistributes data when you scale, and keeps data available when physical machines fail. Motivated by Amazon alike use cases, it stores data as key-value pairs, has a simple operational model, and comes with an HTTP API and many client libraries. Any data can be stored, in any desired format as all objects are stored on disk as binaries. It provides a feature, *Riak Search* which is a distributed, full-text search engine, that allows you to query the data just like you would query it using Apache Lucene or Solr indexes.

Solr[12] is the popular fast open source enterprise search platform from the Apache Lucene project. Lucene is a simple search library that can be easily integrated into your application. Its core facility manages indexes. Documents are parsed and indexed and stored away into a storage scheme, which could be a filesystem, memory, or any other store.

Redis- is an open source, BSD licensed advanced key-value store. It is often referred to as Data Structure server since keys can contain strings, hashes, lists, sets and sorted sets. It is an in-memory system and thus, provides optional durability [Red].

Memcached[13] - it's a distributed memory object caching system which demonstrated that in-memory indexes can be highly scalable, distributing and replicating objects over multiple nodes. It dedicates blocks of memory on multiple servers to cache data from your data store. It is free and open source, high-performance, generic in nature, but intended for use in speeding up dynamic Web applications by alleviating database load. It is being used Facebook, Twitter, Wikipedia, YouTube, and many others. It is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

MemcacheDB - is a persistent variant of memcached; it is a distributed key-value storage system which is API-compatible with memcached. MemcacheDB uses BerkleyDB as a storing backend, so lots of features including transaction and replication are supported [Mem].

BerkeleyDB[14] is a high-performance embeddable database providing SQL, Java Object and key-value storage. It offers advanced features including transactional data storage, highly concurrent access, replication for high availability, and fault tolerance in a self-contained, small footprint software library. It was recently bought by Oracle, the well established vendor for the relational database.
Oracle, has also developed its NoSQL solution, Oracle NoSQL. It is also a distributed key-value database. It offers high availability, rapid fail-over in the event of a node failure and

[12]Apache Solr: http://lucene.apache.org/solr/
[13]Memcached: http://memcached.org/
[14]Oracle Berkeley DB 11g: http://www.oracle.com/technetwork/products/berkeleydb

optimal load balancing of queries. At the core of Oracle NoSQL is BerkeleyDB.
Oracle Coherence[15] - is a memory cache layer on top of a database that takes a key-value approach.

The limitation for key-range processing of key-values stores is overcome by ordered key-value model which significantly improves aggregation capabilities. Some of the above mentioned systems are ordered key-value stores (BerkeleyDB, MemcacheDB) which provide the search function of "range query" because it sorts the keys in ascending order. Preserving some order while storing keys makes it possible to run efficiently a range query to extract the attributes associated with a key [Sho].
Almost all of the other categories of NoSQL systems were built, whether physically or conceptually, upon key-value store principles. Therefore you should expect their applications to be more specialized than, but not completely distinct from, those of key-kalue stores themselves [BBBI]. The main key-value stores offered as a service by the big Cloud vendors are Google AppEngine Datastore[16], Amazon SimpleDB, Amazon DynamoDB, and Microsoft Azure Table Storage that we will introduce in more detail later on.

2.4.2 Document Databases

Document-oriented databases were inspired by Lotus Notes, the IBM product for collaboration. The model is basically a key-value in nature where the value is a document, namely a collection of other key-value collections. And this is the difference with key-value stores which store only scalar values and cannot embed an object into another object. A document can be addressed by unique URL, and given the HTTP and URL orientation, they offer RESTful APIs for the applications. The semi-structured documents are stored in formats like JavaScript Object Notation (JSON), XML, Binary JavaScript Object Notation (BSON). Documents can also contain attachments, thus making document stores useful for content management. The value as opposed to key-value stores, is usually structured and understood by the database. Querying is possible not only by the key, but by the fields of the value as well.

Document databases are more suitable for low latency, high performance and are not suitable if your application requires: multi-document transactions, complex security needs such as: user roles, document level security, authentication, complex joins across collections, BI integration, extreme compression needs. [Cou]. They support ACID transactions only at the document level.
Examples of document databases: MongoDB, CouchDB, Couchbase, DjonDB, RavendDB etc. The two leading document databases are CouchDB and MongoDB. They both are most directly relevant to Business Intelligence because of their more flexible and extensive search and retrieval functionality. They are both open-source and have a big community support. We decided to chose MongoDB in this thesis work.

[15]Oracle Coherence: http://www.oracle.com/technetwork/middleware/coherence
[16]GAE Datastore: https://developers.google.com/appengine/docs/python/datastore/#Python_Datastore_API

CouchDB, an eventually consistent database available under Apache 2.0 license, is being used by BBC and Credit Suisse for dynamic content platforms to store configuration details for its marketing data framework. Another successful use of CouchDB adoption was in a streaming network analysis use case [WDDB12].The schema-free nature support the storage of emails (arbitrary metadata fields) and the simple mechanism of storing binary data support the storage of attachments. CouchDB support *views* which consist of MapReduce functions, stored as documents and replicated between nodes like any other data.

Couchbase[17], which run on a CouchDB/Membase server, it is more suitable for interactive Web applications. It also has a Hadoop connector for analytical heavy workloads. With the new release Couchbase Server 1.8 replaces Membase Server [18].

RavenDB is transactional, open-source document database written in .Net. Its JSON objects can be queried efficiently using Linq queries from .NET code or using RESTful API using other tools.

XML databases is a category of databases that is often considered as document databases because of the structured value they have. XML databases are very similar to document-oriented, the document is stored in a data model compatible with XML. You can use various forms of XML schema definitions (DTDs, XML Schema, RelaxNG) to check document formats, run queries with XPath and XQuery, and perform transformations with XSLT [SF12]. Saving the document in XML, which is a structured format offers the advantage of not treating the value as simply a blob type. Document databases offer JavaScript server side functions. For example a call of MapReduce function would be: mapreduce(keyList, mapFunc, reduceFunc) - which invokes the functions across a key range. Some databases offer REST APIs, low-level or standards-based APIs (Java, Python, etc.), some do provide a SQL-like query language, like MongoDB, SimpleDB. The latter one might help ease the transition from the RDBMS world by putting less effort in the query transformation.

2.4.3 Column-Family Stores

The databases in this category are based on Google's BigTable model and this is the reason they are also referred to as BigTable clones. They are built for storing and processing very large amounts of data. They are also a form of key-value pairs, but they organize their storage in a semi-schematized and hierarchical pattern. Google's BigTable is: "...a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row-key, column key, and a timestamp; each value in the map is an un-interpreted array of bytes." [CDG$^+$08]. This model later influenced the column oriented data stores, like Cassandra, Hypertable, HBase. These are the main databases here, making the competition less in this category compared to relational and key-value stores [RWC12].

Cassandra and HBase feature column families. Column families are groups of related data(columns) that is often accessed together. In a typical column-oriented store, you predefine a column-family and not a column. The number of column families is virtually unlimited,

[17]Couchbase: http://www.couchbase.com/
[18]Couchbase Server: http://www.couchbase.com/membase

thus making a fairly common use of the column name as a piece of runtime populated data [OGOG+11]. A row key is mapped to column families which are mapped to column keys, and this structure is more a two-level map than a table as shown in figure 2.4. They are

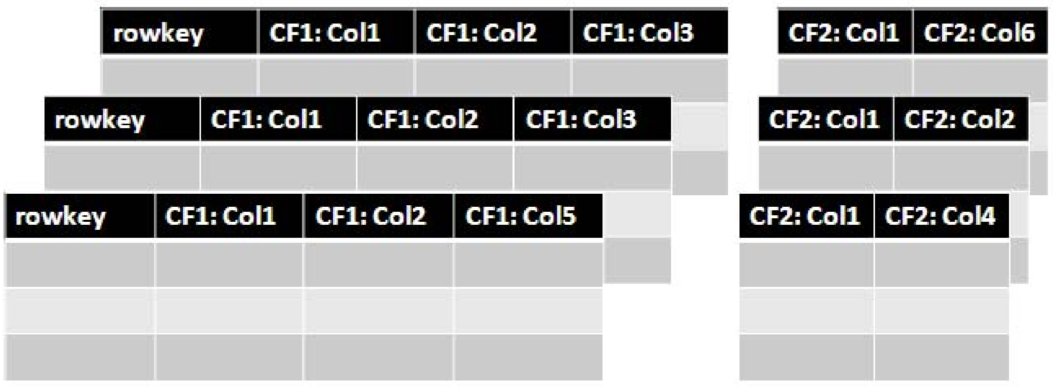


Figure 2.4: Column-oriented Data Model

less schema flexible than document databases, are best used for semi-structured data, not for data whose structure changes from row to row. Said differently, wide columns do well also in grouping entities that have high-level characteristics in common, but with different context-specific attributes [BBBI]. Apache Cassandra [Hew11] is a schema-less data store, meaning that it enforces no requirements that the rows contain similar columns. HBase's architecture is similar to BigTable (using synchronous updates to multiple copies of data chunks). Just as BigTable leverages the distributed data storage provided by the Google File System, Apache HBase provides BigTable-like capabilities on top of Hadoop and HDFS. Also, HBase is included as a component of the IBM InfoSphere BigInsights product which is targeted at the big data market [Moh13]. It is suitable when you need random, realtime read-/write access to your big data[19]. We will explain in more detail the data model of Cassandra, as we have chosen it for our validation approach. Cassandra was originated by Facebook's inbox search, and it has now a full project status at Apache and also an active developer group. It is in commercial use at a variety of companies, such as Digg, Reddit, and Twitter. It has commercial support by DataStax[20].

2.4.4 Graph Databases

Relational databases are simply not great for hierarchical or graph data. These types of modeling require lots of one-to-many and many-to-many relationships, which can't be modeled efficiently in a relational database. Graph databases are most suitable if relationships are paramount as they are ideal for capturing any data consisting of complex relationships such as social networks or product preferences. In other words, they are best suited when the application has datasets with complex interconnection and do not offer support for ad-hoc

[19]HBase: http://hbase.apache.org/
[20]DataStax: http://www.datastax.com/

queries [SF12]. They are a good choice for applications that need fast and extensive reference-following, especially where data fits in memory [Cat]. They are not based on aggregates, that is why they are also referred to as *aggregate-ignorant*. Entities are represented as nodes and the relationship between them as edges. Both, nodes and edges can have attributes. These databases focus more on the relationships between entities than the entities themselves. Edges can be added or removed at any time, allowing one-to-many and many-to-many relationships to be expressed easily and avoiding anything like an intermediate relationship (aka mapping)table that you might use in a relational database to accommodate many-to-many joins. There will be plenty of competition from social networks and other proprietary vendors. Google's Knowledge Graph and Twitter's Interest Graph are similar to Facebook's Social Graph, and traditional database vendors such as Oracle and IBM are getting into the Graph act, too. Microsoft Research's "Trinity" project is another example of a graph database [Neoa]. Additional examples for graph databases are:

FlockDB[21] - Twitter's database, is simply nodes and edges with no mechanism for additional attributes.

Neo4J - allows you to attach Java objects as properties to nodes and edges in a schema-less fashion. Neo4J is fully ACID compliant, without wrapping creation operation in a transaction you get an exception. There are many reasons for using such database, mainly is performance and scalability. If in your algorithm you need to do many join query to get your data from a RDBMS, in Neo4J it is a simple traversal of nodes following a given relationship. Neo4j Enterprise, which runs on clusters is being used by Cisco which is building a master data management system. Other companies are making use of Neo4j, such as Adobe, Squidoo and Intuit [Neob].

Infinite Graph[22] - stores Java objects, which are subclasses of its built-in types, as nodes and edges databases most suitable for navigating relationships, etc. FreeBase(Google). It distributes the nodes across a cluster of servers as opposed to most of the graph databases which run on single servers. And as most of them run on single servers, data is always consistent within a server.

It is obvious that for good performance graph traversal, the related nodes should reside in one server. Even though graph databases are not aggregate-oriented like document databases and key-value stores, there are still some techniques to make these relationship oriented databases scale across a cluster.

SPARQL[23] – is a W3C-standardized Web-scale graph query language that is distinctly more powerful than SQL and is now becoming the de facto language for the emerging Linked Data Web.

As stated previously, in reality this classification is too crude. OrientDB[24] calls itself both a document database and a graph database. MariaDB foundation was created to bring together many developers in the open source project MariaDB/MySQL with one of the goals

[21]Introducing FlockDB: https://blog.twitter.com/2010/introducing-flockdb
[22]InfiniteGraph: http://www.objectivity.com/infinitegraph
[23]SPARQL Query Language for RDF: http://www.w3.org/TR/rdf-sparql-query/
[24]OrientDB: http://www.orientdb.org/

of MariaDB being to be a bridge between NoSQL and SQL. That is the reason they have added support first for Cassandra and are now working on adding support for LevelDB: a key-value store by Google[25]. From the above summary we can abstract and the common characteristics of the NoSQL databases into the following:

- Not based on the relational model
- Schema-free: allowing fast application development
- Distributed: Running well on clusters
- Mainly Open-source: there are some proprietary solutions as well
- Horizontally scalable
- Built for the 21st century Web estates

[25]leveldb: https://code.google.com/p/leveldb/

3 Related Work

The reality is that many companies have migrated their applications to NoSQL databases and have shared their experience they had during the migration process. In Section 3.1 we briefly introduce what are the traditional schema mappings and schema mapping frameworks, and how does our approach differ from them. Very little is done in terms of comparing the relational and NoSQL database schema objects in detail. Mainly the focus has been on how to model the data, but less is done with respect to data types, indexing, and constraints mapping or migration possibility. And in our context mapping also means: in case some schema elements are missing in the target schema, e.g. stored procedures, how can they be emulated at the target data store. In Section 3.2 we will summarize some of the migration use cases with respect to motivation drivers and the best practices during data modeling, and more importantly the impact on the upper application layers. As it is impossible to cover all of them in this thesis, we selected among them those applications that reflect different requirements and cover the different NoSQL categories. Section 3.3 gives a summary of the available tools for moving data from a RDBMS to NoSQL.

3.1 Traditional Schema Mappings

There are currently many kinds of scenarios in which heterogeneous systems need to exchange, transform, and integrate data. These include ETL applications, Object-Relational Mapping (ORM) systems, Enterprise Application Integration (EAI) frameworks [BMPV]. In all these scenarios many data sources with different data models and data formats need to communicate. *Schema mappings* are high-level declarative specifications that describe the relationship between data in two heterogeneous schemas [FHH+]. The are widely used in data exchange, data integration, schema evolution, etc. A schema mapping is the specification on how an instance of a source schema corresponds to potentially many target instances. Figure 3.1 illustrates a simple schema mapping where the two sources are mapped to a target schema, in this case the mappings are simple correspondence lines. As schemas started to become larger and more complex, transformation designers had to express their mappings in complex transformation languages and scripts. In order to do this, first they had to obtain a good knowledge and understanding of the semantics of the schemas and of the desired transformation. Many frameworks have been developed and evolved over time [BMPV]. From the schema representations the frameworks can either generate:

- View - to reformulate the queries against a schema into queries for another schema for data integration purposes or
- Data transformation - from one schema representation to another for data exchange.

Mappings can be created not only between relational schemas, but also between relational and nested schemas, such as XML documents. In our approach schema mapping has a broader

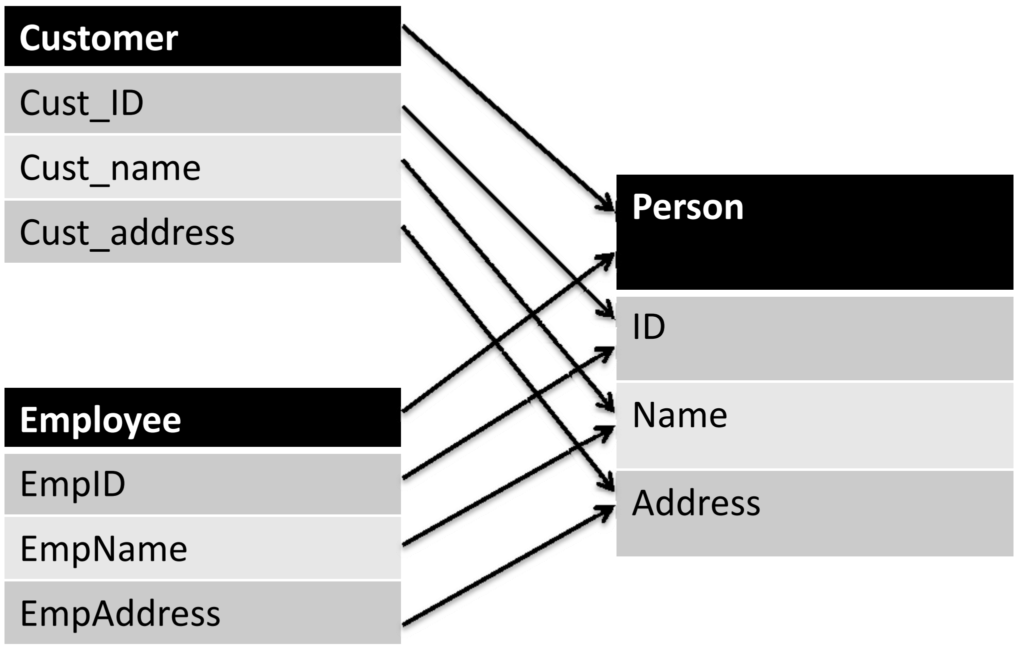


Figure 3.1: Relational Schema Mapping Example

scope, meaning that we investigate which relational schema objects can be created in the target NoSQL schema, and for those that cannot be mapped directly, we investigate how to re-factor the application to emulate them. So, we do not have to deal with the constraints of the target schema as in the normal schema mappings. Our approach consists in taking the relational source schema, migrate it to a NoSQL schema by creating the target NoSQL schema based on some assumptions about query patterns.

3.2 Migration Use Cases from RDBMS to NoSQL

As mentioned previously, there are many application use cases migrating from relational to NoSQL, and it is impossible to cover them all in this thesis. We selected among them those uses case that reflect different application requirements and use different NoSQL categories. Most of use cases about migration to NoSQL don't do a comprehensive analysis of all the relational database schema elements, those that can be mapped at a high level like index types and those that can be mapped physically, like tables, rows, columns.

Netflix.Inc, a company that provides movie and TV shows streaming, migrated from Oracle to Amazon Web Services (AWS) SimpleDB and S3, because AWS promised better availability and scalability in a relatively short amount of time [Ana]. Before migration they had only one data center and were faced with interruption of services in case of outages. Instead of building more data centers and dealing with scalability issues themselves, they chose to migrate to Amazon Web Services, focusing this way on their core competencies. Netflix shared the experience in a report discussing the impacts at the data access and business logic layer for the missing functionality. "SimpleDB's support for an SQL-like language appealed to our application developers as they had mostly worked within the confines of SQL" - thus, we incorporate this feature into our questionnaire, i.e. Does the data store provides SQL-like query language or not. Also, other recommendations for refactoring the application, such as implementation of JOINs, Group BYs, constraints checking, sequences, recommended to be done at the application layer. For the emulation of sequence, they recommend the usage of a distributed sequence generator, in case no natural occurring key can be used as a sequence.

Alexa.com offers Web-scale services built on top of AWS. Building their application on the notion that timely and relevant information is important to a positive Web experience, they chose Amazon SimpleDB over MySQL to store intermediate status and log data, and Amazon S3 to put and retrieve datasets. To power Alexa Site Thumbnail service, Alexa uses Amazon S3 to store and deliver millions of thumbnail images and uses Amazon SimpleDB to automatically index and efficiently query the stored images [Ale].

We learn from their experience that unique identifiers can be generated at the application code, either using functions provided by the programming language, or using UUIDs for more stringent applications. Thus, SimpleDB provides efficient scalability by shifting the responsability of uniquely identifying the data to the application code. Also, because of its SQL-like query language, the query reusability is high [Lea].

eBay has adopted Cassandra in their "Social Signal" project, which enables like/own/want features on eBay product pages. A few use cases have reached production, while more are in development. They give a summary of best practices of data modeling with Cassandra for their use case, from which we subtract some good modeling practices when considering to migrate to Cassandra [Pat].

Foursquare - a location-based social network originally started with MySQL, then moved to PostgreSQL and then all changed when the service took off with users [10gb]. Growing rapidly the company needed efficient scaling for which PostgreSQL promised to involve significant work, so they started reviewing other options, such as MongoDB, Cassandra, CouchDB, and sharded MySQL. They decided to migrate to MongoDB hosted on Amazon Web Services for storing venues and check-ins, the latter being the largest dataset.

Also **Art.sy** which indexes and makes searchable high-quality images of 30000-plus works of art from over 3000 artists, transitioned to MongoDB operated by MongoHQ[1] which offers MongoDB as a Service. We decide to chose this provider also for the validation of our approach.

Nokia's Ovi Places Registry [Far], moved from MySQL to an internally developed NoSQL key-value data store as the data was growing bigger and bigger. They used Apache Solr, the open source enterprise search platform from the Apache Lucene, for query performance. Given their experience with an in-house developed data store they advice to consider an open source NoSQL store if you decide to migrate to a NoSQL solution. This is an important

[1]MongoHQ: https://www.mongohq.com/

feature to be included in our questionnaire, text-search integration with engines such as Apache Lucene or Solr.

Riak is very popular key-value store. To help retailers evaluate and adopt Riak, Basho has published a technical overview: "Retail on Riak: A Technical Introduction." where they discuss more in-depth information on modeling applications for common use cases, switching from a relational architecture, querying, multi-site replication and more. We will reuse and consider this knowledge as input especially during analysis and specification of functional requirements.

Zynga - a provider of social game services - is serving over 235 million active users per month. In a social game, the ratio of reads to writes could be as high as 1:1. In order to cope with this challenging throughput they have redesigned the storage of their game platform. They started using Couchbase which has a Memcached built in, the most widely deployed in-memory caching technology, thus enabling consistently low-latency data reads and writes. They have improved the performance and availability of their games while reducing hardware and administration costs. The same experience we see in the case of ideeli[2] which wanted to offer high performance and being always available for their "add to cart" application [Basb]. Taking this into consideration, for applications looking for high performance, we incorporate the *Built-in caching layer* as a non-functional property for the Cloud data hosting solution, and *Integration of a caching layer* in case the data store doesn't provide one. Also, they built a custom replication protocol atop the existing memcached binary protocol, given the fact that it is open-source, which resulted in reducing the replication time of MySQL from seconds to milliseconds [Zyn].

EPIC, faced with the challenge of scaling the persistence tier due to the enormous amounts of data generated from a single mass event, moved from MySQL to Cassandra to handle the large amount of data. From their experience we learn that the transformation of the system's existing data model is the key design change that must be accomplished for a successful transition from relational to NoSQL technology. There are a lot of lessons learned during this process, with the most important challenges in: software architecture, data modeling, deployment, developers' skills, etc. With respect to data modeling they provide the lessons learned on how to model joins when moving to Cassandra. They have found that in order to enable scalability many of the software engineering best practices, like data normalization and object-oriented design heuristics must be implemented outside of the persistence tier [SA12].

With respect to data modeling best practices, the online documentations of the NoSQL databases provide guidelines on the best practices when using their data store [Mona], [Datb], [Sim]. Product and service specific recommendations and guidelines are available, but in our work we want to abstract from them and provide in first place more general recommendations with respect to the choice of NoSQL solution and the required adaptations of the upper application layers. From the official tutorials about different NoSQL databases, we have abstracted also features to include in our questionnaire.

[2]ideeli: http://www.ideeli.com/

3.3 Schema Mappings Frameworks

Schema mappings are high-level specifications that describe how data structured under one schema (the source schema) is to be transformed into data structured under a different schema (the target schema). This is a very complex task when mapping between RDBMS schemas. The schema mappings are the essential building blocks in data exchange and integration [ACKT]. Data exchange is the process to physically transform the source data into the target format. Many tools and methodologies are developed with respect to RDBMS to RDBMS schema mappings. In this thesis we have the relational data structure as given objects and transform it into a NoSQL data structure.

As NoSQL systems are not in a mature state yet, there is no systematic theory of data modeling techniques. In addition, different techniques apply to different categories as they are based on different data models. Moreover, even within the same data model there are different approaches and different features that different vendors offer. Most of the online documentations of the NoSQL databases provide guidance on data modeling best practices. The same focus is also other resources [Mona], [Sim], [Datb], [Hew11], [Basa]. No one focuses on the migration from a specific relational database, like we do with MySQL 5.6 and PostegreSQL 9.2. Riak [Basd] provides a background and detailed level of understanding when moving from relational to Riak data store.

A collection of data modeling techniques and patterns is given at [HSB]. Around seventeen patterns are identified and explained. No relational database system is taken into consideration. We take in consideration MySQL and PostgreSQL schema, analyze their features against NoSQL offerings to identify what do you lose when migrating to NoSQL databases, identify the solutions in form of patterns that can be applied to these shortcomings.

Datastax, the commercial support of Cassandra, combines Cassandra, Hadoop, and Solr together into one big data platform. It includes in the Enterprise 2.0 version, support for Sqoop, a tool designed to transfer data between a RDBMS and Hadoop. In this release data can be moved with Sqoop not only to a Hadoop system, but to Cassandra as well. Only MySQL tables and data is moved to Cassandra. The tool connect to MySQL via JDBC driver which should be downloaded, while our approach connects to the source database through command line, and no additional driver is needed.
Kettle[3] is an ETL tool from Pentaho for data integration. It provides connectivity with a variety of data sources, among them the NoSQL databases MongoDB, Cassandra, HBase. Kettle (unlike Sqoop that provides only extract load operations) allows developers to create some transformation routines to customize how a MySQL schema and data are moved to Cassandra. There is no comprehensive analysis available with respect to the impact on the data access layer and on the business logic layer when moving to NoSQL databases. Also, an important feature of our methodology is the mapping of data types, knowing that RDBMSs are strongly typed and NoSQL are loosely typed, it's very important to be aware of data that might not be imported properly, or might lose precision in case of numeric values. Furthermore, currently there is no general support that abstract guidelines that does

[3]Pentaho Kettle: http://kettle.pentaho.com/

not focus on a single concrete NoSQL solution. One of the main aspects of our work is the decision support system for providing guidance when choosing the right NoSQL store. To our knowledge there is no comprehensive decision support tool that incorporates the functional and non-functional requirements of the NoSQL databases to support this process and provide guidance about application refactoring. Our focus is on decision support system for NoSQL solution and the adaptations of the application layers, and these aspects are not covered by ETL tools. Migrating from RDBMS to NoSQL is a creative process and is difficult to be automated because there are no standards that these databases adhere to. One has to look beyond a given database and understand how the application uses the data, identify which data sets need to be accessed fast, and which uses are not frequent and perhaps not even required. Then you need to see how the target database can be used to best support these uses.

4 Analysis, Concept and Specification

In the first part of this chapter we analyze the current state of the Cloud Data Migration Methodology and Tool developed by Bachmann [Bac]. We refer to the tool as CDMT in the rest of this work. In Section 4.5 the functional and non-functional requirements of NoSQL stores are analyzed with respect to the questionnaire. We specify the features to be added to the provider knowledge base in order to realize the decision support process. In Section 4.2 and 4.3 we analyze the relational model and the non-relational model with respect to data modeling to identify the adaptations of data access and business logic layer. The selected relational databases used for the validation of our work and their features, such as data types, indexing, and some limitations are analyzed in Section 4.4. The same approach we follow for the selected NoSQL stores in Section 4.5. We summarize a set of good and not recommended modeling practices in NoSQL in Section 4.6. We provide a set of comparison tables summarizing the findings of this analysis.
In the second part, Section 4.7, we map the findings of the analysis to the components of the system. Section 4.7.1 gives a summary of the relational schema objects mapping to NoSQL.

4.1 System Overview

In his diploma thesis, Bachmann developed a methodology for migration of the database layer to the Cloud [Bac]. Bachmann based his methodology on the methodology of Laszewski et al [TN]. His methodology consists of the six following phases:

1. Identify Cloud migration scenario
2. Describe desired Cloud data hosting solution
3. Select Cloud data store
4. Identify patterns to solve potential conflicts
5. Adapt data access layer and upper application layers if needed
6. Migrate data to the selected Cloud data store

He identifies eleven migration scenarios, later reduced to ten [SABL13]. He developed a tool that supports the decision process of selecting a Cloud data store and helps refactoring the application architecture by providing suggestions on Cloud Data Patterns and potential adaptations of the network, data access, and application layer. A Cloud Data Pattern describes a reusable and implementation technology-agnostic solution that should be applied to the

data to emulate the missing functionality when moving them to the Cloud [SABL13].
The application itself is a Java Web application, which uses a local MySQL database as data layer. The tool consists of two components: decision support tool (classifications) and migration tool (migration). The decision support tool includes also the gathering of different Cloud data stores and supports various adapters for target systems, for relational databases in the Cloud and one adapter for source system, MySQL. The focus of this thesis is on

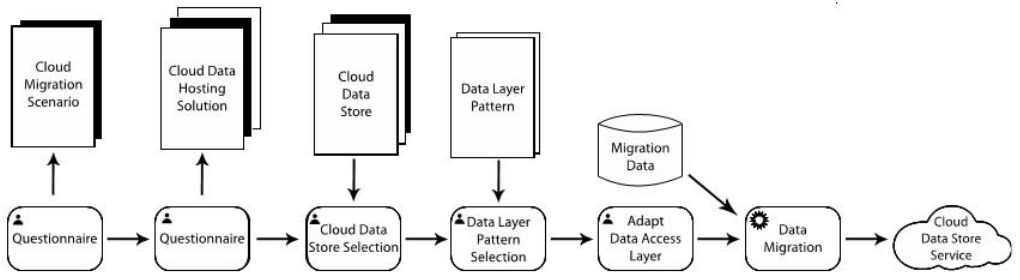


Figure 4.1: Cloud Data Migration Tool[Bac].

migration scenario: *RDBMS to NoSQL* - migration from a RDBMS with strong transactional and consistency guarantees to a NoSQL store with focus on being highly available and offering high performance.
We develop a new methodology for this migration scenario. We extend the Cloud Data Hosting Solution taxonomy incorporating all the features abstracted from Chapter 3 and based on the analysis of NoSQL stores in Section 4.5 and Appendix 8.2. We also remove some of the categories in the taxonomy that cannot be applied to NoSQL stores. For example: *Compatibility criteria*, when adding a new NoSQL data store in the list of Cloud data stores, makes no sense for NoSQL stores. We suggest the needed adaptations at the data access layer and the business logic layer when considering the mapping of relational to NoSQL database schemas. There are two types of users of the tool: software developer and Cloud data store expert. A user can have multiple projects in which he defines the scenarios with the corresponding migration strategy values. At the beginning we analyzed the current state of the tool and methodology and in the following we describe the improvement suggestions.
When selecting the target data store, the comparison of the properties of the desired CDHS with the actual data store selected, do not show all the conflicts. For example, if replication method for the CDHS is set to synchronous, and the target data store offers only asynchronous, no conflict is shown.
Step 1b - Refine Cloud Data Migration Strategy
Source Data Store type and Target Data Store type should not be editable. The user should not be able to change the target data store type as this scenario is meant only for NoSQL databases and is already defined in Step 1a. And if saved these changes, in Step 1c there are shown the conflicts. Instead of generating conflicts we don't allow the user to change the settings, stating previously that the locked values are not possible for editing in this scenario.
Step 1c - the generation of conflicts based on the comparison of the migration strategy chosen by the user and the values that apply to the migration scenario by default.
Step 4a - What is the data store type of the source system? This can be logically detected from the migration scenario. Even though the logic is the same for all the criteria, to allow the user

try many combinations and see the implications, for this migration scenario we don't follow this logic. Does the system support joins? Every RDBMS supports joins, thus to be removed from the list. Only if the system uses joins should be as input, in order to generate the proper adaptations when moving to NoSQL.
During the last step, i.e. the migration of the data, the user can choose between all the available stores, it has to be limited only to the selected data store.
For our migration scenario, RDBMS to NoSQL, we distinguish between the categories that are suitable for this scenario, and also within one category which properties make sense to show and which not.

4.2 Relational Model

Database schema design is the process of choosing the best representation for a data set given the features of the database system, the nature of the data, and the application requirements. Data modeling in RDBMS is consistent, because the theory on which it is based is well established and implementation is standardized. Therefore, consistent ways of modeling and normalizing data are well understood and documented. The relational style allows to model objects as database tables and the relationships between objects as primary and foreign keys that link the tables together. These relationships can then be exploited by issuing queries via SQL. One of the main aims of relational database design is to normalize an input relation schema together with a set of data dependencies into an appropriate normal form. The motivation behind the various normal forms (2NF, 3NF) is to eliminate the problems that are caused by the update anomalies and redundancy problems [Cod90].

After normalizing data into multiple tables, the next step is to model the relations between the tables. There are three major types of relations: *one-to-one* (relates a single record to a single record in a different table), *one-to-many* (relates a single record to multiple records in a different table) and *many-to-many* (relates multiple records in a table to multiple records in a different table). Modeling many-to-many relationships requires an extra 'join-table' or a 'mapping table'. The join-table contains two extra columns that reference the primary key column of each table in the relation. If a relation has its own attributes, a join-table can be used to model the relationship, regardless of the relationship type. The relational model goes hand-in-hand with SQL - a declarative language in which a programmer specifies *what* data he wants and not *how* the system should do it. The queries can range from simple SQL queries, such as filters, e.g. retrieve all order records whose customer ID =100 to more complex ones. More complex constructs cause the database to do some extra work, such as joining data from multiple tables (e.g., what is the name of the product that employee with ID=100 has ordered?). Other complex constructs such as aggregates (e.g., what is the average price of orders per customer) can lead to full-table scans. It is the database engine that takes care of query optimization.

Relational modeling is typically driven by the structure of the available data. The access pattern is not known in advance. The tables are modeled, assumptions are made regarding the access patterns, and these assumptions are translated into predefined optimizations like index

definitions. In other words, in SQL, the data model does not enforce a specific way to work with the data — it is built with an emphasis on data integrity, simplicity, data normalization and abstraction, which are all extremely important for large complex applications. One of the advantages of normalization over hierarchical data is the ability to perform ad-hoc queries, i.e. to join tables on conditions not predicted in the original data model.
In RDBMSs primary keys are automatically indexed, and they are also called *primary indexes*. *Secondary indexes* allow the system to lookup data by different keys, not only by the primary key. For example, if you want to search for a customer ID knowing only his name, in SQL you would write a query like this: SELECT CustKey from Customer where name ="John". Then the database would inspect each row of the Customer table to find the name column value that matches the search string. This is called a full-table scan. This slows down the performance when the data size is big. That is why indexes are used. They store a copy of data that the database can lookup very fast. Creating a secondary index on the *name* column, results in a faster lookup in this case.

4.3 Non-Relational Model

In the NoSQL world there is no standardized and well-defined data model as in RDBMS world. Running the database on clusters changes not only the rules of the data storage, but also the rules of computation. In order to process and access the data efficiently you have to think differently how you organize your processing [SF12]. The data model of these systems represents a big shift from the relational model. In the NoSQL world data modeling is typically driven by the *query access patterns* which are application specific. Based on the queries you need, you define the candidate "keys" to use for partitioning. Doing so makes sure that the data within the range is within one partition. These are application specific, referring to how does the application read and update data. It is fundamental to know your data access patterns before starting data modeling. For example, when retrieving the content of a blog post, will also all the comments for that post be displayed? Or when accessing customer profile information, will also all the orders he has done be displayed? These should be distinguished from the *data access patterns* that describe the way read/write operations are handled by the storage disks. There are four main data access patterns are: sequential reads, sequential writes, random reads, random writes. It is fundamental to know also these data access patterns before choosing a NoSQL store because the current storage systems focus on optimizing for one type of workloads at the expense of other workloads due to limitations in existing storage system data structures. With modern applications using increasingly more complex queries and larger data-sets, data access patterns have also become more complex and randomized [SSM$^+$]. Accessing data sequentially is much faster than accessing it randomly because of the way in which the disk hardware works [Micb]. To reduce random writes, systems such as Cassandra, HBase, Redis, and Riak append update operations to a sequentially-written file called a log (commit-log). The log is frequently synchronized to disk. Cassandra groups multiple concurrent updates within a short window into a single synchronization call, thus achieving high write throughput [Mar]. Data model is driven also by the transaction needs, i.e. how are you going to update the data. Knowing that most of NoSQL stores do not support atomicity across multiple aggregates, you model the data within the boundary of the transaction support: within one row (column-family), document (document-based), item (SimpleDB, DynamoDB, etc), or entity group (GAE Data store).

In NoSQL you first design the queries and then model the data around them. The applications that will make use of NoSQL solutions, have to give up the good software engineering principles such as object-oriented design heuristics and data normalization to gain performance and application development agility. At the heart of NoSQL data model is the principle of *de-normalization*. Application developers should not be reluctant of duplicated data as the assumption with NoSQL technologies is that storage is cheap. Now the complexities of design shift from the persistence layer to the application layer. Scaling out becomes easier if the system doesn't need joins, thus NoSQL stores sacrifice complex query capabilities. In NoSQL systems, joins are often handled at design time during data modeling as opposed to relational models where they are handled at query execution time [HSB].

Most NoSQL systems don't support high level query languages with built-in query optimization. But, there are some exceptions, e.g. MongoDB offers a query optimization feature, the command *cursor.explain()* provides you information on a query plan. They sacrifice the power of declarative query, thus relying on applications to enforce data integrity [SF12]. Even though they don't enforce referential integrity, consequently not being able to implement joins, it is a common design principle to store IDs that reference other entities in your data store. Contrary to the proponents of application development agility as a driver toward adoption of NoSQL systems (together with the simple APIs argument), the skeptics highlight the fact that the application developer will have to worry about optimizing the execution of their data access calls with joins or similar operations as they have to be implemented in the application layer. The query optimizations of NoSQL systems are not even close to the sophisticated ones found in mature RDBMSs [Moh13]. Microsoft identifies eight applications types that can be considered for migration to the Cloud (Microsoft Azure, 2012). Out of these eight, the ones highlighted in bold are those applications that can benefit from NoSQL solutions as well.

1. SaaS applications
2. **Highly-scalable Web sites**
3. Enterprise
4. Business intelligence and data warehouse applications
5. **Social or customer-oriented applications**
6. **Social (online) games**
7. **Mobile applications**
8. **High performance or parallel computing applications**

Table 4.1 and 4.2 provides a summary of where each category of NoSQL databases is best used for and for which use cases it's not the appropriate solution. This classification is a summary based on [SF12], [BBBI], [Oraa] and the analyzed use cases from the industry.

Table 4.1: Application Use Cases for Key-Value And Document Databases

Subcategory	Best used for	Not suitable for
Key-value	• Big Data • Cache or Blob data • Session and Shopping cart data • Online Social Gaming • User profiles	• Complex query and aggregation needs • Relationships between sets of data • Multi-operation transaction • Key ranges processing
Document-oriented	• Big Data • Event Logging • Content Management Systems • Blogging Platforms • Web and Real-Time Analytics • E-Commerce Application • Online Social Gaming	• Multiple document transaction • Ad-hoc queries

Table 4.2: Application Use Cases For Columnar And Graph Databases

Subcategory	Best used for	Not suitable for
Column-family	• For massive scalable systems with (semi)structured data • Big Data • Event logging • Content management systems • Mobile and Social platform • Web analytics purposes	• ACID transactions for writes and reads • For query patterns that changes as it requires changes in the column-family design
Graph databases	• Social networks • Semantic Web data • Routing, dispatch and Location-based Services • Recommendations engines	Global graph operations

We can categorize the drivers of NoSQL adoption as follows:

- Highly available: for consumer facing applications (Cloud OLTP workloads [CST$^+$10]) or dynamic Web sites where high availability is crucial.
- High performance: The class of online gaming applications together with the explosion of smart devices, poses high performance requirements as these applications allow users to quickly launch a game, connect to a server, and collaborate with other players that are online.
- Application development agility: when you are on a tight project schedule, but the data fit more on the relation model. Nokia use case: having a project with a tight deadline, they developed it all with MongoDB as it was very quick to develop. Now they are going back and porting it to MySQL because it's really more of a relational data set [Leo]. For this scenario document databases are recommendable, because they are easier to use than Hadoop or Cassandra.
- Product innovation: when data stores provide high flexible schemas that can support the continuous changing format of the data. The requirement for schema flexibility is not necessarily connected to being highly scalable. You might have a small set of data that their schema changes frequently. This is the reason of being classified as a different use case.

The first two rely on the system being scalable to achieve their scope. The latter two exploit the schema-less nature of the NoSQL databases and the agility they offer in terms of programming.

Most of the data sets mentioned in Table 4.1 and 4.2 like Web traffic log files, social networking status updates, advertisement click-through imprints, road-traffic data, stock market tick data, and game scores are primarily, if not completely, written once and read multiple times. Such data sets have limited or no transactional requirements at all [Tiw11]. The commit/rollback functionality across many tables as in RDBMSs is generally not available in NoSQL solutions — a write either succeeds or fails. The atomicity of transactions is mainly guaranteed at the aggregate level (even though there are some NoSQL systems, like RavenDB that says to support multiple document transactions [Rav]. Using transaction protocols like 2PC and Paxos makes it difficult for databases to scale out, especially between data centers, as WAN communication is involved.

Table 4.3: RDBMS versus NoSQL Features Comparison

Features	RDBMS	NoSQL
Data model	Relational model	Domain driven
Data modeling drivers	Start from available data	Data access and update patterns
Transactions	Almost all support ACID	Atomic transactions at the aggregate level
Data types	Strongly typed	Loosely typed
Joins	Yes	Emulated at the application layer
Indexing	Primary, secondary and different storage types	Limited
Design Complexity	Persistence layer	Application layer
Role-based access functionalities	Yes	No support
Data integrity	Responsible is Persistence layer	Shifted at the application layer
Consistency	Strong (also tunable by the application)	Eventual (also tunable by the application)
Schema mismatch detection	Database	Application/Data Access layer
Query support	Complex and ad-hoc queries	Not suitable for ad-hoc and complex queries
Query language	SQL	REST, Client libraries, Protocol buffers
Query optimization	Responsibility of database	Responsibility is shifted to the application

Table 4.3 summarizes based on the analysis, the differences between RDBMS and NoSQL databases. Data integrity is not ensured by the database, as there is nothing like foreign key, cascade update/delete in NoSQL to preserve the integrity of data. This makes NoSQL stores not suitable for applications that have high requirements for data integrity. Riak support the three methods for querying, ie.e REST, client libraries and protocol buffers [Basc]. The indexing features are more limited compared to RDBMSs, but we see that are being attempts to enrich these features. For example, even key-value stores are adding new features with time, like Dynamo recently announced secondary indexes support [Ama]. Also Riak database supports secondary indexes. Even though NoSQL databases are not good in general for complex and ad-hoc queries, there are exceptions like MongoDB which is famous for its dynamic query capabilities.

4.4 Relational Databases Analysis

Schemas are descriptions of data schema in a formal language supported by the database management system. A schema consists of schema objects and their interrelationships. The schema objects include: columns, tables, relationships, views, indexes, packages, procedures, functions, queues, triggers, types, sequences, materialized views. There are others, such as synonyms, database links, directories etc. which we do not consider in this work. All of these schema objects names are known as *identifiers*. A very popular problem in dealing with RDBMSs is the *impedance mismatch* problem. The object-relational impedance mismatch problem arises when trying to combine object and relational artifacts, which results in conceptual and technical difficulties. More specifically they arise when a RDBMS is being used by an Object-Oriented Programming (OOP) language or style where class definitions are mapped to database tables or relational schemas. One solution to this problem are the ORM frameworks that handle the interactions with the database, enabling the client to interact only with objects and their relationships. These object mappers are convenient, because they facilitate validation, type checking, and associations. But, they can not scale well for large datasets as it pulls the information on relationships between objects into memory[SA12].

The focus of this thesis is on tables, columns, relationships, data types, constraints, indexes for MySQL version 5.6 and PostgreSQL 9.2.

- **Constraints** are used to constrain column data. The supported constraints by both MySQL and PostgreSQL are: Primary key, Foreign key, Not-Null, Unique, Check. MySQL accepts the Check constraints but it ignores them, while PostgreSQL enforces them.
- **Indexes** - They are good for faster retrieval, but slow down insert and update operations. They can be implemented using a variety of data structures such as B-Trees, Bitmap, Hash, etc. MySQL and PostgreSQL create by default an (hash) index on the primary keys of a table. B-tree indexes are more flexible for less-than, greater-than, equals-to matches queries [RWC12]. Hash indexes are used only for equality comparisons,=,<=> but are very fast. Most of the key-value stores rely on this type of single-value lookup. They are not used for comparison operators such as < or > that find a range of values.
- **Stored programs** include stored procedures and functions: database objects defined in terms of SQL code that are stored on the server for later processing. Stored program definitions include a body that may use compound statements, loops, conditionals, and declared variables. User-defined functions(UDF) can be regarded as an external stored function.
- **Triggers** are named database objects that fire automatically when insert, delete, update (or other events) occur. It is a type stored procedure, but it runs based on events on a table instead of just being a set of instructions to be executed repeatedly.
- **Events** are tasks that the server runs according to a schedule.
- **Views** are stored queries that produce a result set when referenced. A view acts as a virtual table.

4.4.1 MySQL 5.6

MySQL doesn't make a distinction between *database* and *schema*. MySQL knows many storage engines such as MyISAM, Innodb, Memory, BerkleyDB, NDB and many more. The implementation details (full text search, transactions, foreign keys, check constraints, upper and lowercase, etc.) are dependent upon the engine. InnoDB storage engine is fully ACID compliant, but using InnoDB in MySQL fails ACIDity, because MySQL doesn't propagate the triggers of foreign keys for this engine.

- Data types We consider for migration the MySQL version 5.6 and the following analysis apply only to this version.
- Indexes - MySQL indexes all data types. It can store the indexes internally as B-Tree, R-Tree, Hash, Fulltext. Knowing what kind of indexes does the data store uses to store indexes is important because each of them supports better a class of query operators. For example, Hash indexes are intended for queries that use equality operators. *Prefix* indexes (also referred to as partial indexes) cover the first N characters of a string column, making the index much smaller than one that covers the entire width of the column.
  Composite indexes are created on more than one column.
  Full-text indexes are used for full-text searches and are created only on CHAR, VARCHAR and TEXT columns.
  Spatial indexes used for spatial data types.
  Unique is an index on a column or set of columns that have a unique constraint. This helps in enforcing data integrity.

Schema/database and table names' case sensitivity depend on the underlying operating system. While column names are case sensitive on all OS, Database and table names are not case sensitive in Windows, but are case sensitive in most varieties of Unix, because they are stored as files in the underlying operating system. The case sensitivity of the identifiers might be relevant if you decide to preserve it, and in case the target data store is case in-sensitive. So, if you are running your database in Unix and migrating to a NoSQL store with SQL-like capabilities where identifiers are case sensitive, the re-usability of the queries will be much higher.

4.4.2 PostgresSQL 9.2

PostgreSQL makes a distinction between a database and schema, as opposed to MySQL. Some of its language features are extensions to the standard SQL. PostgresSQL use PL/pgSQL which is a block-structured language. It is fully ACID compliant [Posa]. PostgreSQL does not know different storage engines as compared to MySQL. A database contains one or more named *schemas*, which contain other named objects such as: tables, data types, functions, and operators. They are analogous to directories at the OS level, with the difference that they are not nested. PostgreSQL uses transactions everywhere. PostgreSQL supports stored functions, which are in practice very similar to MySQL stored procedures.

Table 4.4: MySQL Limitations

Feature	Size Limit	Description
Storage	256TB MyISAM, 56TB InnoDB	Depends on storage engine
Columns	4096 per table, but depends on storage engine	E.g Innodb has 1020 columns per table
Maximum row size	65,535 bytes	In all storage engines
Secondary Indexes	64 per table	
MVCC	In InnoDB supported, no support in MyISAM	Multi Versioning Concurrency Control
Foreign Key	In InnoDB supported, no support in MyISAM	To ensure data integrity

- **Data types** - apart of the numeric, date and string data types, PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays can be created on any built-in or user-defined base type, enum type, or composite type. In New in PostgreSQL 9.2 are the JSON data type and the Range data type which stores a range of values for a given data type. For a complete list of PostgresQL data types, check 8.1. *Text* data type has a variable unlimited length. The SQL standard defines a different binary string type, called BLOB or Binary Large Object. The input format is different from bytea, but the provided functions and operators are mostly the same. For the boolean type TRUE and FALSE are the preferred (SQL-compliant) usage.

- **Indexes** - PostgreSQL stores indexes internally as: B-tree, Hash, GiST, SP-GiST and GIN. R-tree index method has been removed from version 9.2 because it had no significant advantages over the GiST method.
  Function based index: an index column may not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table, e.g. CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1)) creates an index with on the lowercase values. This is supported only by PostgreSQL, not by MySQL.
  Partial index: has a different semantic from the ones in MysQL. These are indexes built over a subset of a table which is defined by a conditional expression (called the predicate) using the Where clause. The index contains entries only for those rows that satisfy the condition.
  Multicolumn indexes (MySQL composite equivalent) are created on more than one column. Currently, only the B-tree, GiST and GIN index types support multicolumn indexes.

- **Constraints**
  Apart of the constraint mentioned above, *Exclusion* is a constraint specific to PostgreSQL. It ensures that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return false or null. Table and column names are case sensitive.

4.5 NoSQL Stores Analysis

We dive into each of the data stores with respect to data model, data types, keys, indexing, querying, partitioning and the limitations they have.

4.5.1 Amazon SimpleDB

- **Data Model** - simply large collections of items organized into domains. Items are described by attribute name-value pairs. Customer account, the equivalent of a RDBMS instance, is Amazon Web Services account to which all domains are assigned. Because of the flexible nature of Amazon SimpleDB, it is entirely possible to use a single domain to store all of the data for an application. Each item can have different number of attributes, an item can have multiple attributes with the same name, an (attribute name, value) pair should be unique.
- **Data types** - in SimpleDB, everything is stored as a UTF-8 string value. Primitive data types like integers and floats are not natively supported. Developers choose a unique string name for each item at creation time. Whatever format that MySQL chose to export your data it will be retained as String in SimpleDB. This makes your migration easier because it minimizes application logic changes [Lea].
- **Keys** - to assign unique identifiers, RDBMSs utilize locking so that to avoid duplicates. But locking has a negative impact on scalability. Amazon SimpleDB chooses a much simpler approach. By shifting the responsibility of creating a unique identifier to the application code, the creation of a unique value is a trivial operation for any programming language. For more stringent applications, you can use a Universally Unique Identifier (UUID) instead. As it doesn't matter whether the identifier follows a pattern or not, when migrating data to SimpleDB, the old values generated by the source databases can be kept.
- **Indexing**- data in each domain is indexed on all attributes. They require no formal definition because the SimpleDB service creates and manages them behind the scenes.
- **Querying** - predefined libraries, languages specific wrappers can be used for the Amazon SimpleDB operations. Attributes can be searched with various lexicographical queries. Conditional puts and deletes are exposed via the PutAttributes and DeleteAttributes APIs by specifying an optional condition with an expected value. They are useful for preventing lost updates when different sources write concurrently to the same item. Sorting is lexicographical in SimpleDB. If you plan on sorting by certain attributes, then zero-pad logically-numeric attributes. For example, the string "10" comes before "2" in lexicographical order. If you zero pad the numbers to five digits, "00002" comes before "00010" and are compared correctly.
- **Partitioning**- you need partitioning in SimpleDB when:

Table 4.5: SimpleDB Identifiers limits

Identifier	Limit	Allowed characters	Sensitivity
Domain name	3-255 chars	a-z, A-Z, 0-9, '_', '-', '.'	Yes
Item name	1KB	All UTF-8 characters that are valid in XML documents	Yes
Attribute name	1KB	All UTF-8 characters that are valid in XML documents	Yes
Attribute value	1KB	N/A	N/A

- The required throughput cannot be provided by a single domain. In order to achieve high performance data can be partitioned across multiple domains, to parallelize queries, thus improving overall throughput. But, this partitioning across domains should be done manually.
- Queries of your application may require most of the items in the domain to be examined. In this case, the size of your domain will likely influence the query performance.
- When the data produced by your application exceeds the storage limit of 10GB per domain.

In case when the dataset does not naturally present an easy parameter for partitioning, hash functions can be applied to create a uniform distribution of items across multiple domains.

- **Joins** are not supported natively by SimpleDB. The data you normally would store in a related table could be de-normalized into multiple values in a single attribute, to avoid intersections.
- **Transactions** - SimpleDB does not support complex transactions. The transactional semantics offered: Conditional Puts/Deletes — enable you to insert, replace, or delete values for one or more attributes of an item if the existing value of an attribute matches the value you specify.
- **Limitations** - Many of the structures defined in a typical relational schema such as: stored procedures, triggers, relationships, and views do not exist in SimpleDB. Fields and types exist in SimpleDB but are flexible and are not enforced on the server. Attribute size limit is 1KB, and combine that with the limitation of a maximum of 256 attributes, the maximum size of an item can be 256 KB. A consideration to consider during data migration, also for the storage requirements of your data sets.

4.5.2 Apache Cassandra 1.2

- **Data model** of Cassandra consist in the following:
  - Keyspace is the first dimension of the Cassandra hash, and is the container for column families. Keyspaces are of roughly the same granularity as a schema or database (i.e. a logical collection of tables) in the RDBMS world.
  - Rowis a collection of columns or super-columns identified by a key. Each column family is stored in a separate file, and the file is sorted in row (i.e. key) major order. The row key determines which node the data is stored on. Related columns, that are accessed together, should be kept within the same column family.
  - Column Families is the analogous of a RDBMS table. A column family (CF) is a container for columns. A column family holds an ordered list of columns, that can be referenced by the column name. They are also called *sparse tables*, because the number of columns can vary per row.
  - Column is the smallest increment of data. It's a triplet that contains a name, a value and a timestamp. The timestamp is used by Cassandra for last write conflict resolution.
  - Super Columns is a column whose values are columns, i.e. a (sorted) associative array of columns.

  The questions you should ask before start the modeling in Cassandra is how do you can organize data into that map to satisfy your query requirements of fast look-up, ordering, grouping, filtering, aggregation, etc. Because of Cassandra's highly distributed nature, you cannot introduce new queries in Casandra by just adding secondary indexes. Still you can start with entities and relationships modeling and then continue modeling around query patterns by de-normalizing and duplicating. A change in schema required a server restart in < 0.7 versions.

- **Data types** - the only schema information that must be defined for a table is the primary key (or row key) and its associated data type. Composite built-in type for columns. Unlike in relational database where column names can be only of String type, both row keys and column names can also be long integers, UUIDs, or any kind of byte array.

- **Keys** - the primary key (row key) determines which node the data is stored on. Cassandra offer the possibility to have compound primary keys: PRIMARY KEY (column1, column2,..,columnN). Composite partition keys: You can declare a composite partition key formed of multiple columns by using an extra set of parentheses to define which columns form the compound partition key: PRIMARY KEY (*(column1, column2)*,..,columnN). In this case all the rows with similar column1 and column2 values will be stored into the same physical node.

- **Indexes** - primary indexes for a table is the index of its row keys. Each node maintains this index for the data it manages. **Secondary indexes** - Cassandra offers secondary indexes where the range of the indexes values is low and finite (low cardinality). For example if the table Customer has billion users, indexing the *nation* column is a good choice, as many users will share the same value. If you create indexes on columns that have many distinct values, a query between the fields will incur many seeks for very few results. If the nation column had been indexed by creating a table such as CustomersByNation, your client application will have to populate the table with data from the Customer table. Cassandra implements secondary indexes as a hidden table, separate from the table that contains the values being indexed. You can create multiple secondary indexes. The limitation is that only equality queries are supported.

- **Querying** - Cassandra is optimized for hash tables rather than ordered tables in performing read and range queries. It uses consistent hashing which ensures no remapping of keys when slots are added or removed. Cassandra Query Language (CQL)is an SQL-like query language, compatible with the JDBC API. Cassandra stores identifiers as lower case. You can force the case by using double quotation marks (in CQL3). There is no support for ORDER BY and GROUP BY statements in Cassandra as there is in SQL. There is a query type called a SliceRange, it is similar to ORDER BY in that it allows a reversal. It supports multidimensional range queries since version 0.7 with a limitation that there must be one dimension with an equal operator in the query expression, which hinders the broad usage of these queries. Cassandra does data validation only on column names, not on row key data type or column value, and that only for sort order purposes.

- **Partitioning** - the two major policies are: *random partitioning*(RP) and *order-preserving partitioning* (OPP). OPP has one obvious advantage over RP: it provides the ability to perform range queries, but it might cause load-balancing problems [OGOG$^+$11].

  Tunable consistency: The most usable level is quorum. Quorum level guarantees that half of the nodes are updated before the action returns. Another option is the *Write all Read one* model in which case every read will be consistent. Cassandra does not update data in-place on disk, nor update indexes, so there are no intensive *synchronous* disk operations to block the write. That is the reason why Cassandra is fast in writes. By setting the durable_writes to *no* (default is set to True), you can bypass the data being written to the commit log, and write only to the memtable. But, in this case you risk losing data. **Triggers** Cassandra doesn't support triggers natively. Even though you can implement your own, like the case of Cassandra Async trigger [GH] a procedure that is automatically executed by the database in response to certain events on the column families. A trigger is set on a column family and is executed in case of any update to the column family. In contrast to traditional triggers, which are synchronous, Cassandra Async triggers are asynchronous and are implemented in Java, by implementing the *execute* method of ITrigger interface.

4.5.3 MongoDB 2.4

MongoDB is one of the most popular document databases and with a strong community support, and these were the reasons we chose this database for the validation in the document database category.

- **Data Model** - MongoDB manages collection of JSON documents. This allows data to be nested in complex hierarchies and still be queryable and indexable. A collection is the analogous RDBMS table with the difference that it has not a predefined schema. Collection and attribute names are case sensitive. A document can contain complex structures such as lists, or even documents.

  An example of a MongoDB JSON document format for an order based on the TPC-H database schema:

```
 Order = {
_id: "123456789",
"OrderKey": 7556,
"CustKey": 9118,
"OrderDate": ISODate("2010-09-24"),
"OrderPriority": 1,
"OrderStatus": "SHIPPED",
"TotalPrice": 130,
"Clerk": 3,
"ShipPriority": 2,
"Comment": "This order has been shipped"
}
```

Listing 4.1: JSON Document Format

  The value of comment can be also a list of documents, for example: comment:["by": "Joe", "date": ISODate("2012-10-15")].

- **Data types** - each document is stored in BSON format, which is a binary-encoded representation of a JSON-type document. MongoDB drivers and clients serialize and de-serialize to and from BSON when they access BSON-encoded data, while the server understands BSON and doesn't need to apply serialization. The extra data types supported by BSON are: regular expression, binary data, and date.

- **Indexing** - each document has a unique identifier, which can be generated by MongoDB or by the application. The _id is the primary key of each document. The _id can be composite attribute as well, thus supporting composite primary keys. MongoDB indexes every _id field in all collections. Indexes can be as well defined on any other attributes of the document. MongoDB supports indexes that include content on a single field, as well as compound indexes that include content from multiple fields. When queried, documents in a collection are returned in natural order of their _id in the collection.

- **Querying** - one of the good features of document databases, as compared to key-value stores, is that you can query the data inside the document without having to retrieve the whole document by its key and then introspect the document. This feature brings

these databases closer to the RDBMS query model.
Server-side JavaScript execution: where aggregation functions written in JavaScript are sent directly to the database to be executed. There is a special system collection named *system.js* that can store JavaScript functions for reuse on the server side. MongoDB offers a new mechanism called Aggregation Framework for queries that require things like MAX, AVG or GROUP BY from SQL, which allows to run some ad-hoc aggregation queries without need to write cumbersome Map-Reduce scripts [MB11]. Some of the new features included in version 2.4 is *Text indexing* which offers native, real-time text search. Also, in order to detect peak times, Working Set Analyzer provides data about server resource usage. API - MongoDB provides drivers for most of the programming languages. Cloud providers for MongoDB as a Service, like MongoHQ and MongoLab do offer REST API as well. MongoDB has good support for Web framework integration. Rails, one of the most popular Web application frameworks, can be used effectively with MongoDB. The data from Rails applications can be persisted via an object mapper.

- **Sharding and Replication** - MongoDB supports ordered, range based partitioning with the any user specified field. MongoDB provides automatic sharding, but you need it only in the following cases, otherwise you will just add extra complexity. 1. Your data set approaches or exceeds the storage capacity of a single node in your system.
  2. The size of your system's active working set will soon exceed the capacity of the maximum amount of RAM for your system.
  3. For write-heavy applications, when a single MongoDB instance cannot write data fast enough to meet demand.
  Sharding is done in an order-preserving manner, thus making it easier to support range queries and indexes.
  Shard Key determines the distribution of the collection's documents among the cluster's shards. The shard key is a field that exists in every document in the collection. MongoDB distributes documents according to ranges of values in the shard key. A given shard holds documents for which the shard key falls within a specific range of values. They can be either single field or compound of more than one field.
  Hashed Sharding are new in MongoDB 2.4. They use a hashed index of a single field as the shard key to partition data across your sharded cluster. They work well with fields that have a good cardinality and increase monotonically like ObjectID values or timestamps.
  Replica Sets and Master-Slave are alternatives for achieving replication in MongoDB. The original implementation was Master-Slave until version 1.6 onwards which contained the Replica Set feature. From then on, Replica Set is a functional superset of the Master-Slave setup which supports asynchronous replication, automatic fail-over and automatic recovery of a member node. Features that master/slave didn't originally support.
  You can specify in the code different levels of consistency at the operation level. This involves a trade-off that you need to carefully make, based on your application needs and business requirements, to decide what settings make sense for slaveOk (i.e. you can read from the replicas) during read or what safety level you desire during write with WriteConcern (if set to REPLICAS_SAFE a write is not acknowledged before being

Table 4.6: Best Workloads and Use Cases for selected NoSQL Stores

NoSQL vendor	Best for workload	Use cases
SimpleDB	Fast reads/ Not very large and complex datasets	Online games, logging, metadata indexing, click-stream logs, catalogs
MongoDB	Real time/ For complex and large datasets	Content and Product data management
Cassandra	Write often read less/ Very large datasets	Applications that need both real-time and analytics capabilities; Enterprise content search

written to disk or propagated to two or more replicas/slaves). By setting slaveOk, you can increase the read performance allowing read requests to be served by slaves. The same feature can be set per operation as well.

- Limitations - the maximum BSON document size is 16 megabytes. Indexed items can be no larger than 1024 bytes. A single collection can have no more than 64 indexes. Database and collection name, must be shorter than 123 bytes. There can be no more than 31 fields in a compound index. The dot character is not permissible in database names and in attribute names.

MongoDB has: *capped collections* and *tailable cursors* that allows MongoDB to push data to the listeners. These collections can be used to emulate triggers. Potential for injection attacks in MongoDB are high as it heavily utilizes JavaScript as an internal scripting language. Most of the internal commands available to the developer are actually short JavaScript scripts. It is even possible to store JavaScript functions in the database in the db.system.js collection that are made available to the database users. Because JavaScript is an interpreted language, there is a potential for injection attacks [OGOG$^+$11].

Table 4.6 gives a summary of the workloads and uses cases best suited for the NoSQL stores we have chosen for the validation of our approach.

The reserved words in the target databases are stored in a table, so that during migration it can be checked whether an identifier contains one of those reserved words, and in this case an error alert should be displayed to the user.

4.6 Modeling Practices

It is not logical to think of canonical modeling patterns that apply to all NoSQL categories because of the different data models that underpin them, but also because vendors of the same category provide different features. Still, there are some modeling patterns that apply to more than one category.

De-normalization- in contrast to a relational database, a NoSQL data store attempts to group similar data together (the aggregate) on disk to limit the number of seeks required to manipulate data. In this way it improves access times, thus achieving performance. It means copying the same data into multiple documents or tables. This pattern is referred to as de-normalization. Apart from high performance, another reason for de-normalization is retaining the integrity of data. For example: when making an invoice, it's risky to store keys that refer to Customer or Price information as they can change over time, and then you would lose the integrity of the invoice document as it was on the invoice date. The price to pay for de-normalization is that the application needs to know all the places where the data are duplicated and needs to correct them, synchronously or asynchronously. If the data is too big to be de-normalized, store them in separately and join at the application level. In this case, the parent-child integrity can be enforced at the data access layer.

De-normalization is a data modeling practice that applies to most of NoSQL stores. It is the key design principle in all data models in order to shape the data for simulating joins or for faster access. Different data stores implement different de-normalization techniques.

Embedding [column-family, document, key-value]: To de-normalize data, store two related pieces of data in a single document. This applies when you have "contains" relationships or you have one-to-many relationships where the "many" objects always appear with or are viewed in the context of their parent documents. The usage of this practice is limited by the update queries you will have, and by the size limitations of the value. In SimpleDB it can be implemented by a multi-value attribute.

Referencing is similar to the foreign keys in the relational world. Instead of embedding, store the ID of the referenced object to indicate the relationship between them. This can be used when modeling complex relationships. As there is no concept of joins, it is the client side application will issue more follow-up queries, i.e. more round trips to the server.

Valueless column [column-family databases] - sometimes also referred to as *Valuable key*: a key can itself hold a value, thus the column value may not be defined at all. For example, if you have a column family for holding the orders of costumers (OrdersByCustomer) then the customer ID will be row key of the new column family and the column names will be his order IDs, thus is not needed to store the column values at all. The Figure 4.2 illustrates the idea behind this practice.

Composite Column - build your column name out of one or more component values. This pattern fuses together two scalar values with a separator to use as a key.
Nested set [Document databases] is a practice used for modeling the hierarchical data set.
Rich documents - in document databases is a good practice to keep and design rich documents, so that data can be accessed together. The size of the document has to be kept into account while applying this practice.

In addition to good practices during modeling, there are also some not recommended practices derived from the industry use cases when using NoSQL solutions.

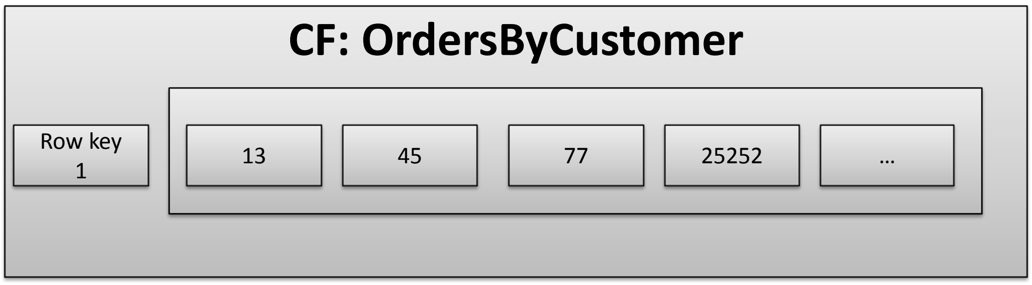


Figure 4.2: Valueless column in Cassandra

- Do not use super columns - [Column-family databases].
  They are a legacy design from a pre-open source release and have been criticized by the Cassandra community for performance issues and the lack of support for secondary indexes. This design was structured for a specific use case and does not fit most use cases. Super columns read entire super columns and all its sub-columns into memory for each read request. This results in severe performance issues. Additionally, super columns are not supported in CQL 3. The same "super column like" functionality (or even better) can be achieved by using composite columns [Pat]. "However super columns have been shown to impose a 10-15% performance penalty on reads and writes, so we have decided not to use them as our current object modeling tasks do not require their use." [SA12]. Use composite columns instead. Composite columns provide most of the same benefits as super columns without the performance issues [Ant].

- Motley types - [MongoDB databases]
  Even though it is allowed, avoid mixing of types in a single key's value within one collection, it makes the application logic complex, and makes BSON documents difficult to parse in certain strongly typed languages [Ban11].

- Fat documents - [Document databases]
  Don't think of a document as a real-life document, avoid to create such fat and complicated documents. You should make the difference between a rich and a fat document. Relationships between collections can exist. Keep the document size small (e.g. in MongoDB under 100KB per document unless unless storing raw binary data) and do not nest more than few levels deep [Ban11]. This helps with faster updates when bringing the document to memory and better readability of the documents.

One-to-one relationship: use *Embedding* pattern, unless you know the document size might exceed the target document storage limit or if you don't need joins. In this case use *Referencing*.
One-to-many: can be modeled using either Embedding or Referencing.
Many-to-Many relationships in RDBMS are modeled using mapping tables which store the pair of keys. In NoSQL store these tables can be modeled using the *Array keys* or the *Embedding* pattern.

The below are recommendations for the naming conventions:

- Preserving case-sensitivity - If you want to store the case sensitivity of your identifiers, in Cassandra you should quote them, because it stores them internally as lower case. Also while querying them, you should use the quotation marks. This should be kept in mind during query transformation. Also Amazon SimpleDB doesn't provide case-insensitive query. The query clause LIKE in MySQL does not distinguish between upper and lower case, while in PostgreSQL LIKE is for case sensitive searches, and ILIKE case insensitive searches. MongoDB collections names are case insensitive.
- Short column/attribute names as they are stored for every row/document and will result in increased storage. For example, Amazon S3 limits the length of object identifiers, so you must organize your data in a way that accommodates this and other Amazon S3 constraints. MongoDB has a limit to the length of the field to be indexed, which is 800 bytes.

Data type conversion:
SimpleDB zero-padding - when the data store support only lexicographical queries, like the case with SimpleDB, zero pad the value in order to achieve numerical comparison. MySQL has the ZEROFILL feature which can help to fill columns with zero at the beginning while selecting (they are stored without 0). You may alter the integer columns to ZEROFILL before exporting the data, so that lexicographical queries in SimpleDB will return correct numerical comparisons.
Cassandra has only Text and Varchar, so all the char, varchar and text data types should be converted to one of these two when migrating to Cassandra.

4.7 Concept

4.7.1 Schema Objects Mapping

NoSQL stores do not make any distinction between a database and a schema as PostgreSQL does. You will have to create a different database to accommodate each of your schemas. There is no one-to-one translation between a RDBMS schema and a NoSQL schema. What drives the schema model of the target data store, as stated many times before, are the application specific access patterns and the storage size limitations of the data store.

- Tables, Columns: for these objects a one-to-one mapping exist with a NoSQL object on the target store. Table 4.7 shows a summary of this mapping.
- Object Identifiers - the SQL standard is to use single quotes for data and double quotes for identifiers. MySQL allows to use single and double quotes for data and for identifiers, while PostgreSQL sticks to the SQL standard.
- Data types mappings - is the data type supported by the target store? In case the target data store doesn't have the data types being used by the application, additional techniques if possible can be used. For example, SimpleDB provides hints how to

Table 4.7: Logical Data Model mapping

Database Family	Database Vendor	Logical Data Model
RDBMS	MySQL	Schema/Database, Table, Row, Column
RDBMS	PostgreSQL	Schema, Table, Row, Column
Key-value	Amazon SimpleDB	Customer Account, Domain, Item, Attribute
Key-value	Amazon DynamoDB	Database, Table, Item, Attributes
Key-value	GAE Datastore	Account, Kind, Entity, Properties
Key-value	Windows Azure Table storage	Storage account, Table, Entity, Properties
Column-family	Apache Cassandra	Keyspace, Column family, Row, Column
Document-based	MongoDB	Namespace, Collection, Document, Field

convert the Date and Numerical types so that lexicographical queries return correct results. You can apply the zero-padding technique to achieve numerical-like comparison. For example, the string "10" comes before "2" in lexicographical order. If you zero pad to five digits, "00002" comes before "00010" and are compared correctly[Sim]. This doesn't work with negative numbers.

MySQL doesn't have a real boolean value. BOOL and BOOLEAN are synonyms for TINYINT(1). A value of zero is considered false. Nonzero values are considered true. PostgreSQL knows a real boolean data type; it can store for the value True many options, True, 't', 'true', 'y', 'yes', 'on', '1'; same for the False value. MySQL doesn't support array data type, while PostgreSQL does, it supports also multidimensional arrays. This is important when migration from PostgreSQL.

- Index types - a summary is given in Table 4.8.

MongoDB support Unique index but only on non-sharded collections.

After describing the schema objects of MySQL and PostgreSQL, and analyzing the features of NoSQL stores, the following are the schema objects that cannot be migrated to NoSQL. For some of them we provide the corresponding impact in the data access layer and the application layer on how to simulate them.

- Stored procedures and triggers because they are composed of SQL statements which is not supported by the NoSQL databases (partially supported). The security context like Definer, Invoker in which the stored procedures execute, is not possible to achieve in NoSQL databases because of missing role-based access functionalities.
- Indexes are usually created on-the-fly, as data is added. For example, SimpleDB indexes data automatically, you cannot define indexes.
- Referential integrity constraints
- Security privileges

Table 4.8: Indexes Mapping

Index type	MySQL	PostgreSQL	Migration to NoSQL
Primary	Yes	Yes	Yes
Secondary	Yes	Yes	Partially
Unique	Yes	Yes	Mostly no
Full-text	Yes	Yes	Mostly no
Prefix	Yes	N/A	Mostly no
Partial	N/A	Yes	No
Multi-column (Composite)	16 columns per index	32 columns per index	MongoDB (32 fields per index)
Function-based	N/A	Yes	No
Sparse	Yes	Yes	MongoDB
Spatial	Yes	Yes	Mainly document databases
Hash	Yes	Yes	Yes
Bitmap	N/A	Yes	Few
B-Tree	Yes	Yes	Yes

Table 4.9: RDBMS Schema Objects Migration Summary

Fully migrated	Partially migrated	Cannot be migrated
Tables, Rows, Columns	Data types, indexes, constraints (PK)	Constraints (FK, Not Null, Check, Unique), Stored Procedures, Triggers, Operators, Functions, Autoincrement

- Partitioned table definitions

There is no support for auto increment (Roll-Your-Own): this functionality it is not supported natively by the database, but it can be emulated; for example, invoking a function that set the value of the field by counting the existing ones. This will impact the application layer where this functionality has to be implemented (reading the last value either from a queue, a collection or a table, and then increment it). The idea of a globally sequential number is not a good idea in distributed systems as you have to make all participants agree and accept the evolution of the sequence, which prevents scalability.

4.7.2 Systems Trade-offs

Every data model is meant to solve different problems. Different NoSQL data models are meant for different application workloads and scenarios. To make the right decision when choosing the NoSQL store for your business problem, requires a very good low-level understanding of the target store. The plethora of NoSQL products with many different feature

sets, makes it almost impossible to recommend only by pure data model alone. Furthermore, the most obvious differences between NoSQL systems are between the various data models, and thus introducing a lack of apples-to-apples comparison. In addition, the number of emerging Cloud serving systems and the wide range of proposed applications, makes it difficult to understand the trade-offs between these systems and the workloads for which they are suited [CST$^+$10]. This questionnaire provides a support for the different properties that an application requires its data store to have by extending the existing questionnaire by Bachmann.
No system can be best for all workloads, and different NoSQL systems make different trade-offs in order to optimize for different applications [CST$^+$10].

It is very important to understand how the features of RDBMS and NoSQL technologies compare (or complement each other) to understand and make the right choice for the database that suits best the functional and non-functional properties of your application. Even though some databases seem to offer all the features an application might need, they don't have the appropriate industry or community support to allow for a confident choice for critical applications. You have to know the capabilities and constraints in details in order to pick up the right database.
The best schema design is a product of deep knowledge of the database and the application making use of this database. The new Cloud serving systems sacrifice complex query capabilities, transactional model, data integrity features for achieving high performance, elasticity and high availability. This was also the main motivation to the development of these new systems, the difficulty in achieving these three properties in traditional relational databases [CST$^+$10].

We have abstracted the at the high level the following trade-offs that these systems make to serve different application needs to incorporate in our methodology. For each of these trade-offs, specific properties of the Cloud data hosting solution will be affected.

- **Read performance versus Write performance**
  Every data store is designed to be best at read or write performance, but not at both of them. For example, column-family stores are designed for high throughput in writes, while key-value stores for high read throughputs. For small read/write requests the document databases are more suitable. Apache HBase is suitable when you need random, real-time read/write access to your big data. Most disk access patterns in MongoDB do not have sequential properties [10ga].

- **Durability versus Latency**
  We can distinguish between applications that have strong durability needs, where every transaction must be committed to disk and replicated in case of node failure and applications that are willing to relax these requirements in order to achieve the highest possible speed. This is made possible by the tunable persistence design approach of some databases. For example, in MongoDB, the *Write concern* allows you to trade write performance with knowing the status of the write [Wri]. If you can afford to lose some data (in case *mongod* process crashes or some network error occurs), e.g. in case you are doing high throughput logging, then you can set this feature to *low*. The same

with *durable_writes* feature in Cassandra. Redis gains amazing performance by caching writes in memory before committing to disk, in exchange for increased risk of data loss in the case of a hardware failure [RWC12].
This trade-off is mapped to the two following decisions:
Persistence approach choice for the data store: on-disk, in-memory or tunable durability. The on-disk writing technique flushes data to disk before acknowledging a request, writing to disk requires more time, thus you gain durability by trading off some performance. During the query transformation this should be taken into consideration, i.e. to set the appropriate settings when issuing write requests.

Replication strategy: synchronous versus asynchronous replication:
With synchronous replication (a node doesn't acknowledge a write request without propagating to all replicas), you can achieve higher durability. In asynchronous approach, the data store acknowledges the request and then propagates in the background to the replicas, thus reducing write latency.

- **Consistency versus Latency**
  This trade-off translates into the decision between synchronous or asynchronous replication strategy, as in the previous trade-off. A synchronous replication ensures replicas consistency, but potentially introduces high latency during writes/updates. The requirement for consistency level will drive the data model of your application. For example, knowing that document databases offer consistency at a document level and not across documents, this will drive the grouping of related data of a transaction in one document. In case of weak consistency, how will the application deal with versions? Because the conflict resolution is the responsibility of the application, you have to pay attention to the efforts you will put in resolving such conflicts. In a simple key-value model it is easy to compare versions to determine the latest value written to the system, but in systems that return sets of objects it is more difficult to determine what is the correct last updated set [Vogb].

- **Performance predictability versus Query features**
  For NoSQL stores it is difficult to predict throughput. Not all Cloud data stores offer this feature. Amazon DynamoDB provides fast and predictable performance with seamless scalability. It automatically spreads the data and traffic for a table over a sufficient number of servers to meet the request capacity specified by the customer in the Service Level Agreement (SLA) [Voga]. In order to yield predictable performance under all circumstances, it imposes limitation in querying the data. For example, the SimpleDB Query API, supports only comparison operators that map to efficient index access operations [AWS]. But, since the domains maintains a large number of indexes, its working set does not always fit in memory. This impacts the predictability of a domain's read latency, particularly as dataset sizes grow [Voga]. To simply look at the different benchmarks for some of the NoSQL databases is not a good idea. The document size, number and size of indexes, and the type of operation will all play a part in the actual numbers of benchmark results. You cannot judge on the performance based on the benchmark numbers with dummy workloads [Myt]. The most important thing you should do is to test their performance in the scenarios that are specific to you. Reasoning

about how a database may perform can help you build a short list, but the only way you can assess performance properly is to build something, run it, and measure it [SF12].

All the trade-offs explained above are translated into properties of Cloud Data Hosting Solution categories, thus extending the choices of the criteria when selecting a Cloud NoSQL data store. Also, depending on the chosen trade-off, the respective adaptations of the upper application layers will be shown to the user.

Looking at the industry use cases we abstract the following workloads that modern applications have to deal with.

- Read-heavy application
  Some characteristics of the data store that can be suitable for this workload: A Master/Slave deployment where the adding of slaves increases the read performance, like the case of Wordnik, an online dictionary on how words are used today [Wor]. Also, data stores that offers built-in caching layers or can integrate another caching layer on top of them, such as Memcached.
- Write-heavy application
  For write-heavy applications a master-less architecture like the one of Cassandra or Riak is more suitable. A system with a Master-slave replication is most helpful when you have a read-intensive workload, because in this case you are limited by the ability of the master to process updates [SF12].

  There are applications that deal with a mix workload, i.e. read and write heavy workloads. EBay recommends given its experience to keep read-heavy data separate from write-heavy data because they scale differently, even though this is a recommendation also for the world outside NoSQL [Pat].

4.7.3 Extending Cloud Data Hosting Solution Taxonomy for Functional Requirements

The Cloud Data Hosting Solution taxonomy is extended with many categories, and some existing categories are extended with more possible options from which the user can chose. Instead of the existing options: Schema and Schema Customizable, we change these criterias into *Schema definition* and *Schema flexibility level*.

- Schema flexibility level in order to identify the suitable storage model
  - Unstructured: Blob store
  - Minimally structured : key-value store
  - Semi-schematized: document-based, column-oriented or graph database. Document databases compared to column family stores offer higher flexibility because of the allowed nested structure.
- Querying features

  - SQL-like query language - so that SQL query reusability is high and less efforts are put into query transformation.
  - Aggregate queries support
  - RegExp queries support
  - Full-text search
  - Map-Reduce functions
  - Secondary indexes
  - Geo-spatial indexes which are useful for location-based applications.

- Transaction level support
  - Atomicity at aggregate level: single document, item, single entity group
  - Across many entities: Windows Azure Table Storage and Google App Engine Datastore support atomicity across multiple entities, still with the restriction of being within one partition. (WATS supports batch transactions across group entities, but within the same partition, with the limitation of 5 entity groups per transaction). Also Cassandra in its version of 1.2 introduced batch operations, but they are not real ACID, they do not support the isolation during the execution, i.e. one process can read the first operation after being executed, even though the rest is still in execution.

4.7.4 Extending Cloud Data Hosting Solution Taxonomy for Non-Functional Requirements

- Partitioning - the physical location of a table is based on the Partition Key (sharding key in MongoDB) selection. The partition key selection is one of most important architectural decisions for a Cloud application. If chosen properly it avoids the hot-spots and makes querying easier. Partitioning support can be:
  - Automatic: the datastore itself takes care of the partitioning of data by using consistent hashing usually. E.g. DynamoDB automatically partitions the data and workload over a sufficient number of servers to meet the scale requirements that you provide.
  - Manual: it is you that should manage the partitioning and re-partitioning of your data. E.g. SimpleDB uses hashing algorithms to create a uniform distribution of items among multiple domains. Still, you have to write your own sharding logic across domains in case they exceed 10GB of storage.

  In case of automatic partitioning, it is important to know which partitioning algorithm the data store supports because it has impact on the range queries.

1. Hash/Random Partitioning - routing is simple, e.g. Cassandra, MongoDB
2. Range/Order-Preserving Partitioning - is a prerequisite to support the range queries and scans (Cassandra, MongoDB, Azure Table). But this requires more overhead in terms of maintaining routing and configuration nodes.

- Durability - What persistence design model does the data store adopts, defines the durability.
  - Durable: when the data store writes only to disk, e.g. DynamoDB.
  - Non-durable: when the data store writes only in-memory, e.g. Redis.
  - Tunable durability: when both disk and in-memory methods are applied by the data store. Cassandra and MongoDB offer tunable durability. If the data store supports both techniques you can specify with each write request the durability level.
- CAP (Consistency, Availability, Partition Tolerance):
  Consistency Model: Currently it has Strong, Weak and Eventual. NoSQL stores offer also Client Tunable consistency, a customizable per operation consistency approach where the settings can be set per operation. Thus, each operation can result in being strongly consistent, or eventually consistent. As already explained this is a decision you make based on the priorities of the requests. Thus, the options to choose from are: strongly consistent, eventually consistent, tunable for reads, tunable for reads and writes.

  In case of partitioning: The choices for the data store, instead of *available* versus *not available*, should be between: consistent, available or tunable. As explained in 2.3, the choice between availability and consistency is much more fine grained and complex. Perfect availability and consistency is rare to achieve in the presence of partitions, but there is a range of flexibility for handling partitions and recovering from them.
- Replication - apart of the replication strategy which can be synchronous or asynchronous, another aspect of replication might be relevant for some use cases. A data store can offer the *rack-aware* feature which can keep replicas of data in different physical racks, which helps ensure uptime in the case of single rack failures (e.g. Cassandra supports this feature).
- Performance: Distinguish between read and write performance because the application might be read or write heavy. (e.g. event logging require fast writes, product catalog requires fast reads).
- Data Constraints: In addition to Max Item, Max Domain, Max Item/Row/File Number Per Instance, Max Size Per Instance (All Domains, All Tables, All Buckets), adding maximum *Index size*.
- Caching
  1. Built-in caching layer (e.g. Cassandra has built-in caching layer)

2. Support for integrating a caching layer such as Memcached, on top of the data store. In this case the data store offers Memcached API.

- Server-side scripting
  The MapReduce functions can be offered by the NoSQL store itself, databases such as CouchDB, MongoDB, Riak, offer this functionality. They give the user the possibility to define MapReduce functions in Java, Python, PHP, etc. and run them against the data.

- Community support
  Even though some databases seem to offer all the features an application might need, they don't have the appropriate industry or community support to allow for a confident choice for critical applications.

In Table 4.10 we give a summary of the categories that extended the Cloud Data Hosting Solution.

4.7.5 Application Layer Adaptation

The following are a set of adaptations needed when using a NoSQL database. Not all might apply to one case.

- Schema validator
  In the NoSQL world the data has to be parsed by the application and in case of a schema mismatch it is the application that should catch and throw these errors, not the database as was the case with RDBMSs. This should be taken into consideration when refactoring the application [SF12]. You might avoid this, by implementing a schema validation at the Data Access Layer as was the approach followed by Netflix [Ana]. This doesn't apply to all NoSQL systems and might also change in the future as new features will be added to the existing solutions. For example, a number of mature MongoDB object mappers provide an extra layer of abstraction on top of the basic language drivers, and you might consider using one on a larger project [Ban11].

- Outsource Complex Queries: "NoSQL data stores usually don't support complex queries using joins and aggregate functions on the data store level, but there are external services that could be used to defere complex queries on NoSQL data stores, e.g. using a MapReduce cluster. Using those services located closely to the data store can increase performance and reduce traffic compared to simulating complex queries on the client." For example, Amazon DynamoDB doesn't support MapReduce, but it can be integrated with Amazon Elastic MapReduce (Amazon EMR) [Gra].

- Joins
  If you embed in one document, or in a multi-value attribute the data that needs to be joined, there is no need to impact the application tier. The document databases are the best fit for this requirement as they allow nested objects. Otherwise, you have to rewrite multiple queries to get the join results.

Table 4.10: Cloud Data Hosting Solution Taxonomy

Category	Property	Data store
Caching	Built-in caching layer	Cassandra
	Integration with a caching layer	Riak, Drupal
Data partitioning	Manual	Cassandra
	Automatic	Riak, Drupal
Partitioning method	Random	Cassandra, Riak
	Order-Preserving/Range	Cassandra, DynamoDB, WSAT, MongoDB
Multi version conflict resolution	Client side	DynamoDB, Riak, Voldermort
	Server side	Cassandra, Riak
Full-text search	Built-in text-search	Riak, MongoDB
	Integration with text-search engines such as Lucene, Solr	Cassandra
Performance	Predictable	SimpleDB, DynamoDB, WSAT
	Non predictable	Cassandra, MongoDB
Consistency model	Strongly consistent	WSAT
	Eventually consistent	Riak
	Tunable for reads/writes	Cassandra, MongoDB
	Tunable for reads	SimpleDB, DynamoDB
Querying	SQL-like query language	MongoDB, SimpleDB, Cassandra
	MapReduce	Cassandra, MongoDB, Riak, Couchbase
	Materialized views	CouchDB, VoltDB, Couchbase
Hybrid deployment support	N/A	Cassandra
Sharding	Auto	MongoDB, DynamoDB, Couchbase, Riak, WATS
	Manual	SimpleDB
Replication	Synchronous	Cassandra, SimpleDB, DynamoDB, GAE HRS, Membase
	Asynchronous	Cassandra, MongoDB, GAE Master-slave datastore

- Stored procedures/functions
  The stored procedures and triggers, which are an important part of a RDBMS, cannot be migrated to NoSQL systems. You can emulate them by making the appropriate changes in your code. For example, the corresponding stored procedures of the relational world, are JavaScript programs in MongoDB and CouchDB (in the latter one also known as views). All the NoSQL databases with server-side scripting support are suitable if you need to implement such functionality in the application.

- Integrity Constraints: Not Null, Check, Unique can be implemented at this application tier.

- Conflict resolution - applicable when you chose the data store to be available in case of partitions. The inconsistencies that arise when systems chose availability over consistency can be solved by using causal ordering mechanisms like vector clocks and application-specific conflict resolution procedures. These conflicts rises when multiple nodes can write at the same time to the same object, resulting in different versions, i.e. inconsistent replicas. The data store might solve this conflict resolution during reads, e.g. the mechanisms *last write wins* in which the data store simply ignores the earlier versions and returns the last updated one to the client. This removes the need for a round-trip to the client and simplifies the API. Another approach at the application layer might be taken, Programmatic Merge, where all versions are returned to the application and it can decide how to solve the conflict, for example the DynamoDB choses to merge the conflicted shopping carts.

- Transactions
  If the target data store does not support atomic transactions at multiple documents level, and your application needs them, implement the transaction management logic in your application code. This means managing locking with custom logic. Use "Time Stamps" to manage currency of data, unless this is not provided by the database itself (e.g. Azure Table Storage provides it by default for each item).

- Autoincrement, Sequence, TimeStamp
  Most of the NoSQL stores don't provide the auto increment feature, if you want a unique sequence number you have to generate it at the application code. DynamoDb doesn't have the feature of getting a server timestamp, you have to issue it from your application code.

- Triggers
  There is no native support for triggers in NoSQL. Different approaches might be taken that need a change at the application code. For example, if the data store has PubSub features, the client that is writing a value might also publish that the value changed to a channel. In this way the other interested application parts might read from that channel and take the respective actions. This is an asynchronous trigger approach. MongoDB also offers *capped collections* and *tailable cursors* that allows MongoDB to push data to the listeners that can be considered triggers equivalents.

- Secondary indexes computation logic
  When the data store has no or limited support for secondary indexes, is the application layer that has to implement the logic, e.g. when using Riak, the application has to tag the objects being stored with the metadata that it wants to run queries on. Also, every time you write to the database, you also need to maintain your index.
- Data Encryption
  If the application requires storage encryption, to address this requirement one approach is to encrypt field-level data within the application layer using the requisite encryption type that is appropriate for the application. Another option is to use third-party libraries that provide disk-level encryption as part of the operating system kernel. 10gen has a partnership with Gazzang, which has a certified product for encrypting and securing sensitive data within MongoDB.
- Integration logic with search engines - In case there is no internal integration with a search engine, there is a need for writing code to interface with search systems like Solr, Lucene or ElasticSearch[1].

Sometimes might be need also manual interventions, e.g. Cassandra 1.2 introduced batch operations, executing as atomic operations in the sense that if some part succeed, the batch will succeed. But, if the client and coordinator fail at the same time — because the client is an application server in the same data center, and suffers a power failure at the same time as the coordinator — then there is no way to recover other than manually crawling through the records and reconciling inconsistencies.

4.7.6 Data Access Layer Adaptations

- Schema validation
  The schema validator can also be implemented at the DAL, like the case of Netflix. Because of the lack of domain-specific schema validation in NoSQL stores, querying a misspelled attribute, you might not get an error at all. In order to avoid the schema mismatch problem, the schema validation can be implemented in a common data access layer [Ana].
- Data integrity enforcement
  As integrity constraints like foreign keys are not supported by NoSQL stores, the parent-child integrity can be also enforced at the data access layer.
- Case sensitivity - in case the data store is case sensitive, like SimpleDB, the updates with wrong case will fail silently [Ana]. Stick to either upper or lower case of identifier names and in this case the data access layer normalizes (e.g. TO_UPPER or to_lower) all access to attribute and domain names in case of SimpleDB, collection names in case of MongoDB, etc.

[1]ElasticSearch: http://www.elasticsearch.org/

5 Design

In this chapter we present the architectural solution taken into account to build the system which fulfills the requirements specified in Chapter 4. This chapter describes an architectural and technological solution for the concepts and specified system requirements of Chapter 4. Section 5.1 gives and overview of system architecture and in Section 5.2 we provide an overview of the design of the migration component.

5.1 System Overview

Figure 5.1 shows an overview of the system architecture. The system has still two components, the decision support system and the migrations component which are enhanced and extended based on the requirements specified in Chapter 4. The methodology is realized at the decision support system, and the migration component is extended with another database source adapter and three target adapters.

5.2 Architectural Overview

The component decision support system shown in Figure 5.1 is decomposed in the Figure 5.3. Figure 5.2 shows the first steps of the decision support system. As we extend and existing methodology and tool, we focus only on one migration scenario, i.e. from RDBMS to NoSQL. After creating a project, if the user choses another migration scenario, he will go through the existing methodology developed by Bachmann.

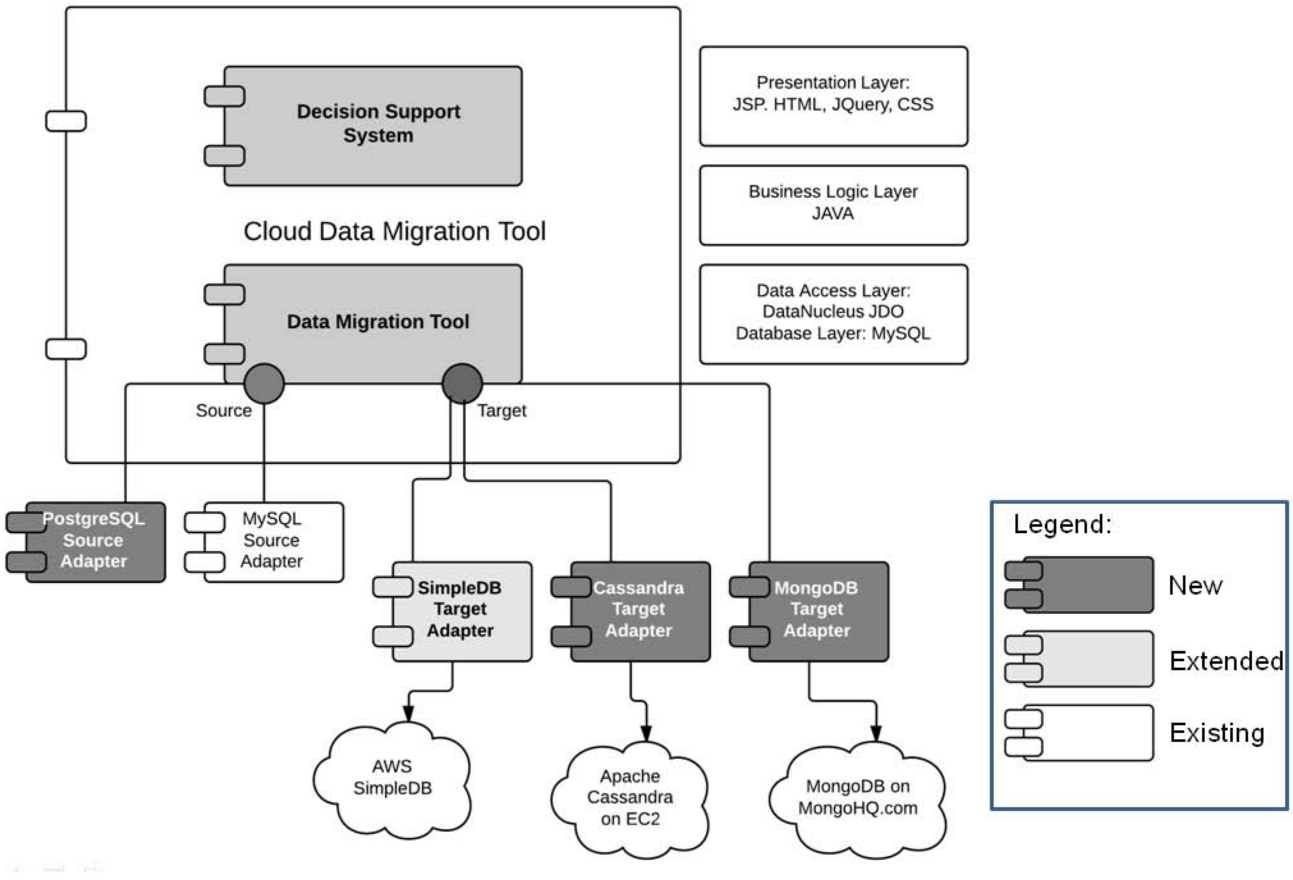


Figure 5.1: System Architecture Overview

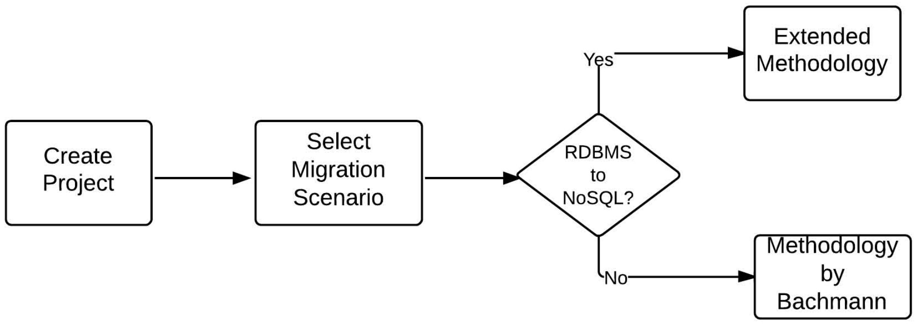


Figure 5.2: Data Migration Methodology - First Steps

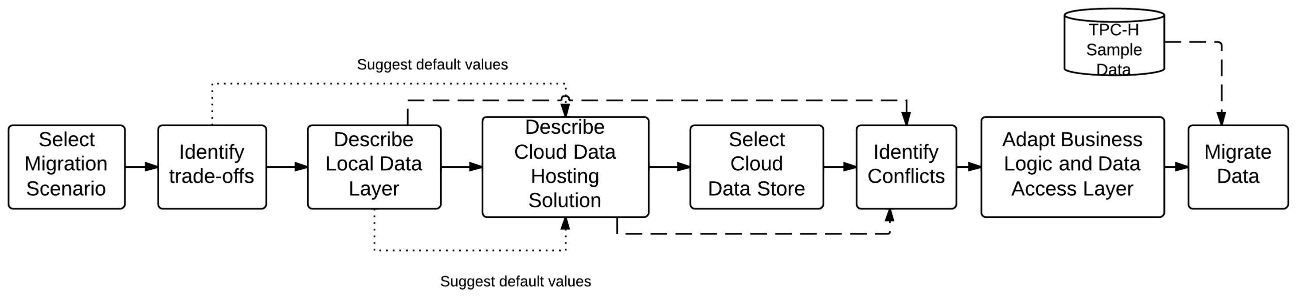


Figure 5.3: Extended Methodology for RDBMS to NoSQL Migration Scenario

If the migration scenario is from RDBMS to NoSQL, the extended methodology steps are described in Figure 5.3. The trade-offs that NoSQL systems make, analyzed in Section 4.7.2, are incorporated into the second step of the methodology. Describe local data layer is the *Step 4a* in Bachmann's methodology which describes the local database layer that the application is using. In case of our migration scenario, we extend the list of properties for the two existing criteria, and add *Querying* category. Moreover, we move this step before the user describes the CDHS, in order to suggest default values for the criterias based on the selection of the local database requirements. What type of indexes does your application use? Are range queries utilized by the application? The operating system on which the local database is running (Windows, Unix). If the OS is Unix, the database and table names are case sensitive. This has to be preserved during migration, e.g. utilizing double quotes.

Adapt data access layer and upper application layers if needed - when migrating to NoSQL there will always be adaptations needed at these two layers because of the missing functionality and different query languages of the source and target databases.

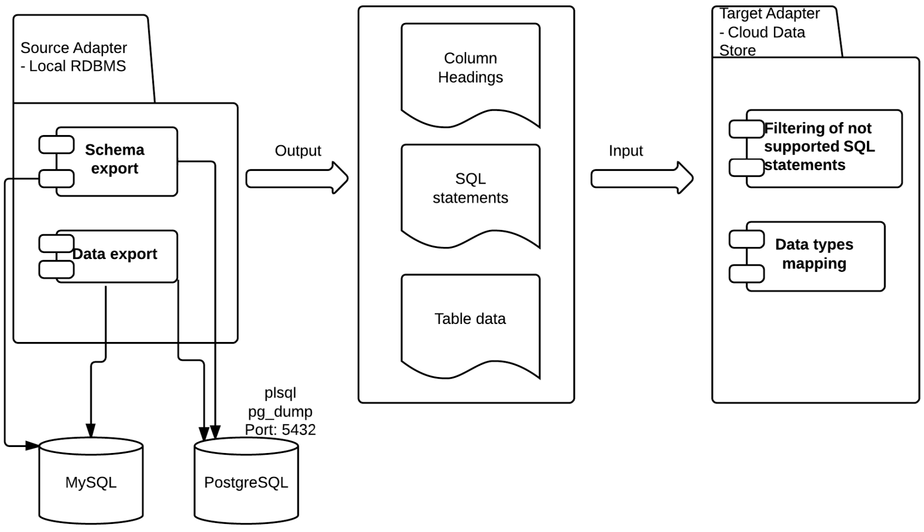


Figure 5.4: Migration Component Architecture

As the MySQL source adapter was developed by Bachmann, we preserve the same output format for PostgreSQL adapter, so that PostgreSQL adapter can be used for the other migration scenarios too, see Figure 5.4. For each table in the PostgreSQL source database, we generate three files with the same name and different types:

- .csv file that contains the column names of the table
- .txt with the data
- .sql stores the SQL statements for the schema elements creation

Given the fact that there are not referential integrity constraints in NoSQL, such as foreign key declaration statements, we ignore these from the content of the *.sql* files.

6 Implementation

We selected three NoSQL databases of different data models with the purpose to cover the three main categories of the spectrum of NoSQL databases, and also leveraging the NoSQL options of the big Cloud vendors. In this chapter we describe the implementation of the decision support system and the migration component (i.e. the source and target database adapters) and the challenges faced during the implementation phase.

6.1 Realization and Validation

To validate our approach we use the schema and the sample data from TPC-H benchmark [Tra]. TPC-H is a benchmark that illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. In our work we use this benchmark only for the purpose of having standardized relational schema (Figure 6.1) to migrate to a NoSQL database. Given the fact that this benchmark is designed to examine large volumes of data, it fits well with the purpose of why do organizations consider migrations of applications to NoSQL databases.
We loaded the sample data provided by TPC-H in the two source databases, MySQL 5.6 and PostgreSQL 9.2. in a system running on Ubuntu 12.4 operating system. *dbgen -h* command gives us 1GB of data divided into eight tables. After the loading of the data, the integrity constraints between tables specified in the file *dss.dri* were applied.

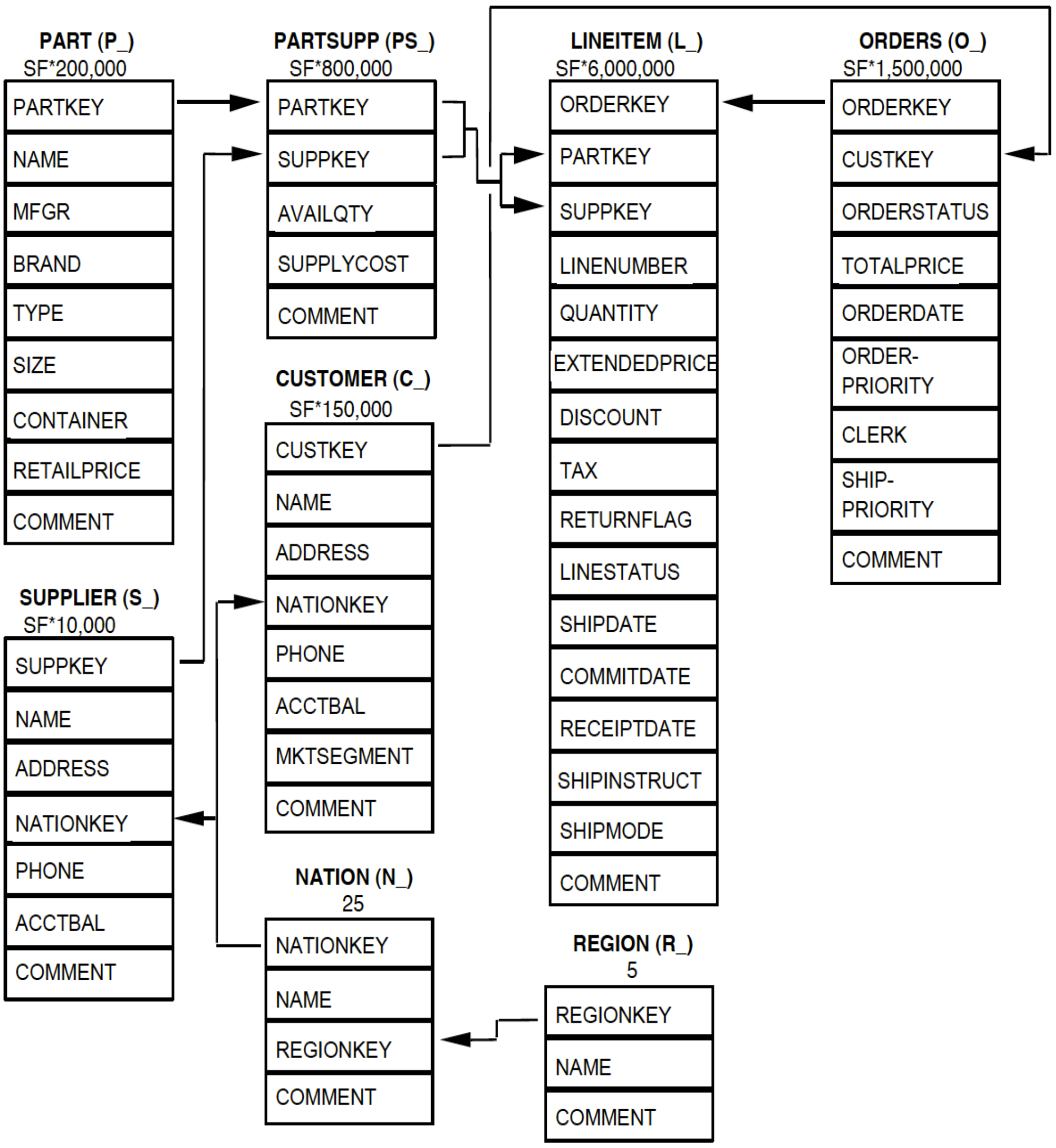


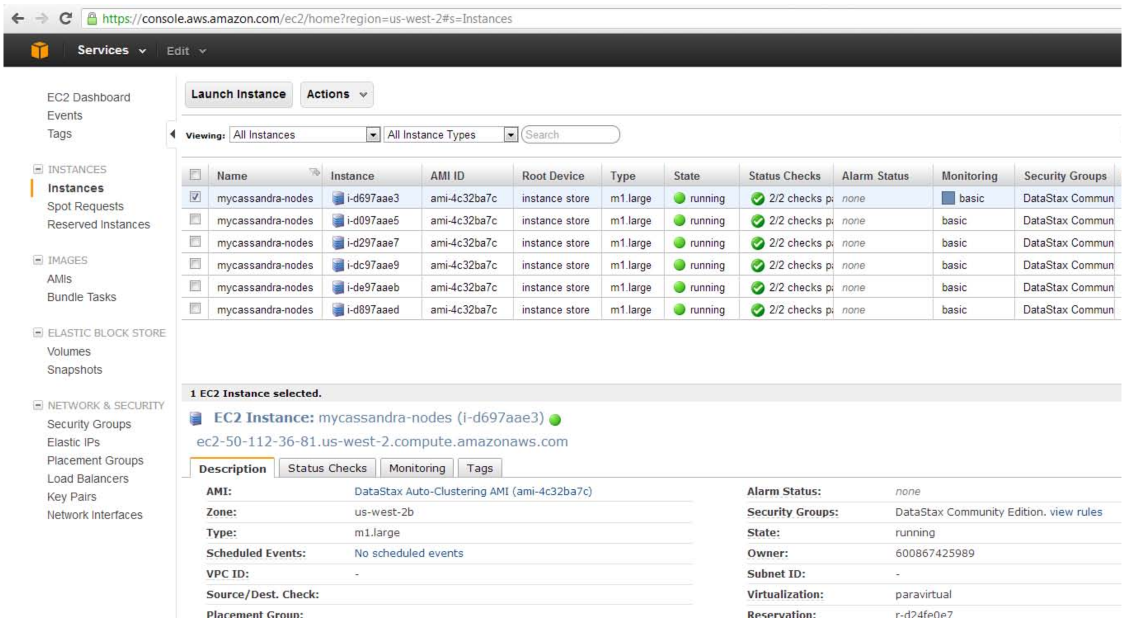
Figure 6.1: TPC-H Database Schema

PostgreSQL Source Adapter:
pg_dump is the utility, that uses the default port 5432 for backing up a PostgreSQL database. With the option -n schema it selects both the schema itself, and all its contained objects. The source adapter of PostgreSQL also takes the schema name as a parameter, because within a database there can be more than one schema, as opposed to MySQL. While for getting the SQL statements we use *psql* command We implement this adapter based on the MySQL adapter, i.e. following the same output format, developed by Bachmann with the purpose that our PostgreSQL adapter can be used by his migration scenarios as well, meaning when transitioning to relational databases in the Cloud. For each Cloud NoSQL data store, we save the list of reserved words in a table, and during migration check if each of the object being migrated, like database name, table and column name are not part of this list. The data types that need to be replaced, are also saved in a table, which has two fields only, data store name, data type name. Based on the data source selected, we read the respective data types, for example *Varchar* type has to be mapped to *String* type in the NoSQL database.

As there is no UI for entering the data for example, for data access layer and business logic layer adaptations, entering new CDHS categories, for updating existing ones, etc., the data were entered directly through MySQL Workbench SQL Development component. A development of a GUI for such purposes can be considered for future work.

DataNucleus Access Platform is an open-source implementation of several Java persistence standards, provides support for interfaces of the persistence frameworks: Java Data Objects (JDO) and the Java Persistence API (JPA). There are two approaches for persisting application data in a NoSQL system, either through the Persistence API or a Mapping framework like DataNucleus. The former one is tailored to a specific NoSQL solution. DataNucleus is unique in that it supports all standardized persistence APIs for a very wide range of datastores, thus having the needed functionality for the CDMT which supports different data storage types. DataNucleus supports persisting and retrieving objects to and from MongoDB datastore (using the datanucleus-mongodb plugin), but in this work we used the Java driver for MongoDB. It has support also for Cassandra. Amazon SimpleDB:
We reuse the SimpleDB adapter from Bachmann, but we create the target attributes of the items taking into consideration the assumptions we make about one-to-many relationships, i.e. embedding or referencing, not just a simple mapping of one table column name to one item attribute name.

Apache Cassandra 1.2 on Amazon EC2:
We created an account on Amazon EC2 using the educational grants provided by Amazon for students. With official support from Datastax, we use Amazon EC2 to set up a Cassandra cluster provided the Amazon Machine Image (AMI) by Datastax Community edition [Data]. The configuration of the cluster is as follows: –clustername myDSCcluster –totalnodes 6 –version community.
We start with creating an EC2 security group for Datastax Community Edition which allows to choose which protocols and ports are open in our cluster. EBS volumes are not recommended. Initially we found difficulties in launching the cluster in some availability zones, like Eu-Ireland and Asia Pacific, while it was successful using US-West Oregon zone. In Cassandra data volumes, EBS throughput may fail in a saturated network link and adding capacity by



https://console.aws.amazon.com/ec2/home?region=us-west-2#s=Instances

Viewing: All Instances | All Instance Types

	Name	Instance	AMI ID	Root Device	Type	State	Status Checks	Alarm Status	Monitoring	Security Groups
✓	mycassandra-nodes	i-d697aae3	ami-4c32ba7c	instance store	m1.large	running	2/2 checks p	none	basic	DataStax Commun
	mycassandra-nodes	i-d097aae5	ami-4c32ba7c	instance store	m1.large	running	2/2 checks p	none	basic	DataStax Commun
	mycassandra-nodes	i-d297aae7	ami-4c32ba7c	instance store	m1.large	running	2/2 checks p	none	basic	DataStax Commun
	mycassandra-nodes	i-dc97aae9	ami-4c32ba7c	instance store	m1.large	running	2/2 checks p	none	basic	DataStax Commun
	mycassandra-nodes	i-de97aaeb	ami-4c32ba7c	instance store	m1.large	running	2/2 checks p	none	basic	DataStax Commun
	mycassandra-nodes	i-d897aaed	ami-4c32ba7c	instance store	m1.large	running	2/2 checks p	none	basic	DataStax Commun

1 EC2 Instance selected.

EC2 Instance: mycassandra-nodes (i-d697aae3)

ec2-50-112-36-81.us-west-2.compute.amazonaws.com

AMI:	DataStax Auto-Clustering AMI (ami-4c32ba7c)	Alarm Status:	none
Zone:	us-west-2b	Security Groups:	DataStax Community Edition. view rules
Type:	m1.large	State:	running
Scheduled Events:	No scheduled events	Owner:	600867425989
VPC ID:	-	Subnet ID:	-
Source/Dest. Check:		Virtualization:	paravirtual
Placement Group:		Reservation:	r-d24fe0e7

Figure 6.2: Cassandra Cluster Setup View

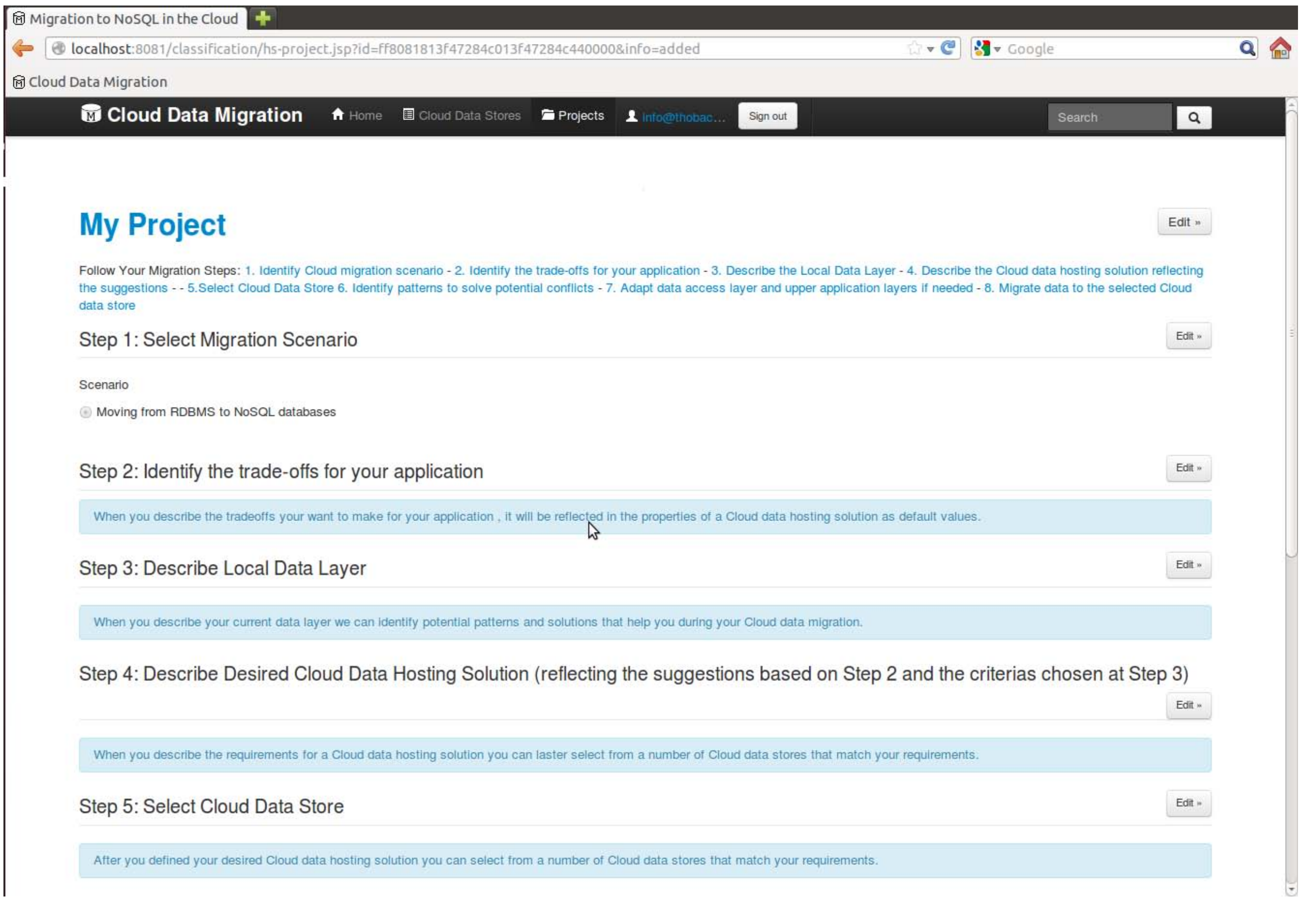
increasing the number of EBS volumes per host does not scale. A *.pem* file is a private key created and saved locally in order to login to the Cassandra community cluster. In order to access the Cassandra system from the secure shell of our Ubuntu OS, we had to change the file permissions with *chmod 400* in order to be able to use it for authentication.

Cassandra clients:
There are many high-level Cassandra clients for Java, Scala, Ruby, Python, Perl, PHP, C++, and other languages, written as conveniences by third-party developers. It is important to chose a client that will stay up to date with Cassandra's updates [Hew11]. Thrift is low-level API, it is a code generation library for clients in Java, C++, Erlang, Haskell etc. Many drivers encapsulate Thrift in their implementations. We use the Astyanax driver provided by Netflix on github repository. After cloning and building the driver, we use the generated JAR files: astyanax-cassandra-1.56.43-SNAPSHOT.jar, astyanax-core-1.56.43-SNAPSHOT.jar, astyanax-thrift-1.56.43-SNAPSHOT.jar, and in addition two other JARs: cassandra-thrift-1.2.0-rc1.jar and cassandra-all-1.2.0-rc1.jar downloaded separately. We can connect either to the cluster through the command line and using Command Line Interface (CLI) with the command: *cassandra-cli -host localhost -port 9160* or we can connect to a specific node in the cluster using the private IP of the node, e.g. *cassandra-cli -host 10-255-1-202 -port 9160*.
Cassandra is more strongly typed compared with the other two databases.

```
CREATE COLUMN FAMILY Customers
WITH comparator = UTF8Type
AND key_validation_class=UTF8Type
AND column_metadata = [
{column_name: C_NAME, validation_class: UTF8Type}
{column_name: C_ADDRESS, validation_class: UTF8Type}
{column_name: C_NATIONKEY, validation_class: UTF8Type}
{column_name: C_PHONE, validation_class: UTF8Type}
];
```

Listing 6.1: Cassandra Column Family Creation

MongoHQ[1] provides MongoDB as a service with the limitation of 2GB of data which is enough for our validation approach. It itself is hosted on Amazon EC2. MongoHQ provides users with an API token that should be passed to all the URLs during the interaction with database. After starting using the REST interface, we switched to using the Java driver as it provides better and faster access even for basic operations, an advice given to us by the support team of MongoHQ. Compared to the other two databases, MongoDB is easier and has a shorter learning curve. It consists in only one JAR java driver and a set of simple APIs for creating *collections* and inserting JSON documents into them.


Migration to NoSQL in the Cloud
localhost:8081/classification/hs-project.jsp?id=ff8081813f47284c013f47284c440000&info=added
Google
Cloud Data Migration
Cloud Data Migration
Home
Cloud Data Stores
Projects
info@thobac...
Sign out
Search
Edit »

My Project

Follow Your Migration Steps: 1. Identify Cloud migration scenario - 2. Identify the trade-offs for your application - 3. Describe the Local Data Layer - 4. Describe the Cloud data hosting solution reflecting the suggestions - - 5.Select Cloud Data Store 6. Identify patterns to solve potential conflicts - 7. Adapt data access layer and upper application layers if needed - 8. Migrate data to the selected Cloud data store

Step 1: Select Migration Scenario

Scenario

Moving from RDBMS to NoSQL databases

Step 2: Identify the trade-offs for your application

When you describe the tradeoffs your want to make for your application , it will be reflected in the properties of a Cloud data hosting solution as default values.

Step 3: Describe Local Data Layer

When you describe your current data layer we can identify potential patterns and solutions that help you during your Cloud data migration.

Step 4: Describe Desired Cloud Data Hosting Solution (reflecting the suggestions based on Step 2 and the criterias chosen at Step 3)

When you describe the requirements for a Cloud data hosting solution you can laster select from a number of Cloud data stores that match your requirements.

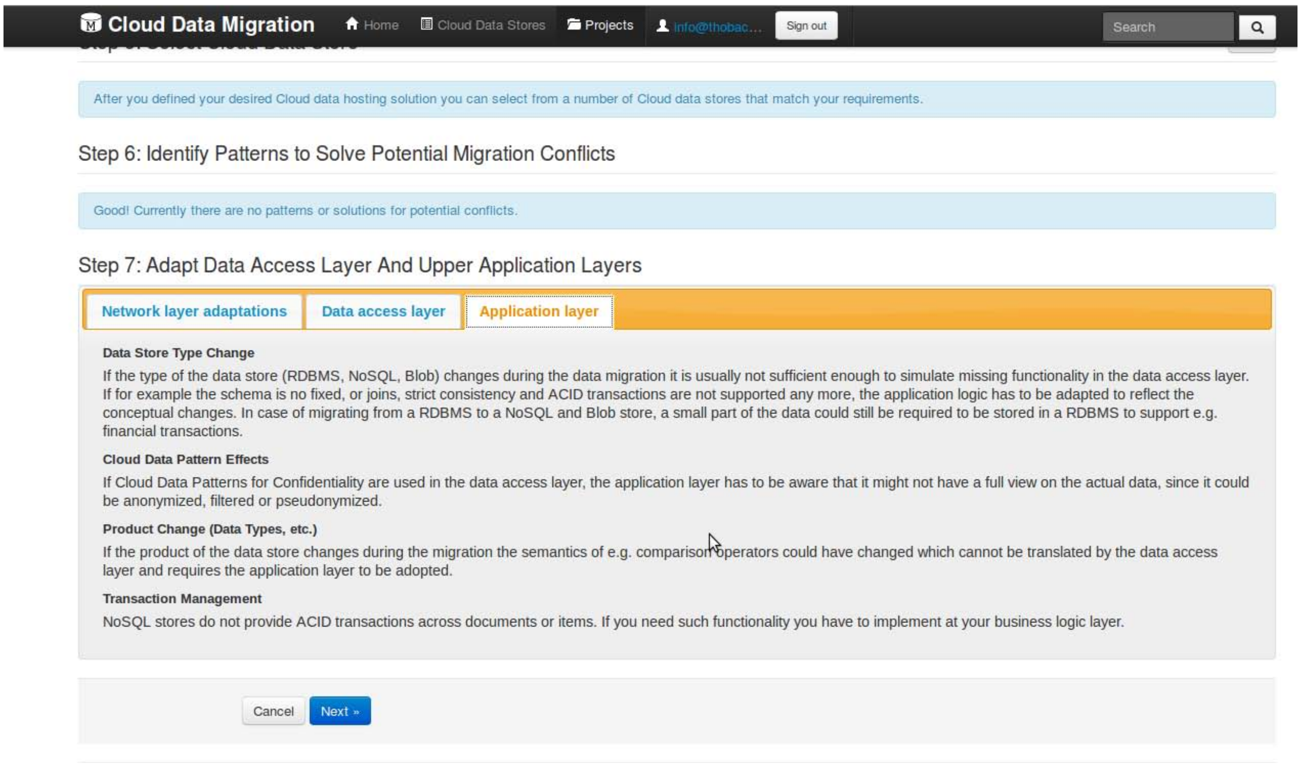
Step 5: Select Cloud Data Store

After you defined your desired Cloud data hosting solution you can select from a number of Cloud data stores that match your requirements.

Figure 6.3: Decision Support System Screenshot

For easier comparison with the existing methodology steps, refer to Section 4.1.

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



Cloud Data Migration | Home | Cloud Data Stores | Projects | info@thobac... | Sign out | Search

After you defined your desired Cloud data hosting solution you can select from a number of Cloud data stores that match your requirements.

Step 6: Identify Patterns to Solve Potential Migration Conflicts

Good! Currently there are no patterns or solutions for potential conflicts.

Step 7: Adapt Data Access Layer And Upper Application Layers

Network layer adaptations | Data access layer | Application layer

Data Store Type Change

If the type of the data store (RDBMS, NoSQL, Blob) changes during the data migration it is usually not sufficient enough to simulate missing functionality in the data access layer. If for example the schema is no fixed, or joins, strict consistency and ACID transactions are not supported any more, the application logic has to be adapted to reflect the conceptual changes. In case of migrating from a RDBMS to a NoSQL and Blob store, a small part of the data could still be required to be stored in a RDBMS to support e.g. financial transactions.

Cloud Data Pattern Effects

If Cloud Data Patterns for Confidentiality are used in the data access layer, the application layer has to be aware that it might not have a full view on the actual data, since it could be anonymized, filtered or pseudonymized.

Product Change (Data Types, etc.)

If the product of the data store changes during the migration the semantics of e.g. comparison operators could have changed which cannot be translated by the data access layer and requires the application layer to be adopted.

Transaction Management

NoSQL stores do not provide ACID transactions across documents or items. If you need such functionality you have to implement at your business logic layer.

Cancel | Next »

Figure 6.4: Decision Support System Screenshot (contd)

6.2 Discussion and Lessons Learned

At the heart of NoSQL data modeling is the principle of de-normalization. Data modeling and consequently the schema design is driven by the application specific query patterns, thus it can't be thought of an automatic way of moving from RDBMS to NoSQL in case of complex relational schemas and unknown application query patterns. This is the reason and a great finding of our work, why it is not feasible to have a plug-in mechanism for mapping from a concrete relational database to a NoSQL store.
Data integrity is not ensured by the NoSQL database, as there are no referential integrity constraints, cascade update/delete, that are the proven mechanisms of RDBMS to preserve the integrity of data. This makes NoSQL stores not suitable for applications that have high requirements for data integrity. Even though, also if you use the MyISAM engine of MySQL it does not ensure data integrity and transaction support. In traditional RDBMS, the data can be retrieved using any query tools. In NoSQL databases, there are query tools but is the application the owner of the data and serves them using services.
When considering adoption of NoSQL databases, there are many barriers that might prevent the companies from doing so. They have to be carefully considered based on the use case. One of these are the security features when considering a NoSQL for the Cloud. Most of the NoSQL stores were not designed with security in mind and exposing those instances in the Cloud might be too risky. Even though some might offer more security features than others, still they are not in a mature state. Also, the community and commercial support is a factor

that influences decisions, especially for production systems.

7 Outcome and Future Work

In this final chapter we summarize the results and contributions of our work and provide recommendations for future work.

7.1 Recommendations

Many successful NoSQL adoptions are an example of polyglot persistence. You should not think of NoSQL databases as a magical hammer for all the problems. They are meant for specific use case and make trade-offs to achieve those objectives. Design your data model such that operations are idempotent. In an eventually consistent and fully distributed system idempotent operations can help a lot. They allow partial failures in the system, as the operations can be retried safely without changing the final state of the system. It can allow you to work with eventual consistency without causing data duplication or other anomalies [Pat].
In order to achieve good performance, building a caching layer on top of the NoSQL store back-end, in case it doesn't have a built-in, has been proved to be a successful approach and is being applied by Web social applications, especially gaming sites. Also using key-value store of Oracle NoSQL for gaming application and using RDBMS for ad-hoc analytics, is another example of a polyglot persistence [Oraa].
You can make use of queues to separate writes to the database and maintenance of indexes in case of no indexing support from the database, thus you shorten the response time as you don't have to wait for the index to be created or updated. For those NoSQL stores that have limited search functionality, an integration with other search engines like Apache Solr or Lucene, is a good practice. And this might be useful for media and content based applications.
In MongoDB, it takes time and resources to deploy sharding, and if your system has already reached or exceeded its capacity, you will have a difficult time deploying sharding without impacting your application [Monb]. So the recommendation here is to think of sharding during your data modeling, think of which collections you want to shard and which are the sharding keys.
When using MongoDB as a Service from MongoHQ, it is recommendable based on advices received from the MongoHQ support during communication with them, in case your application is in Java to use the MongoDB driver for Java and not the REST APIs, because even for basic interactions using the driver is going to provide faster and better results than going over HTTP. MongoDB also recommends to use replica sets instead of master/slave setup to achieve replication for production environments. In NoSQL stores it is very important how you decide to split the data. This is a decision you should pay a great attention since the beginning, in order to avoid impacting negatively the performance and operations. A bad partitioning key can result in "hot spots", i.e. certain machines responsible for serving the

biggest amount of data and requests. One of the means to avoid these hot spots is consistent hashing which distributes data evenly across nodes.
If you want the query transformations efforts to be minimal, you can use a NoSQL that support SQL-like query language (a subset of SQL), like SimpleDB, Cassandra (with its CQL), and making the necessary changes at the data access layer. If your application uses range queries, chose a database that supports range partitioning, so that you avoid crossing partition or servers boundaries that will result in decreased performance.
Use MapReduce in a controlled manner and outside your peak production hours, cause it might effect your performance. Successful use cases of using NoSQL in the Cloud, show that they migrated also the business logic layer to the Cloud. If you put into the same AWS availability zone, the application and the database layer, you reduce the communication latency of the application with the database.

7.2 Contributions and Future Work

In this section we summarize the main contributions of our work with respect to the research questions specified in Section 1.4. In this thesis we provide a comprehensive analysis of NoSQL databases characteristics, commonalities and differences with RDBMS. Suitable applications for using NoSQL are applications that do not have high requirements for ACID transactions, data integrity, complex queries. We provide a summary about the relational schema objects that can be migrated and for those that cannot be migrated, the respective impact on upper application layers is provided.
General trade-offs are involved when moving from RDBMS to NoSQL, related to transactional behavior, querying features, data integrity, and trade-offs that NoSQL databases make to serve certain application workloads. We incorporate these trade-offs as part of the questionnaire. As NoSQL stores were driven by different needs than those of RDBMSs, we provide recommendations on how to refactor the other application layers when moving from RDBMS to NoSQL, i.e. data access layer and business logic layer. We support the decision process, migration and the refactoring of the application by extending the Cloud Data Hosting Solution with properties for NoSQL databases; by extending a methodology based on a questionnaire for migration of the database layer to the Cloud. Moreover, in order to support the migration we extended a prototype by offering support for migration from PostgreSQL and MySQL to AWS SimpleDB, MongoDB and Apache Cassandra.

We make implicit assumptions with regard to data modeling in the target store, i.e. how the data is accessed. For some of the one-to-many relationships we save only the ID of the referenced object, and for others we embed the values in the parent document. As future work we propose a more complicated data set with all types of indexes and more data types to be used for migration data. Also, finding a way to materialize the best modeling practices of the selected NoSQL store before migrating the data.

7.3 Conclusion

Migrating from RDBMS to NoSQL is a creative process and is difficult to be automated, because there are no standards that these databases adhere to. One has to look beyond a given database and understand how the application uses the data, identify which data sets need to be accessed fast, and which uses are not frequent and perhaps not even required. Then you need to see how the target database can be used to best support these uses. For achieving high performance and high availability, the NoSQL stores make trade-offs, such as no ACID transactions, most of them do not support ad-hoc and complex queries. Data modeling in RDBMS is consistent because the theory on which it is based is well established and implementation is standardized, while NoSQL stores are not in mature state yet and there are many proprietary implementations. In order to judge about a NoSQL store performance, the document (item) size, number and size of indexes, and the type of operation will all play a part in the actual numbers of benchmark results. You cannot judge on the performance based on the benchmark numbers with dummy workloads. The language drivers have an impact on the data types you can utilize. Different drivers for the same NoSQL database do not support the same data types. The decision depends on your data sets and the rule number one to NoSQL design is to define the query scenarios first. Once you really understand how you want to query the data, then you start exploring the various NoSQL solutions in the market. The default unit of distribution is key. Therefore you need to remember that you need to be able to partition your data between the nodes effectively in case your application deals with big data, otherwise you will end up with a horizontally scalable system, but still with all the work being done by one node. Because of lack of standards which is normal for all new technologies until they mature, interoperability is a concern when thinking to moving applications from one NoSQL system to another. Some companies are developing their proprietary system and some of them are open-source. Also, the differences in data models of the different systems will make the portability of applications to a different system a very complicated process. There is no magic hammer for all the problems, you have to find for each one the right persistence approach.

8 Appendix

Because we did an analysis of other NoSQL databases, apart of those used for validation, and the knowledge gained from them was incorporated in our questionnaire, we put the analysis in this appendix chapter.

8.1 MySQL and PostgreSQL Data Types Summary

Table 8.1: PostgreSQL 9.2 Data Types

Type	Subtypes	Description
SQL types	int, smallint(2), real, double precision, char(N), varchar(N), date, time, timestamp, and interval,	variable-length with limit
Numeric	smallint(2), integer(4), bigint(8), decimal, numeric, real(4), double precision(8), smallserial(2), serial(4), bigserial(8)	variable-length with limit
Monetary	money(8)	currency amount
Character	char(n), varchar(n), text	variable-length with limit
Binary	bytea	variable-length,1 or 4 byte binary string
Date/Time	timestamp(8), date(4), time(8 or 12)	both date and time (no time zone)
Boolean	boolean	True, ’t’, ’true’, ’y’, ’yes’, ’on’, ’1’ or False, ’f’, ’n’, ’0’
Enumerated	enum (4)	static, ordered set of values
Geometric	money(8)	currency amount
Network Address	cidr, inet, macaddr	store IPv4(7 or 19 byte), IPv6(7 or 19 byte), and MAC addresses (6 byte)
Bit string	bit(n), bit varying(n)	strings of 0’s and 1’s, 8 byte
Text search type	tsvector	represents a document in a form optimized for text search
UUID	UUID (128bit)	Also referred to as GUID

Table 8.2: MySQL Data Types

Data type	Type (byte)	Description
Numeric	TINYINT(1), SMALLINT(2), MEDIUMINT(3), INT(4), BIGINT(8), FLOAT(4), DOUBLE [PRECISION], REAL(8); variable for DECIMAL(), NUMERIC(), BIT(M)	Depends on the storage engine
Date and Time	Date(3), Time(3), DateTime(8), TimeStamp(4), Year(1)	
String	CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT, ENUM, SET	

8.2 More NoSQL Stores Analysis

The following are the NoSQL databases that we analyzed but didn't use for validation. Because we have subtracted information from this analysis, we are including it here.

8.2.1 GoogleApp Engine Datastore

Datastore is a dedicated back-end storage mechanism for AppEngine applications. As such it does not provide direct access for development tools or applications owned by different accounts. From a functionality perspective, this may be considered limiting; however, from a security perspective, it minimizes the possibilities to invade into a Datastore database [PC]. The Datastore comes with two flavors:

- Master/Slave Datastore which is suitable only for a limited class of applications that do not require high availability of data and can tolerate spikes in latency
- High Replication Datastore (HRD) released after provides high availability for reads and writes by storing data synchronously in multiple data centers. It provides strong consistency only for reads and ancestor queries. All the other queries are eventually consistent. The HRD implements the Paxos algorithm for an increased fault tolerance and data consistency, by having slower writes.

The Master/Slave version is not recommended to be used by Google. The following applies to the HDR.

- Data Model
  Data objects are known as entities. An entity has one or more named properties, each of which can have one or more values. The schema-less nature of the data stores consist in entities of the same kind (equivalent to the RDBMS table) need not have the same properties, and an entity's values for a given property need not all be of the same data type. (If necessary, an application can establish and enforce such restrictions in its own data model.) The key of an entity consist of the following three attributes { *The kind*:

categorizes the entity for the purpose of Datastore queries
An identifier: for the individual entity (a *key name* string created by the application or an integer *numeric ID* automatically assigned by the Datastore)
ancestor path: locates the entity within the Datastore hierarchy, but is optional.
The entities in the Datastore form a hierarchically structured space similar to the directory structure of a file system. An entity can optionally designate another entity as its parent and have many children entities. Entities of the same kind may not have the same properties. *Root entity* is an entity without a parent. *Entity group* - entities descended from a common ancestor. They are a unit of both consistency and transactionality.
The sequence of entities beginning with a root entity and proceeding from parent to child, leading to a given entity, constitute that entity's ancestor path. The complete key identifying the entity consists of a sequence of kind-identifier pairs specifying its ancestor path and terminating with those of the entity itself, e.g. Person:GreatGrandpa / Person:Grandpa / Person:Dad / Person:Me RDBMS composite keys can map to an ancestor chain.

- Data Types
  The following are the property values types supported by GAE Datastore: Integers, Floating-point numbers, Strings, Dates, Binary data.

- Indexing
  The Datastore predefines a simple index on each property of an entity, except long text strings (Text), long byte strings (Blob), and embedded entities (EmbeddedEntity). Also, if you don't want to maintain an index for a certain propery, you set it as *unindexed*. An App Engine application can define further custom indexes in an index configuration file named datastore-indexes.xml. These predefined indexes are sufficient to perform many simple queries, such as equality-only queries and simple inequality queries. For all other queries, the custom indexes defined in the configuration file datastore-indexes.xml are used. If the application tries to perform a query that cannot be executed with the available indexes (either predefined or specified in the index configuration file), the query will fail with a DatastoreNeedIndexException. The Datastore imposes limits on the number and overall size of index entries that can be associated with a single entity. Number of index entries per entity <=20,000 or more than a total of 200 markers are not allowed. For this, the service scales very well even for large amounts of data.

- Querying
  The Datastore offers low-level or standards-based APIs (java and python) that decouple your application from the underlying App Engine services, making it easier to port your application to other hosting environments and other database technologies. Queries over a single entity group, called *ancestor queries*, refer to the parent key instead of a specific entity's key. Datastore only cares about the pieces of data it needs to build indexes, the rest of your entity is seen as a sealed blob of bytes. Joins and aggregate queries aren't supported within the Datastore. Also it doesn't offer a full-text search engine yet, the feature is under development. The low-level Java Datastore API provides class *Query* for constructing queries and the *PreparedQuery* interface for retrieving entities from the Datastore.

- Transactions - Datastore supports atomic transactions. A single transaction can apply to multiple entities, as long as these entities are descended from a common ancestor. In designing your data model, you should determine which entities you need to be able to process in the same transaction. Then, when you create those entities, place them in the same entity group by declaring them with a common ancestor.
- Limitations Kind names starting with two underscores (__) are not allowed, as they are reserved. A record must be up to 1 MB in size. Operations such as joins, filters for inequality or sub-queries are not supported.

8.2.2 Azure Table Storage

Windows Azure Table Storage (WSAT) is a NoSQL database offering from Microsoft which fits more with Key-Value stores. It stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Windows Azure cloud. To allow clients to address all of their storage in the Cloud and scale to arbitrary amount of storage over time, Azure leverage DNS as part of the storage namespace and break the storage namespace into three parts: an account name, a partition name, and an object name. The service can be one of the following: blob, table or queue. The *Table* is for the NoSQL service.

- Data model A storage account is a globally unique entity within the storage system. The storage account is the parent namespace for the Table service, and is the basis for authentication. You can create any number of tables within a given storage account, as long as each table is uniquely named within the account and doesn't succeed the 100TB storage size. Each Entity defines a collection of Properties. A property is name/-value/type pairs, similar to a Column. Thus it supports flexible schema: no schema is associated with a table and within one table different entity types can be solved. There are two types of properties:
  - System properties whose names are reserved and included automatically for every entity in a table. *PartitionKey* - string value that identify the partition that an entity belongs to. If you have unique PartitionKey values for your entities, then each entity belongs in its own partition. If the values are increasing or decreasing then Azure might create range partitions. *RowKey* - string value that uniquely identify entities within each partition. *Timestamp* - DateTime value that holds the last time the entity was modified. A property that is needed for versioning and is only set by the Table service during insert/update operations. You should treat it as an opaque value as it is used only internally to provide optimistic concurrency.
  - Custom properties that can be supplied by the user. Scaling is affected by the PartitionKey values. It is a best practice to favor smaller partitions because they offer better load balancing. Larger partitions may be appropriate in some scenarios, and are not necessarily disadvantageous. For example, if your application does not require scalability, a single large partition may be appropriate.

  - Data types
    Property types: Binary (array of bytes up to 64KB),Boolean, DateTime (64-bit value), Double, GUID(A 128-bit globally unique identifier), 32-bit (int) and 64-bit (long) Integer, String (UTF-16 encoded, up to 64KB).

- Indexing Unlike a table in relational databases that allow you to manage indexes, Windows Azure tables can only have one index which is always comprised of the PartitionKey and RowKey properties. You cannot add more indexes or alter the existing ones, thus you cannot tune the performance of your table. You should put the emphasize on how you will choose the PartitionKey and the RowKey have to consider during modeling.

- Querying Querying tables and entities in the Table service requires careful construction of the request URI. Query results are sorted by PartitionKey, then by RowKey. Ordering results in any other way is not currently supported.

- RESTful API - Open Data Protocol (OData) is Microsoft's generalized XML data serialization format used to query, create, update data in the repositories it wraps [BBBI]. Any application can interact via REST APIs with the Table service as long as it can send HTTP requests to ODATA-based data services and process the OData feed that a data service returns. The commands to retrieve data use intuitive URL patterns and open HTTP verb conventions, they can return results not only in ATOM/XML format, but in JSON format too. Also through Java, Nodej.js, PHP, .NET client libraries.

- Consistency WATS is strongly consistent within one data center. It provides local replication: synchronous replication before returning success. Geo replication across data centers using asynchronous replication after returning success to user, thus being eventually consistent. If you want to have consistent writes across multiple data centers, then that it has to be done at the application level.

- Partitioning is automatically done partitioning data across servers. It provides local replication: synchronous replication before returning success. Geo replication across data centers using asynchronous replication after returning success to user, thus being eventually consistent. If you want to have consistent writes across multiple data centers, then that it has to be done at the application level.

Batch transactions or entity group transactions can be performed either via REST or by using .NET Client Library for WCF Data Services. Table service provides a limited form of ACID semantics. With a limit of 100 operations per transaction, they provide also economic benefit as it is billed as a single operation. As the Table service doesn't enforce referential integrity, you have to provide meta-data for it. The primary key for a Windows Azure table is: the *PartitionKey* and *RowKey* properties which form a single clustered index within the table.

8.2.3 Amazon DynamoDB

DynamoDB is another highly scalable and predictable in performance NoSQL offering from Amazon.

- **Data Model** Table is a collection of items. When defining tables (or updating), you also specify the capacity to be reserved in terms of reads and writes: Number of item ops per second x item size. Item is a collection of attributes and is identified by its primary key. Items within the same table can have different number of attributes (schema-free). Attribute is a name-value pair. Name is a String and the value can be: string, number, binary, string set, number set, or binary set.

- **Data types** There are two types of data types: Scalar (Number, String and Binary) and Multi-valued (String Sets, Number Set and Binary Sets). Sets are not lists, their elements are unique.

- **Keys** Hash Type Primary Key - in this case the primary key is made of one attribute only, a hash value. Hash and Range Type Primary Key - is made of two attributes. The first attribute is the hash attribute and the second one is the range attribute. Amazon DynamoDB builds an unordered hash index on the hash primary key attribute and a sorted range index on the range primary key attribute. The Hash Type property is used for partitioning, the Range Type property is for optimizing range based operations within a partition.

- **Indexing** Amazon DynamoDB stores structured data, indexed by primary key, and allows low latency read and write access to items ranging from 1 byte up to 64KB. It didn't support Secondary Indexes till April 2013. And thus, it was not a good fit a for a majority of interactive Web applications that need to present data in many different dimensions, and for each dimension an index is needed in order to perform efficient queries.

- **Querying** - mainly uses primary keys to access the data. It supports two query APIs: QUERY and SCAN.
  A *Query* operation gets the values of one or more items and their attributes by primary key (Query is only available for hash-and-range primary key tables). You must provide a specific HashKeyValue, and can narrow the scope of the query using comparison operators on the RangeKeyValue of the primary key. Use the ScanIndexForward parameter to get results in forward or reverse order by range key. Dynamo doesn't support the SQL concept of NULL.

  SCAN : The Scan operation scans the entire table and returns one or more items and its attributes. In order to yield predictable performance under all circumstances, the Query API only supports comparison operators that map to efficient index access operations. The BatchGetItem is an eventually consistent operation returns the attributes for multiple items from multiple tables using their primary keys. You can set ConsistentRead to True and have strong consistency reads at the operation level or per-table basis (when issuing Batch get operations). You can specify filters to apply to the results to refine the values returned to you, after the complete scan. JSON is used as the format for sending data and for responses, but it is not used as the native storage schema.

- **Partitioning** Amazon DynamoDB automatically partitions data over a number of servers to meet your request capacity. In addition, DynamoDB automatically replicates your data synchronously across multiple Availability Zones within an AWS Region to ensure high-availability and data durability. It provides automatic replication and fail-over.

- **Limitations** Size item(row) up to 64KB. This limits the use cases of DynamoDB. For example a page crawler may store the entire page content in one field. Also, if you want to store a document as a field, e.g. an image, which much likely exceeds the 64KB. You simply cannot use DynamoDB for such use cases. One workaround to store binary data is to encode them using base64 enconding, but it increases the size. Query or Scan operations: The maximum number of item attributes that can be retrieved for a single operation is 100. Also, the number of items retrieved is constrained by a 1 MB the size limit.

Bibliography

[10ga] 10gen. MongoDB Operations Best Practices. http://info.10gen.com/rs/10gen/images/10gen-MongoDB_Operations_Best_Practices.pdf.

[10gb] 10gen, Inc. Foursquare. http://www.10gen.com/customers/foursquare.

[ACKT] B. Alexe, B. T. Cate, P. G. Kolaitis, and W.-C. Tan. Characterizing Schema Mappings via Data Examples.

[Ale] AWS Case Study: Alexa. http://aws.amazon.com/solutions/case-studies/alexa/.

[Ama] Amazon. Amazon DynamoDB Announces Support for Local Secondary Indexes. http://aws.amazon.com/about-aws/whats-new/2013/04/18/amazon-dynamodb-announces-local-secondary-indexes/.

[Ana] S. S. Anand. Netflixs Transition to High-Availability Storage System.

[Ant] Anti-patterns in Cassandra. http://www.datastax.com/docs/1.2/cluster_architecture/anti_patterns#super-columns.

[AWS] Amazon Web Services Forum. https://forums.aws.amazon.com/thread.jspa?threadID=85511.

[Bac] T. Bachmann. Entwicklung einer Methodik für die Migration der Datenbankschicht in die Cloud, Diploma Thesis no. 3360, Institute of Architecture of Application Systems, University of Stuttgart, 2012. http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3360&mod=0&engl=0&inst=IAAS.

[Ban11] K. Banker. *MongoDB in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[BANE09] Bernstein, P. A, Newcomer, and Eric. *Principles of Transaction Processing*. Morgan Kaufmann, 2009.

[Basa] Basho. From Relational to Riak. http://basho.com/assets/RelationaltoRiakDEC.pdf.

[Basb] Basho. ideeli Uses Riak For Peak-Load Service Continuity. http://basho.com/assets/Basho-Case-Study-ideeli.pdf.

[Basc] Basho. Riak for Retail and eCommerce Platforms. http://basho.com/riak-for-retail-and-ecommerce-platforms/.

[Basd] Basho.com. Relational to Riak. http://basho.com/assets/RelationaltoRiakDEC.pdf.

[BBBI] A. J. Brust and I. Blue Badge Insights. NoSQL and the Windows Azure Platform.

[BMPV] A. Bonifati, G. Mecca, P. Papotti, and Y. Velegrakis. Discovery and Correctness of Schema Mapping Transformations. http://disi.unitn.it/~velgias/docs/BonifatiMPV11.pdf.

[Bre] E. A. Brewer. Towards Robust Distributed Systems. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[Bre12] E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[Cat] R. Cattell. Scalable SQL and NoSQL Data Stores.

[CDG+08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[CM] M. Cooper and P. Mell. Tackling Big Data. http://csrc.nist.gov/groups/SMA/forum/documents/june2012presentations/fcsm_june2012_cooper_mell.pdf.

[Cod90] E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Cou] Couchbase. Couchbase Survey Shows Accelerated Adoption of NoSQL in 2012. http://www.couchbase.com/press-releases/couchbase-survey-shows-accelerated-adoption-nosql-2012.

[CRS+] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. http://www.mpi-sws.org/~druschel/courses/ds/papers/cooper-pnuts.pdf.

[CST+10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems With YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[Data] Datastax. Launching the DataStax Community AM. http://www.datastax.com/documentation/cassandra/1.2/cassandra/install/installAMILaunch.html.

[Datb] DataStax. Understanding the Cassandra data model. http://www.datastax.com/docs/1.2/ddl/index/.

[DG08] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DHJ+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[Far] E. Farrell. SQL to NoSQL, Lessons learnt migrating a large and highly-relational database into classic NoSQL. `http://de.slideshare.net/endafarrell/lessons-learnt-coverting-from-sql-to-nosql`.

[FGC+97] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. *SIGOPS Oper. Syst. Rev.*, 31(5):78–91, October 1997.

[FHH+] R. Fagin, L. M. Haas, M. Hernandez, R. J. Miller, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exchange. `http://disi.unitn.it/~p2p/RelatedWork/Matching/FaginHHMPV09.pdf`.

[GH] M. Grinev and M. Hentschel. Extending Cassandra with Asynchronous Triggers. `http://maxgrinev.com/2010/07/23/extending-cassandra-with-asynchronous-triggers/`.

[Gra] A. Gray. Amazon Web Services Blog. `http://aws.typepad.com/aws/2012/01/aws-howto-using-amazon-elastic-mapreduce-with-dynamodb.html`.

[Gro] E. S. Group. Big Data-as-a-Service. `http://www.emc.com/collateral/software/white-papers/h10839-big-data-as-a-service-perspt.pdf`.

[Hew11] E. Hewitt. *Cassandra : The Definitive Guide*. O'Reilly, 2011.

[HSB] NoSQL Data Modeling Techniques. `http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/`.

[IBM] IBM. Big data: Why it Matters to The Midmarket. `http://http://www.ibm.com/midmarket/us/en/article_BusinessAnalytics4_1212.html`.

[Ins] M. G. Institute. Big data: The next frontier for innovation, competetion, and productivity. `http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation`.

[Lea] LearnAWS.com. Migrating from MySQL to Amazon SimpleDB.

[Leo] J. Leonard. Nokia Entertainment: Why we went Mongo. `http://www.computing.co.uk/ctg/analysis/2260724/nokia-entertainment-why-we-went-mongo#ixzz2QB8Au1iX`.

[Lob] L. Lobel. Programming Beyond Relational Features in SQL Server 2008. `http://oreillynet.com/pub/e/1699`.

[Mar] A. Marcus. The NoSQL Ecosystem. `http://www.aosabook.org/en/nosql.html`.

[MB11] E. Meijer and G. Bierman. A co-Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 54(4):49–58, April 2011.

[Mem] Memcached. `http://memcached.org/`.

[MG09] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical report, July 2009.

[Mica] Microsoft. Microsoft Official Training Materials for Microsoft Dynamics. `http://ftp.tiaonline.org/gomembers/8400B_FRx67_ENUS_RDEII/8400B_FRx67_ENUS_RDEII/FRx67_ENUS_RDEII_C.pdf`.

[Micb] Microsoft. Random and Sequential Data Access. `http://technet.microsoft.com/en-us/library/cc938619.aspx`.

[Moh13] C. Mohan. History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 11–16, New York, NY, USA, 2013. ACM.

[Mona] MongoDB. Data Modeling Considerations for MongoDB Applications. `http://docs.mongodb.org/manual/core/data-modeling/`.

[Monb] MongoDB. Sharded Cluster Overview. `http://docs.mongodb.org/manual/core/sharded-clusters//`.

[Myt] D. Mytton. MongoDB Benchmarks. `http://blog.serverdensity.com/mongodb-benchmarks/`.

[NE] D. Nelubin and B. Engber. Ultra-High Performance NoSQL Benchmarking. `http://odbms.org/download/NoSQLBenchmarking.pdf`.

[Neoa] Facebook's Social Graph, Neo4j show rising use of graph databases. `http://www.neotechnology.com/2013/01/facebooks-social-graph-neo4j-show-rising-use-of-graph-databases/`.

[Neob] Neo Technology execs: How Neo4j beat Oracle Database. `http://www.neotechnology.com/2013/02/neo-technology-execs-how-neo4j-beat-oracle-database/`.

[NoS] List of NoSQL Databases. `http://nosql-database.org/`.

[OGOG$^+$11] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov. Security Issues in NoSQL Databases. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 541–547, 2011.

[Oraa] Oracle. NoSQL Database for Mobile Social Gaming. `http://www.oracle.com/technetwork/products/nosqldb/overview/nosql-database-for-mobile-gaming-1591208.pdf`.

[Orab] Oracle Press Release. Oracle Oracle Announces General Availability of MySQL Cluster 7.2. `http://www.oracle.com/us/corporate/press/1521659`.

[Pat] J. Patel. Cassandra Data Modeling Best Practices, Part 1. `http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/`.

[PC] L. Pizette and T. Cabot. Database as a Service: A Marketplace Assessment. `http://www.mitre.org/work/tech_papers/2012/11_4727/cloud_database_service_dbaas.pdf`.

[PCZ12] A. Pavlo, C. Curino, and S. Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[Posa] PostgreSQL Online Documentation. `http://www.postgresql.org/docs/9.2/static/index.html`.

[Posb] PostgreSQL. PostgresQL hstore. `http://www.postgresql.org/docs/9.1/static/hstore.html`.

[Pra] Prasanna Venkatesh and Nirmala S. NewSQL — The New Way to Handle Big Data. `http://www.linuxforu.com/2012/01/newsql-handle-big-data/`.

[Pri08] D. Pritchett. BASE: An Acid Alternative. *Queue*, 6(3):48–55, May 2008.

[Rav] RavenDB. Transaction support in RavenDB. `http://ravendb.net/docs/client-api/advanced/transaction-support`.

[Red] Redis Database. `http://redis.io/`.

[Ria] Riak Database. `http://basho.com/`.

[RWC12] E. Redmond, J. Wilson, and J. Carter. *Seven Databases in Seven Weeks. A Guide to Modern Databases and the NoSQL Movement*. Oreilly and Associate Series. Pragmatic Bookshelf, 2012.

[SA12] A. Schram and K. M. Anderson. MySQL to NoSQL: Data Modeling Challenges in Supporting Scalability. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, SPLASH '12, pages 191–202, New York, NY, USA, 2012. ACM.

[SABL13] S. Strauch, V. Andrikopoulos, T. Bachmann, and F. Leymann. Migrating Application Data to the Cloud Using Cloud Data Patterns. In *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*, pages 36–46. SciTePress, 2013.

[SF12] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[Sho] N. Shoji. Key-Value Store "MD-HBase" Enables Multi-Dimensional Range Queries.

[Sim] A. SimpleDB. Developer Guide (API Version 2009-04-15). `http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/DataModel.html`.

[SSM$^+$] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-Trees. `https://www.usenix.org/conference/fast13/unifying-file-systems-and-databases-efficiently`.

[Sta] Stamford, Conn. Gartner Says Big Data Will Drive $28 Billion of IT Spending in 2012. `http://www.gartner.com/newsroom/id/2200815`.

[Tiw11] S. Tiwari. *Professional NoSQL*. Wrox programmer to programmer. John Wiley, Hoboken, N.J. Wiley Chichester, 2011. Index.

[TN] T.Laszewski and P. Nauduri. Discovery and Correctness of Schema Mapping Transformations. Elsevier Science, 2011.

[Tra] Transaction Processing Performance Council. TPC-H: TPC Benchmark^{TM}H. `http://www.tpc.org/tpch/`.

[VBB11] W. Voorsluys, J. Broberg, and R. Buyya. *Introduction to Cloud Computing*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2011.

[Voga] W. Vogel. Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications. `http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html`.

[Vogb] W. Vogel. Eventually Consistent - Revisited. `http://www.allthingsdistributed.com/2008/12/eventually_consistent.html`.

[WDDB12] B. Wylie, D. Dunlavy, W. Davis, and J. Baumes. Using NoSQL databases for streaming network analysis. In *Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on*, pages 121–124, 2012.

[Wor] Wordnik, from MySQL to MongoDB. `http://www.slideshare.net/mongosf/from-mysql-to-mongodb-at-wordnik-tony-tam`.

[Wri] MongoDB, Write Concern. `http://docs.mongodb.org/manual/core/write-operations/#write-concern`.

[Zyn] Zynga. Building a Scalable Game Server. `http://code.zynga.com/2011/07/building-a-scalable-game-server/`.

All links were last followed on October 31, 2013.

Acknowledgement

I am heartily thankful to my supervisor Steve Strauch from the University of Stuttgart for his encouragement, guidance and support in all the phases of this master thesis.

Rilinda Lamllari

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Stuttgart, June 27, 2013 ______________________

(Rilinda Lamllari)