FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Towards Providing Unified Access To Cloud Data Storage Services

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Rene Nowak
Matrikelnummer 0725177

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Mag. Dr. Philipp Leitner

Wien, 19.07.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

Towards Providing Unified Access To Cloud Data Storage Services

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Rene Nowak
Registration Number 0725177

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Mag. Dr. Schahram Dustdar
Assistance: Mag. Dr. Philipp Leitner

Vienna, 19.07.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Rene Nowak
Leystraße 166\19

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

With an increasing number of devices using applications, the amount of data that has to be managed has grown rapidly. This means, that scalability of databases gets more and more important. So in the last years so called not only SQL (NoSQL) databases have become popular because they can be distributed more easy than traditional relational databases.The problem is, that different data stores use different ways to manage their data and provide specific interfaces to persist and load this data sets. This makes it hard for application developers to use this new technologies in their applications. So using different types of persistence, as described in the polyglot persistence model, means that a lot of code has to be written for covering the different APIs.

The following thesis describes and presents a solution to access different types of data storage through one interface. Therefore, the differences between the management and storage of data are described and discussed. Based on this discussion, a common interface will be designed. Because the API can not cover all different functionalities, developers will also get the possibility to access specific libraries, to use special functions for a data store. This means, that data can be stored where it fits best, without losing any functionality or flexibility. In some cases, it is useful to move data from one data store to another. This often requires a lot of migration code to read and write data. In this thesis, we will also present a solution for this problem and a prototype implementation of this feature will be discussed.

Kurzfassung

Die Menge an Daten die von heutigen Applikationen verwaltet werden muss, hat durch die steigende Zahl an Geräten, die diese Anwendungen benutzen, zugenommen. Das bedeutet natürlich, dass die Skalierbarkeit der zugrunde liegenden Datenbanken immer wichtiger wird. In den letzten Jahren sind daher neben den weit verbreiteten relationalen Datenbanken sogenannte NoSQL Datenbanken populär geworden, da diese wesentlich leichter auf verschiedene Server verteilt werden können. Das Problem dabei ist, dass die verschiedenen Datenbanken unterschiedliche Arten benutzen um die Daten zu verwalten und auch verschiedene Schnittstellen bereitstellen um diese verfügbar zu machen. Dieser Umstand macht es Entwicklern schwer diese neue Technologie in ihren Anwendungen zu verwenden. Um also die verschiedenen Arten der Speicherung wie im Model der Polyglot Persistence beschrieben umsetzen zu können, ist eine Menge Code erforderlich.

Die voliegende Arbeit beschreibt und präsentiert eine Lösung um die verschiedenen Arten von Datenbanken, über eine einheitliche Schnittstelle, verwenden zu können. Dafür werden zuerst die Unterschiede in der Verwaltung und Speicherung beschrieben und diskutiert. Auf Basis dieser Diskussion wird anschließend die gemeinsame Schnittstelle definiert. Da eine gemeinsame Schnitstelle natürlich nicht alle Funktionen der einzelnen Datenbanken abedecken kann, wird eine Möglichkeit geschaffen um es den Entwicklern zu ermöglichen, auf spezifischere Schnittstellen zu zu greifen. Dadurch wird es möglich Daten dort zu speichern, wo diese am besten passen ohne dabei irgendeine Funktionalität oder Flexibilität ein zu büßen. In einigen Fällen kann es auch notwendig sein, dass Daten von einer Datenbank direkt in eine andere verschoben werden müssen. Dazu ist es oft notwendig, eine Menge an Code zu schreiben der die Daten zuerst liest und anschließend wieder schreibt. In der vorliegenden Arbeit wird auch für dieses Problem eine Lösung beschrieben und ein Prototyp implementiert.

Contents

CHAPTER 1

Introduction

In the last years, software engineering research and practice have put more and more focus on Cloud Computing [13, 82, 87], which offers different types of computing resources over a network. These services are normally distributed across a large number of servers and network resources to provide a reliable infrastructure for the users of the services. Modern Cloud Frameworks are expected to be scalable, fast and should be available all the time.

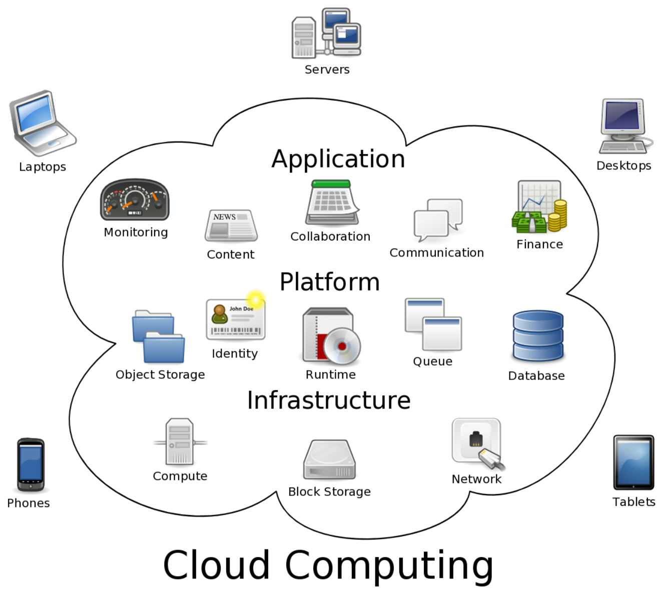


Figure 1.1: Cloud Computing [56]

Cloud computing means, that user run their services in distant server farms and use them over the internet with different devices like laptops, tablets, phones or local servers. The infrastructure and framework where these services run is often hidden from the users and seems to be like a black box. For communication, standardized interfaces and protocols are used. From the customer view, cloud computing is the shift from local resources and infrastructure to remote services. This is especially interesting for small companies and start-ups which often do not have the money to build up their own data centers. In the cloud different forms of services can be provided like applications, middleware or infrastructure to run custom applications. Large companies often operate their own cloud for internal scalable services which are not expected for public usage. A more detailed introduction in cloud computing will be given in Chapter 2.

In nearly each modern application, it is also necessary to manage data. Nowadays the amount of data which has to be handled by applications has increased and so the requirements to the storage systems, especially the databases, have changed. Database systems should now be able to be reliable and scalable to manage the increasing amount of data produced by the applications and their users. This means, that it should be possible to simply extend an existing database and scale with the system and the number of users. Common Relational Database Management Systems (RDBMS) use relations, often called tables, to manage data [4, 65].

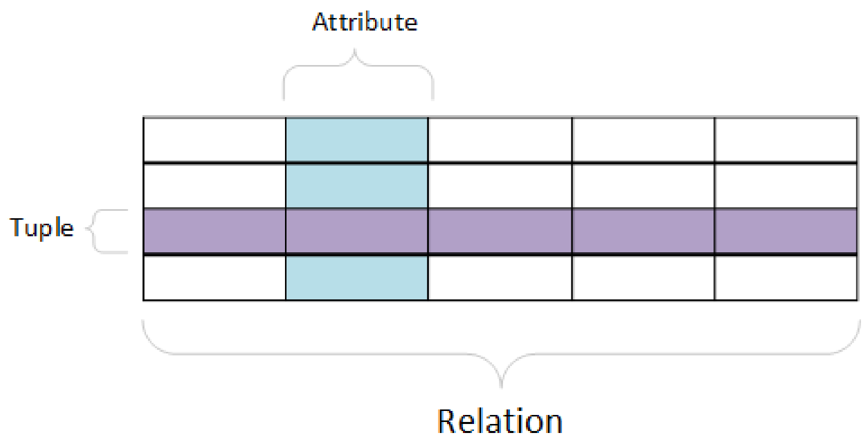


Figure 1.2: Relational database concept

This concept is shown in Figure 1.2. Relations consist of tuples (rows) and attributes (columns). Most databases nowadays use this concept of relations to manage their data. The problem is, that since the invention of RDBMS, the amount of data which has to be handled by a database has increased and is still growing. This also means, that data management can not be done by a single server anymore and has to be distributed across various servers instead. There are different approaches to handle distribution of relations like horizontal and vertical fragmentation [4]. But there is always a trade-off between scalability and performance, which are both important factors in modern applications. So with Cloud Computing getting more and more important, new ways of managing data have to be found to fit the requirements of these systems. One of this new sort of databases are so called NoSQL databases [21, 73, 77]. This new kind of databases covers both traditional relational databases which use relations to persist data and non relational databases which do not use relations to manage data. Actually NoSQL databases can be grouped into three categories [21, 37, 77]

- **Key-Value**
  Key-Value datastores manage data in tuples. This means, that data gets stored as key-value pair where the key is the unique identifier (equivalent to the primary key in relational databases) of the record. Most of the modern programming languages provide an equivalent mechanism to store data. The advantage of this approach is, scalability and performance. But when complex data with relations between the different records should be stored, key-value data stores are not very helpful. It is also not possible to perform complex queries and aggregations on the data stored in this type of database.

- **Column Oriented**
  Column Oriented databases manage data in columns. This means, that all records of a given column are stored together. In common relational databases, data is managed in rows. One of the advantages of this approach is, that different sets of data can have different numbers of columns. In contrast to row oriented data management, there is no need to store null values for not used columns in a data set. Columns can be scaled across different servers easily supporting a large amount of data.

- **Document Oriented**
  In document oriented databases data is managed in so called documents. A document has an unique identifier (id) and can have various fields to store data. One of the advantages of the document approach is, that data structures can be nested. These kind of data stores can manage all data which can be expressed as a document.

One of the main differences between relational and all types of non relational databases is, that data in non relational databases does not require to be structured [61]. This means, that all kind of data can be stored in the same location, unlike relational databases where different types of data have to be stored in different relations. All of the mentioned data store types provide some basic mechanisms to query data but with a less powerful search engine. In most cases simple Java Scripts can be used to handle data in the data stores. To run this kind of queries on all database nodes, Map Reduce is used to receive and aggregate the result sets [18, 21]. The main problem with NoSQL databases is, that they only gurantee eventual consistency what can be problematic in some use cases.

The CAP theorem illustrated in Figure 1.3 states, that it is impossible for a distributed system to provide all of the three guarantees consistency, availability and partition tolerance [21, 73]. So both, relational and non relational databases, can only choose two of the three properties mentioned above. Today wide spread databases normally support consistency and availability. Especially in use cases where critical data (e.g. banking data) is handled, consistency is an important property and still comes before scalability. In terms of scalability, partition tolerance is required. This means, that even if a node fails the whole system is able to fulfill the request. Most of the NoSQL databases choose availability and partition tolerance, but can be configured

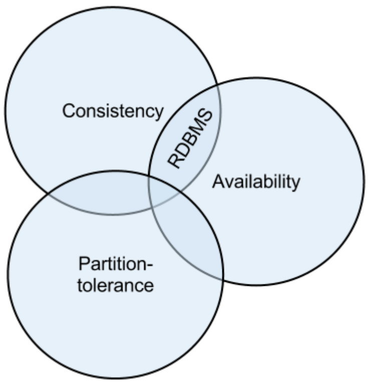


Figure 1.3: CAP Theorem

to be for example consistent and available [37]. The important thing is, that there can only be two out of three properties fulfilled. In modern cloud applications, the availability and scalability of the used data storage is an important factor for the success of a system. The requirement for consistency can often be relaxed to build systems for a large number of users producing a huge amount of data. So often the data in this data stores should be **B**asically **A**vailable, **S**oft state, **E**ventual consistency (BASE) [18, 21]. This is in contrast to RDBMS where data has **A**tomicity, **C**onsistency, **I**solation, **D**urability (ACID) [18, 21].

1.1 Motivation

Nowadays applications are expected to handle different types of data and should be able to fulfill requests with a high performance. This means, that today an application has to manage structured and unstructured data at the same time. Clearly it would be possible to store all data in a RDBMS as well, but this will sooner or later lead to bad performance and storage usage. So the question software developers should ask them self today is, why just use one type of database and not different types ? Each sort of database has its strength and together a system of data stores can create an ecosystem that is more powerful and even more flexible than a single standalone storage. This consideration has lead to a persistence model called polyglot persistence [37, 39, 74, 84]. The idea behind polyglot persistence is the same as polyglot programming. In polyglot programming, a team uses different programming languages for different problems. So programmers are able to use features from each language to solve a problem faster and more elegant. The same concept can be applied to storage systems. Use a set of databases to store different types of data. So each set of data can be stored where it fits best. This is can also help to increase development speed because data has not to be prepared for persistence in a relational database.

In Figure 1.4 there is an example of a retailers web store realized with polyglot persistence [39]. As we can see, there are a lot of different types of data which have to persisted an managed

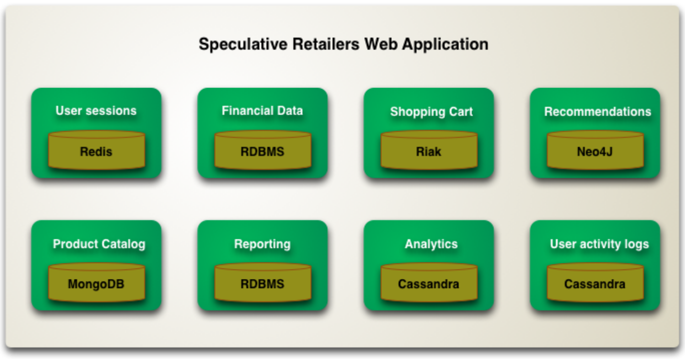


Figure 1.4: Example Polyglot Persistence [39]

by the web application. Most of the databases in this example are non relational but there are still RDBMS present. Financial or reporting data is usually structured and can therefore be stored in tables. Another problem is, that especially in the case of financial data, transactions and ACID properties are required. This sort of data has to be consistent all the time and should be handled completely or not at all. On the other side, products in a product catalog normally have different properties and different descriptions. This unstructured data can be stored in a document based data store because a catalog can be seen as a collection of documents. Key-Value stores, for example riak, can be used to persist shopping card information. Located in this kind of data store, information for a specific shopping card can be found very fast. Some NoSQL databases already support polyglot programming, but actually an universal tool is missing to help developers using different data stores at the same time in there applications [37].

Another problem of using different storage types is, that each system has different ways to manage data and therefore requires different handling of data access.

Datastore	**Unit**	**Interface**
RDBMS	Tables	JDBC
Riak, Amazon S3	Buckets	REST
CouchDB	Databases	REST
Redis	Databases	Client, JDBC, ...

Table 1.1: Data store Data Units and Interfaces

In Table 1.1 there are four examples of databases and how they manage data and connections for applications [8, 18, 73]. Classic relational databases manage data in tables and use for example JDBC for connections. Today in most applications an extra layer is used to map objects to tables and to handle database access. Riak and Amazon S3, which are both key-value data stores, use buckets to group data and provide a rest service for managing data and buckets. With the use of rest services, operating on data is very easy and does not require special tools. Another advantage is, that data can for example be shown in a browser an so developers do not require extra programs to look what there software has persisted. CouchDB and Redis simply use databases and do not group data within databases anymore. Redis supports a wide range of possibilities to establish a database connection (e.g. JDBC, standalone Client). Different ways of data management and access make it hard for application developers to use polyglot persistence in their products.

1.2 Contribution

This thesis deals with the problem of finding a simple way to access different data stores through one interface. To provide a solution, a common interface for accessing the data stores will be created and a sample implementation for some data stores will be provided. The following research questions will be addressed in this work:

- Is it possible to create a simple interface for accessing different data store types ? Is it also possible to get access to specific libraries to use special functionality provided by the database ?
- How can different databases be used together to support polyglot persistence ?
- How can objects be moved from one data store to another without writing a lot of migration code ?

1.3 Organization

Chapter 2 provides an overview of Cloud Computing and the usage of data stores. Modern technologies for persistence like ORM will be presented. At the end rest services and the use of them will be explained. In Chapter 3 recent research work in the field of data stores and Cloud Computing will be presented. The design of a common interface for the different databases will be described in Chapter 4. The basic implementation will be presented in Chapter 5 and a short example with an evaluation is discussed in Chapter 6. The last chapter will describe some future work and plans for improving the handling of the different data stores.

CHAPTER 2

State of the Art

2.1 Cloud Computing

In the last years, cloud computing has become very popular for providing different types of services over the internet. Especially for small companies and start-ups, this type of services can be very helpful to build up an IT infrastructure without investing a lot of money [81]. The advantage using cloud computing in this case is, that customers only pay for computing and storage power they really need because resources follow the load. This pay per use model is one of the reasons why cloud services become more and more popular [81]. Another use-case is the field of research and data analytics, where often a lot of computing and storage power is needed to run simulations or calculate models. In this case, cloud computing can help researchers to save money because they do not need to buy there own hardware and only have to pay for resources they really need [9]. They also have infinite resources available for there calculations and these resources are available on demand. By providing IT infrastructure as services over the internet, outsourcing to countries with lower prices for electricity and other resources necessary to run data centers, providers and customers can save a lot of money [13].

The definition of cloud computing is not very clear at all because often different services and technologies are covered by the various existing definitions. In [81] for example, about 20 different definitions where analyzed and compared to get a standard definition. Cloud computing does not refer just to the software and services delivered to the customers, it also covers the hardware used in the data centers to provide those services [13]. The minimum features all cloud services have together, are scalability, pay-per-use and virtualization [81]. Scalability in the case of cloud services means, that a large pool of resources is provided by infrastructure providers. These resources can easily be expanded to meet the requirements and to handle increasing service demands [87]. Pay-per-use means, that users only have to pay for resources they really use. This is especially attractive, when resourcing power is needed just for short periods of time. In cloud computing, virtualization is used to virtualize both, software and hardware and to provide different monitoring technologies [81]. The idea behind cloud computing as it is used today is not new. In 1960 John McCarthy already described that facilities will be provided like a utility. When Google CEO Eric Schmidt was using the word cloud computing to describe a new business model for providing services over the internet, it became very popular [87].

Sharing resources to provide high computing power to users is already used in the field of Grid Computing. This the reason why Cloud Computing is often confused with Grid Computing [81]. Table 2.1 compares different features of grids and clouds:

Feature	**Grid**	**Cloud**
Resource Sharing	Collaboration (VOs, fair share).	Assigned resources are not shared.
Resource Heterogeneity	Aggregation of heterogeneous resources.	Aggregation of heterogeneous resources.
Virtualization	Virtualization of data and computing resources.	Virtualization of hardware and software platforms.
Security	Security through credential delegations.	Security through isolation.
High Level Services	Plenty of high level services.	No high level services defined yet.
Architecture	Service orientated.	User chosen architecture.
Software Dependencies	Application domain dependent software.	Application domain independent software.
Platform Awareness	The client software must be Grid enabled.	The SP software works on a customized environment.
Software Workflow	Applications require a predefined workflow of services.	Workflow is not essential for most applications.
Scalability	Nodes and sites scalability.	Nodes, sites, and hardware scalability.
Self-management	Reconfigurability.	Reconfigurability, self-healing.
Centralization Degree	Decentralized control.	Centralized control (until now).
Usability	Hard to manage.	User friendliness.
Standardization	Standardization and interoperability.	Lack of standards for Clouds interoperability.
User Access	Access transparency for the end user.	Access transparency for the end user.
Payment Model	Rigid.	Flexible.
QoS Guarantees	Limited support, often best-effort only.	Limited support, focused on availability and up-time.

Table 2.1: Grid vs Cloud Characteristics [81]

Nowadays, grid computing is essentially used in research to run large data analysis and computing tasks on a shared hardware resource.

Cloud computing used today can be grouped into three different types, based on the level of access granted to the computing resources and the services provided.

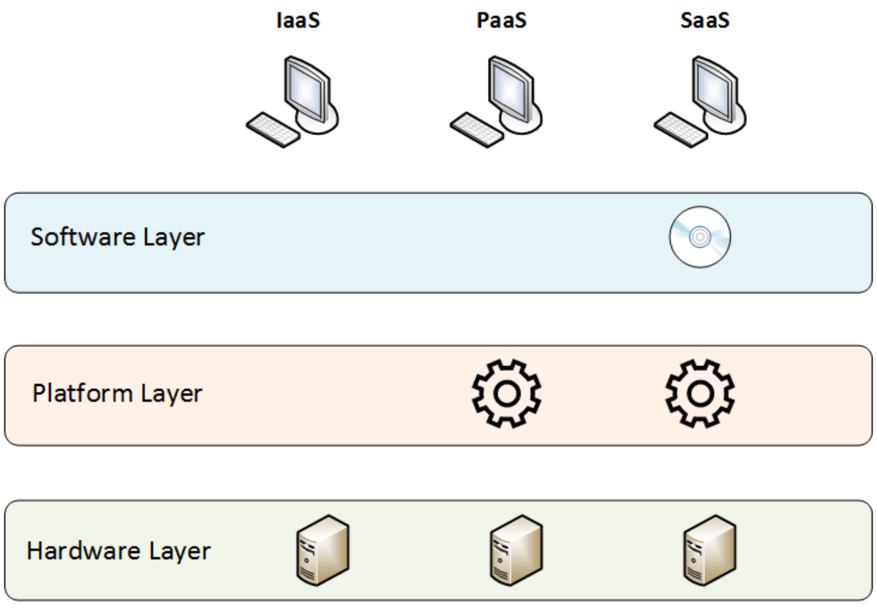


Figure 2.1: Cloud Architecture

Figure 2.1 shows the three groups and the layers included which are in practice Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

- IaaS provides on demand infrastructure services. These resources are normally provided in the form of virtual machines [87]. Users get full access to the virtualized system and can install the software and services they need. This also means, that users of this type of services are responsible for the security of there systems.
- PaaS provides a middleware or development environment, which can be accessed and utilized online to create services [62]. The difference to IaaS is, that users do not get direct access to the virtualized hardware and have to use the provided environment instead.
- SaaS provides on-demand applications over the internet [87]. The advantage of this service is, that software can be used on every device with internet access and no additional software has to be installed on the clients.

Together this three services build up the business model of cloud computing. Different providers offer there services according to these categories. Figure 2.2 shows the categories based on the access type.

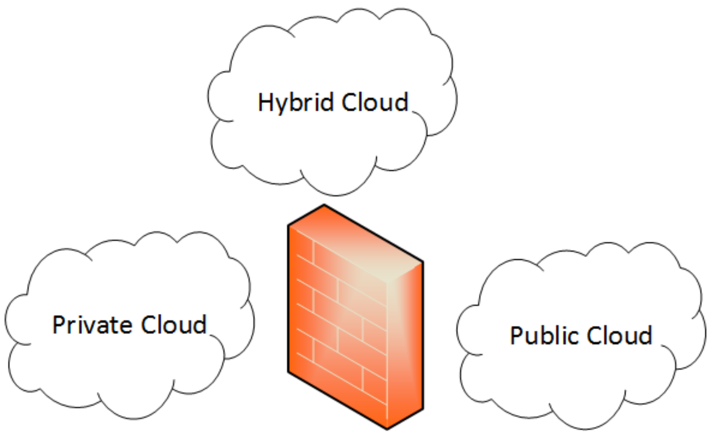


Figure 2.2: Cloud Types

- Private Clouds are designed for exclusive usage inside a single organization. These clouds are built and controlled by the organization and offer the highest degree of control over performance, reliability and security. But often this type of clouds is criticized because they are similar to traditional server farms.

- Public Clouds are clouds which offer their resources as services to the general public. This type of clouds often lacks fine-grained control over data, network and security settings.

- Hybrid Clouds are combinations of private and public clouds. Some parts of the service infrastructure runs in a private cloud and is limited to the organization and the remaining part is available for general public. These type of clouds tries to address the limitations of each approach. Designing a hybrid cloud requires carefully determining the best split between the private and the public parts of the cloud.

The hype of cloud computing is sometimes compared to the transformation from mainframes to personal computers. In 1980 when personal computers arrived, there appeal was to liberate programs and data from central computing centers [48]. Nowadays, there is a transformation from personal computers back to centralized computing and data centers and cloud computing is a part of this movement. But as every other technology, cloud services have some pros and cons that have to be considered when planning to use cloud services. Some of the pros are:

- **Scalability**
  This is one of the main advantages because resources do not have to be calculated on maximum load anymore and instead scale with the requirements.

- **Availability**
  Services provided by large companies are known for there high availability and up-time.

- **Pricing**
  Users have to pay only computing and storage resources they need and do not have to pay for services they not require.

But as mentioned before, there are also some cons which users should be aware of. Some of the cons often mentioned in connection with cloud computing are [13]:

- **Service Failures**
  If users rely on cloud services, failures in the provided services can cause required software or data to be not available. Often customers expect the same quality as known from large companies (e.g. Google) for new services too. Providers for that use geographically distributed data centers and redundant hardware to guarantee always available services. But despite these efforts it can happen, that failures occur and computing and data resources are not available. One solution for this problem is, to use different providers and to not rely on a single company.

- **Proprietary APIs**
  Most cloud providers today use there own set of APIs. This means, that for application developers changing the provider or using different providers is a lot of work. Especially the access of data is often very different and requires special interfaces. A solution for this problem would be the standardization of the different APIs. Re-implementations of proprietary interfaces can also help to build up services that do not rely on one provider.

- **Performance Unpredictability**
  Experiences have shown, that sharing of CPUs and memory among virtual machines works well but that network and I/O sharing is often problematic. This is especially problematic if a lot of data has to be transferred. So the best solution for this problem is, to keep data at a single point whenever it is possible. Improvements in virtualization and operating systems can also help to optimize sharing of I/O resources between different virtual machines.

There are many open questions in cloud computing and a lot of research has to be done so that actually existing problems can be solved in the future. Providing services over the internet is very complex because many factors have to be considered.

Today there exist many different cloud service providers. One of the companies offering this types of services is Amazon with its products EC2 and S3 [40]. EC2 is Amazon's IaaS platform and offers many different types of virtual machines. All operations in this clouds can be triggered over well documented web services and there also exist libraries for different programming languages [7]. S3 is a high scalable key-value cloud storage and can be combined with EC2. It also offers web services for data manipulation and configuration. Another big company providing services over the internet is Google. With Google's app engine, developers can run software which uses the Google middleware on there cloud [14]. App engine is a PaaS that offers libraries and utilities to run scalable applications inside the Google cloud. Currently there is only support for three different programming languages [43]. For data management, big table, a distributed storage system for structured data, can be used [22]. Goggle also provides very popular SaaS platforms. With Google Docs for example, documents, spreadsheets and presentations can be created and managed over the internet [44].

2.2 NoSQL

With cloud computing getting more and more popular, there are also new ways of persisting and managing data. The problem of common ways of data management is, that this solutions are not very scalable and so they are not very useful in cloud environments because one of the main goals of this systems is scalability. So in the last years, a lot of research has be done, to find new ways of storing data and providing access to this new type of storage systems [22, 34, 73]. Especially the way how data is managed inside the database has changed and new data models were developed, to meet the requirements of modern applications. Nowadays, all of the wide spread cloud systems use and support this new type of data persistence and provide easy to use APIs, to manage data in cloud applications [8, 43].

The term used for this new databases is NoSQL and is used to describe all different variants of modern data stores. This means, that also common relational databases are covered by this term. NoSQL databases usually provide different ways of accessing data than Structured Query Language (SQL) databases but many of them also provide SQL like query languages [37, 77]. The main goal of this storage systems is, to provide a scalable and always available data storage, to manage the increasing amount of data in modern applications, especially web applications [73]. In contrast to traditional relational databases, this new type of data stores often provides only eventual consistency. This means, that data is only eventually the same on all nodes of the storage systems and users of applications can maybe see updated information later [73]. But for most of the applications today, scalability and handling of many users at the same time, is more important than consistency. Of course there are still systems like banking applications, where consistency is a very important property, but this type of programs will not use NoSQL databases to manage their data.

Modern data stores use different ways and models to store and manage data. There does not exist one perfect solution to provide a scalable and available database. Instead there are many different approaches, to provide this properties and each of this solutions has a special use case. Basically all of the existing NoSQL databases can be grouped into different categories based on there data model [37].

Key-Value [37]

Key-Value data stores provide a very light weight data model to manage data. Basically data is managed in tuples, consisting of a key and a value field. Many modern programming languages also provide such data structures, to manage application data. The key field is used to identify an entry in the database and can be compared to the primary key in relational databases. In the value field, different types of data can be stored no matter which format or encoding the data has, in contrast to relational databases, where data fields in the same column are required to be of the same type. The motivation for this data model is, that in many cases only simple sets of data need to be persisted. This means, that the data persisted does not have complex relations, which have to be managed by the database. Individual information, like for example the credit of a shopping card, can be persisted in key value stores. The main advantage of this approach is, that data can be persisted and queried with high performance. Another benefit of this kind of data management is, that different data sets can be distributed across different servers easily. This is possible because there do not exist any database side relations between the tuples and so scaling up the system is possible without much effort.

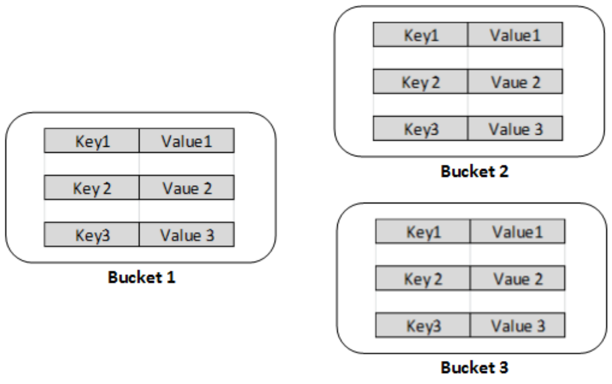


Figure 2.3: Key-Value data model

As shown in Figure 2.3, key value data stores manage data in so called buckets. Buckets are logical units for grouping key value pairs inside the data stores. This can be compared to tables or schemas in relational data bases. So a data set in the database is uniquely identified by the bucket containing the tuple and the key assigned to it.

Nowadays, many different implementations of key value data stores exist and nearly all of them provide different features to manage data in a more elegant way. Some of this products also provide simple ways of defining relations between data tuples. One of the most popular key value stores is Amazon Dynamo, which is used in many of Amazons products. Dynamo is a fast and scalable key-value data store, that is used in many different applications for example S3 and which was developed by Amazon. Dynamo can be directly used in the form of DynamoDB [6]. In this section, we will give a short introduction on how Dynamo works and the design of the system. All information explained here can be found in [34].

Dynamo was designed to run on thousands of servers distributed in various data centers all over the world. For this purpose, it was necessary to design a new database system to handle the problems occurring with such a large distributed system. The goal of the design was to get an always writeable data store. Therefore the architects of Dynamo chose a key-value based data store to fulfil this requirement. One of the main reasons for this decision is, that most relational databases lack scalability because they spend a lot of effort in keeping data consistent. This is not necessary in most cases because different versions of data can be merged when the client reads the data and merging can not happen automatic. Another goal was to not violate any Service-level Agreement (SLA).

The main advantage of Dynamo is, that there is no single point of failure because a peer-to-peer system is used to distribute data. A distributed hash table is used to manage data information and each node in the system has enough routing information to directly route a request to the corresponding node. The different nodes of the system are located in form of a ring where a consistent hashing function determines the position in the ring where the data gets stored. Multiple nodes can be located on a single physical server. The number of nodes a machine hosts, depends on the power of the machine leading to a heterogen system. To avoid the data loss in the case of a system crash, data is replicated on other nodes. A special algorithm is used, to determine the nodes for replication considering that these nodes are located on different physical machines in different data centers. To manage different versions of data, vector clocks are used. The advantage of this system is, that data can be written even if the latest version of the data set can not be found. In this case, an older version gets modified leading to a new branch of the data. When a client then requests the data, both versions are returned and the client has to merge them back to a single version. The interface of Dynamo only provides two methods, a get method to request data and a put method to store new data. Both methods use a context to manage metadata information. In this metadata, for example the vector clock is stored. For maintaining consistency, a special protocol is used. This protocol allows users to configure two values R and W. R is the minimum number of nodes that must successfully participate to a read operation and W is the minimum number of nodes that must successfully participate in a write operation.

In the Dynamo data store, each node consists of three main parts. The request coordinator, membership and failure detection and a local persistence engine. The components are developed in Java and a Berkley Database Transactional Data Store is used to persist data. Depending on the size of the data, also MySQL can be used but in general Dynamo handles small data sets.

Column Oriented [37]

Column Oriented data stores use columns to manage data. This means, that data in a column is stored together and data distribution is based on columns. The advantage of this approach is, that unlike to relational databases, rows in a column oriented storage can have different columns. So each row does only have the columns necessary to store the specific data set. Another difference to traditional relational databases is, that columns do not have types. This means, that different sorts of data can be stored in one column no matter what type the data has.

In Figure 2.4, we can see the data model of a column oriented data store. In this example, we have two column families “person” and “student”. Column families are used to group columns. This means, that columns in one column group are logically connected and will normally be read together. The data store tries to save the members of the family together to increase performance when reading data. The family student for example consists of two columns “firstName” and

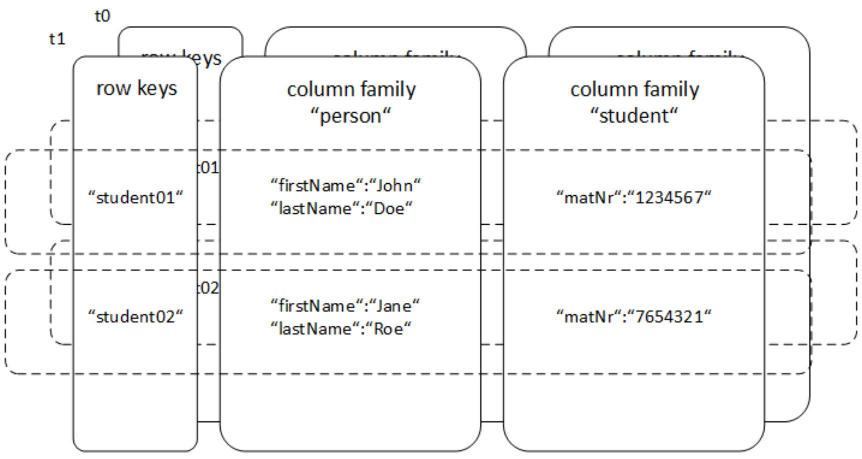


Figure 2.4: Column data model

"lastName". In our example, we have two rows which are identified by a unique row key. The row key can be compared to the primary key of a relational database and is used to identify the rows inside a table. An entry in the data store can be selected by a combination of row key, column family and column name. To get for example the first name of the first student, we have to write something like "student01/person:firstName". Another interesting aspect we can see in our example is, that for each column there exist different versions denoted t0 and t1. So if an entry in a column is updated, the previous values will be saved and a newer version of the column is created. In most data stores, users can control how many versions are stored before they will be deleted.

The most popular implementation of the column oriented data stores is BigTable which was designed by Google. In contrast to the Amazon Dynamo data store explained in the last section, BigTable is designed to store structured data [34]. The explanations given in this section, combined with a detailed description of the system can be found in [22].

The main idea behind BigTable was to design a distributed storage system for managing structured data which is scalable, reliable and has a high availability. To reach this goal, there is no support for a full relational data model and transactions are limited. The data model consists of a multidimensional map which is indexed by rows, columns and timestamps. Transactions in the data model can only be applied on rows. Data gets stored as uninterpreted strings. Each read or write operation on a single row is atomic. Rows are ordered and managed in ranges of rows so called tablets. This means, that clients can increase performance by storing related data in the same row range. Columns are grouped using so called column families where each family usually consists of columns storing the same type of data. A cell in BigTable can store different versions of the same data where each version is identified by a time stamp. Users can decide how many versions of a certain data set are stored. The Google File System (GFS) is used to store log and data files. Data gets persisted in a special data format the so called SSTable file format. For managing the different servers in the system, the high available and persistent distributed lock service Chubby [20] is used. The implementation of BigTable consists of three main parts, a client library, one master server and many tablet servers. In this architecture, the master server is responsible for assigning the different tablets to the tablet servers available in the system. Each tablet server manages multiple tablets. By using the client library, it is not necessary to use the master server for locating a tablet and so load on this component is reduced. The location of the different tablets is stored in a so called METADATA tablet. A tablet is assigned to exactly one server at each time and if this server crashes the master server will quickly relocate the tablet to another server available. Special compactions are used to keep storage consumption low and to avoid transferring large sets of data over the network. The developers of BigTable have implemented a lot of improvements to increase performance of the whole system. Therefore users can group column families in so called locality groups. Families in one group are stored together, so corresponding data can be accessed faster. Another improvement is the use of two levels of caching and so called bloom filters [17]. To avoid slow performance due to logging, a special logging mechanism was implemented using a single file for each tablet server.

Google BigTable was first used in production in 2005 and is today used in about sixty projects. The advantage of this storage system is, that new servers can be added on the fly and data can be stored in a fast and scalable way. Using transactions only per row and the non-existent relational data model require developers to write programs in other ways then usually but the wide usage in Google products has shown that this is possible.

Document Oriented [37]

In document oriented data stores, data is managed in so called documents. A document consists of fields, which have a name and a value. Each document in the database can have different fields and does not require to have a special structure. A document is identified by a special id field. This field can be compared to the primary key in a relational database and is used to uniquely identify a document.

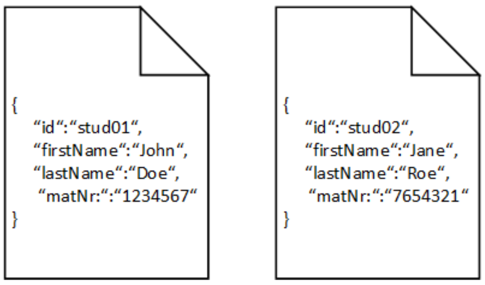

```
{
  “id“:“stud01“,
  “firstName“:“John“,
  “lastName“:“Doe“,
   “matNr:“:“1234567“
}
```
```
{
  “id“:“stud02“,
  “firstName“:“Jane“,
  “lastName“:“Roe“,
   “matNr:“:“7654321“
}
```

Figure 2.5: Document data model

Using our example from the last section, we can see the representation in a document oriented data store in Figure 2.5. As we can see, each student is represented by a separate document. Each of the documents has assigned a unique id “stud01” and “stud02” in our example. In databases like CouchDB [11] and MongoDB [68], documents are represented as simple JavaScript Object Notation (JSON) strings. This means, that a document is only required to be a valid JSON string to get persisted in the data store. Because documents do not have relations, they can be distributed across multiple servers easily.

Today CouchDB together with MongoDB are the most popular implementations of document oriented databases. Both provide nearly identical features and only differ in some special functions [50]. CouchDB for example, stores different versions of a document and therefore adds a special revision field to the document. This means, that updating a document creates a newer version of the document and does not replace the old one. Documents can be queried using MapReduce [18].

2.3 Data Management

Nowadays, a variety of different databases exists, using special data models to persists data and to provide features required for some use cases. Basically, we can group the way of managing data inside the data stores into three main categories mentioned in this section.

Relational Data Management

Today one of the most used ways of managing data is relational data management. This means, that data is stored in so called relations based on the relational model [25]. Storing data in relations, helps applications to keep their information persisted in a structured way. Together with a special language for defining the structure of the data, nearly all modern relational databases provide a powerful query language to find information in the storage. This languages are called Data Definition Language (DDL) and SQL and were defined in the SQL-92 standard [80]. In most applications today, relational databases are used to store data. Nearly all modern programming languages provide libraries for connecting to this data stores. One of the main advantages of this sort of databases is, that they guarantee consistency which is an important property for many applications. Together with transactions, this property is required in applications where critical data, for example banking data, is handled.

The relational data model requires data to be stored in a structured way [65]. This means, that data in the same relation has the same structure and properties. In most applications, this is the case and fitting the produced data in this structures works well. But often there also exist situations, in which information can not be structured this way and applying transformations is necessary to persist the data in a relational database. Often a lot of code is required to transform the data and fit it into the given structures. Another problem with this sort of data store is, that they where developed to run on single centralized servers and so distribution is difficult in general. Vendors of relational databases provide different solutions to solve this problem but often limitations are necessary to apply them [21].

Modern applications normally produce a lot of data, which in most cases is stored in relational databases. Persisting data in databases helps developers to keep their data in a central storage and to run queries on the stored data sets. Making it easy for users to write programs that access large shared data bases, is called the data programmability problem [16]. The main reason why this is not easy in general is, that often complex mappings are necessary to access the data programmatically. Different representations of data often exist because of heterogeneity and impedance mismatch. Heterogeneity is the problem, that different data sources are independently developed by different people. Because of different purposes, these data sources often use different data models to represent data. Impedance means, that logical schemas required by applications often mismatch the physical representation needed by databases. So in general, it is necessary to map data from the logical view of applications, to the physical representation in data stores [3]. The definition, development and testing of these mapping in general takes about 30-40% of the total development effort [57]. Many applications also use custom solutions for this problem and so there is a lot of code duplication between different software projects. Often the mapping code is not reusable because specific mapping strategies are implemented an these strategies only work for a specific data source or data model.

In the last years, a lot of research and development has been done to provide general mapping tools which can be applied to different data sources and data models. There are many research papers, presenting solutions to heterogeneity and impedance mismatch [32] [76] [85]. A popular solution for this problem, are so called Object-Relational Mapping (ORM) tools like Hibernate [51], Oracle Toplink [71] or the Microsoft ADO.NET Entity Framework [2]. These middleware frameworks allow developers, to define mappings with the help of special mapping structures or annotations. The model defining the relations between the objects and the database is called the entity model and is a refinement of the Entity-Relationship (ER) model of Chen [23]. So the entity model is placed between the object world of the applications and the database [69]. The main advantage of this solution is, that programmers can think in terms of objects and the relationships between this objects and do not have to think about the representation in the database. The mapping is completely handled by the middleware. Even inheritances can be modelled and are mapped to the database by the ORM tool when specified in XML mapping files or code annotations.

Object Oriented Data Management

In the 1990s a new sort of databases was introduced, which supported persisting objects directly without requiring any transformations. This new sort of databases, called object oriented databases, supported the object data model to store data [15,58]. This is differently to relational databases, where data is stored in relations and application programmers have to map their data. For developers, this sort of data management is more natural because most programming languages are also object oriented. The object oriented model supports a lot of powerful methods, for example inheritance which can be used to model complex structures. This properties expected object oriented databases to replace relational databases [47].

Today object oriented databases are not very popular and most applications still use traditional relational databases. One of the reasons for this development is, that object databases often use relational databases as underlying system and only a new layer is added. So many of the database vendors like Oracle offer object oriented features for their database systems [70]. Often this extensions add some limitations or provide a less powerful query language. Object oriented features like inheritance where not supported by some of the implementations or performance issues occurred. In some cases, the problem of mapping data was not solved and was moved to the database administrators instead [47]. So impedance mismatch still exists and with the development of ORM object oriented databases lost importance.

An especially in the field of embedded systems very popular object oriented database is db4o [33], which can be directly started from the application or as a standalone server. It is not a Database Management System (DBMS), but has a very small size what is the reason why it is used in embedded systems. Basic features like ACID and Caching are supported in a limited version [47]. Different methods to query data are supported by db4o. For example native queries or Query-By-Example can be used to find data in the data store. The problem with all of this methods is, that in each case an application is required. There is no possibility to run queries like SQL to analyse data.

Data Management in the Cloud

With cloud computing getting more and more popular, it becomes more important to find suitable solutions to manage data in software deployed in the cloud. Programs which manage large amounts of data are generally good candidates for moving into the cloud. Different database providers are working on solutions to make data management easier and to provide databases as a service [46]. The main problem that has to be considered when developing cloud applications is, that not each type of software is fitting in the cloud paradigm. Basically we distinguish between two types of software, transactional data management software and analytical data management software [1]. In most cases, it does not make sense to move software that requires transactional data access to the cloud. Modern cloud data stores, often reach scalability by relaxing consistency and so this products are not suitable for transactional applications. On the other hand, data analysis software with a high number of reads, is a good candidate for the cloud. For creating analytical reports, only snapshots of data are required and consistency is not necessary in general. As mentioned in [46], database management systems in the cloud should have the following properties:

- **Efficiency**
  Cloud resources are paid per use and so efficiency is very important because faster services are cheaper this way.
- **Fault Tolerance**
  Fault tolerance means, that the failure of a single node does not crash the whole system. Especially in the case of large queries running on data in the cloud, it is necessary that the failure of a single node does not mean, that the complete query has to be repeated.
- **Ability to run in a heterogeneous environment**
  Different nodes in a cloud environment often have different performance. Modern cloud

systems should be able to analyze performance and to avoid the case that the whole system is slow because of bad performance of a single node.

- **Ability to operate on encrypted data**
  For security reasons, sensitive data is encrypted and so cloud analysis has to be able to work with encrypted data as well.

- **Ability to interface with business intelligence products**
  Often different business intelligence products are uses to analyze data because analysts are not familiar with the underlying technology to receive data manual. So modern cloud database systems should provide a way to use this software.

Today there exist different approaches to fulfil all of the requirements but there is still a lot of research necessary for practical use.

One of the main problems in modern cloud computing systems is, that there does not exist a standard Application Programming Interface (API) for administrating and managing cloud software. Different vendors provide various interfaces and libraries to manage their clouds and often this software is proprietary. This means, that it is not possible for customers to change their cloud provider without adapting their applications hosted in the cloud. One solution to solve this problem are the Open Cloud Computing Interface (OCCI) [67] specifications and the Cloud Data Management Interface (CDMI) [24] standard. This standards and specifications try to develop a general interface for controlling and managing cloud services from different providers. The advantage of this strategy is, that users can move their applications without any changes.

CHAPTER 3

Related Work

In this section we will discuss different research papers in the field of cloud storage and cloud data management.

3.1 File Systems and APIs

One of the most important parts for modern cloud storages are scalable file systems and powerful APIs to access them. So a lot of research has been done to find the best methods for persisting data on disks and to provide access to these data sets.

Most cloud systems today use specialized file systems to support high performance and scalability for applications [9]. The main focus of these storage systems is to support the following features:

- scalability
- high reliability
- low cost
- efficiency

Providing these properties, modern file systems are built to support the use of read intensive applications. These programs often use frameworks like MapReduce to perform data analysis on large sets of data distributed on different storage servers. The problem is that sometimes users also want to run normal write intensive applications in the cloud. In contrast to programs developed for the cloud, these applications often rely on the Portable Operating System Interface (POSIX) which is actually not supported in most cloud file systems. Actually, there is a lot of research for building a storage stack which also supports applications not developed for the cloud [9]. One approach is to compare cloud file systems with file systems used in clusters.

Experiments have shown that using MapReduce in a cluster based system leads to bad performance. The reason is, that smaller block sizes used in these systems to minimize network traffic lead to a higher number of tasks in MapReduce because there is one task per block. On the other side, increasing block size reduces performance in traditional applications. So the solution is to find a setting where large blocks are used for cloud applications and small blocks for the rest. The concept of metablocks presented in [9] can be used to achieve this behaviour. A metablock is simply a group of blocks located on the same disk and used as unit. To fully support this grouping, allocation algorithms have to be changed to use metablocks. Non cloud applications can still use normal blocks to handle data in the file system.

Another research topic discussed in [72] is the search for a scalable API for file systems. Today, storage stacks typically provide two types of interfaces, the base object and its meta-object interface. The first category of interfaces is used to store the data objects while the second one is used to manage the metadata for the data object. In most cases, this metadata is stored together with the real data to provide additional information required for applications. Backup software for example can use metadata to remember when a specific data set was archived last. The problem with this metadata is, that often different applications need different types of information stored together with an object. With an increasing number of additional information, the performance of the whole system is decreasing. One approach to solve this problem is inspired by the use of so called non relational databases in modern web applications. Modern Data Intensive Scalable Computing (DISC) programs use these new storages to manage their data in a scalable and fast environment. In contrast to RDBMS, these data stores are very scalable and do not require data to have a specific structure. This can be done by relaxing some of the ACID properties provided by relational databases. So in most cases these systems for example do not provide consistency or atomicity. The interesting observation using these new storage systems is, that they often start looking more like a file system.

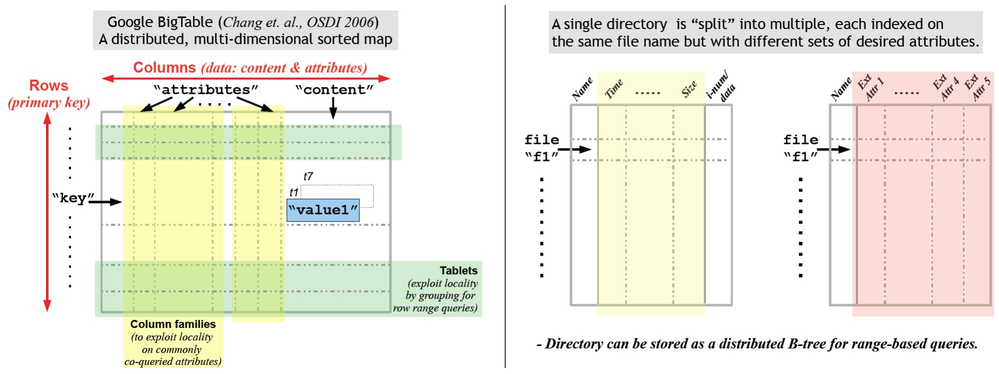

Google BigTable (Chang et. al., OSDI 2006)
A distributed, multi-dimensional sorted map
Columns (data: content & attributes)
Rows (primary key)
"attributes"
"content"
"key"
t7
t1
"value1"
Tablets (exploit locality by grouping for row range queries)
Column families (to exploit locality on commonly co-queried attributes)
A single directory is "split" into multiple, each indexed on the same file name but with different sets of desired attributes.
Name
Time
.....
Size
i-num/data
file "f1"
Name
Ext Attr 1
.....
Ext Attr 4
Ext Attr 5
file "f1"
- Directory can be stored as a distributed B-tree for range-based queries.

Figure 3.1: File Attributes in Column Oriented DB (taken from [72])

So one attempt so solve the scalability problem in file system is to use modern non relational databases to persist the data as illustrated in Figure 3.1. In this example, for each directory a separate table is created. A file in the directory is represented by a row in the corresponding table where the file name is used as primary key. In the shown example a column oriented database is used. This means, that each row can have a different number of columns storing different types of data. So the advantage is that metadata, often called extensible attributes, can be stored together with the file data in one row. Applications requiring new meta data can simply add a new column to the rows without decreasing performance or scalability.

The research papers presented in this section have shown that there are different solutions to solve the problem of modern cloud data management at the file system level. In the solution discussed in this thesis, we will use an already existing storage stack. This means, that available APIs and non relational databases will be used.

3.2 Mapping

A problem also occurring with non relational databases is the impedance mismatch between the logical representation of data in the programming languages and the physical representation. So a lot of research has been done to solve this problem and provide solutions. In nearly all modern applications using relational databases, special mapping tools are used to map the data representations. The concept of these ORM tools can also be applied to non relational data stores.

One of these new types of data stores often used in modern web applications is Cassandra which was originally developed by Facebook and is now available as an Apache project. The motivation using this new sort of databases is that they provide high performance and good scalability which is an important feature for modern programs. Cassandra therefore supports scaling over different nodes, provides consistent hashing and quorum based read and write operations. The problem using this new technology is, that programmers need o get familiar with the different properties and functions. This often requires some time to learn how to use these new databases and how to organize data. Another problem is, that often companies do not want to reimplement there software to use the new data stores because this is often a very expensive task. The best solution to avoid this is to use mapping tools supporting existing APIs like Java Persistence API (JPA) or Java Data Objects (JDO). In [42] a mapping strategy for mapping Java Objects to Cassandra using an existing API is presented. The main challenge in developing such a tool is to find a suitable mapping between the language representation and the data management in the data store. This means, that for each object type a mapping has to be defined. The framework developed in [42] solves this problem by using one column family per class. In a column oriented database like Cassandra, each row can have a variable number of rows so there is no limitation for the number of fields per class. One drawback in contrast to relational databases is, that there do not exist any relations between the different data sets persisted. This means, that object relations require a special mapping strategy to be represented in the storage. One solution presented in [42] is to use a column for the row keys from related objects. Using this strategy a good read and write performance can be reached without limiting scalability. A remaining problem using these storage systems is, that complex queries are not possible and need to be handled in the application. This solution leads to a bad performance in applications using many queries to fetch data.

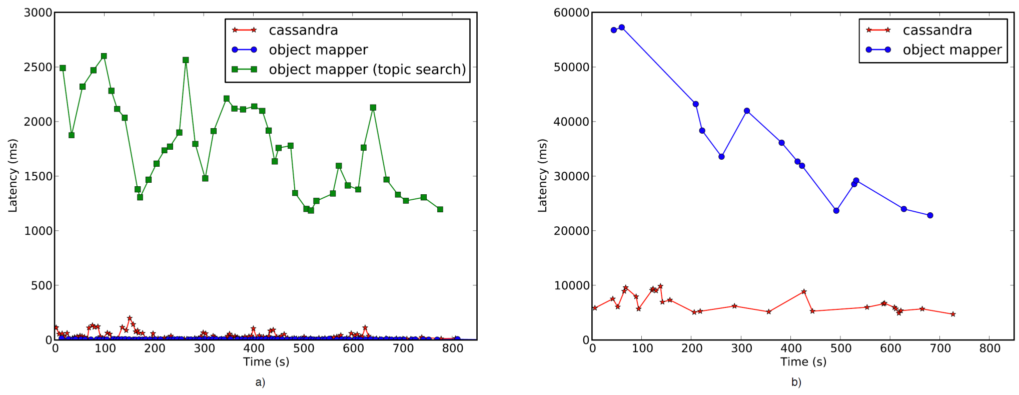


Figure 3.2: Latencies in TPC-W (taken from [42])

In Figure 3.2 we can see the benchmark of TPC-W. The performance of the mapping tool depends on the operation executed. In (a) we can see that fetching data based on indexed columns has a good performance while for example selecting the best seller list leads to fetching thousands of objects resulting in bad performance.

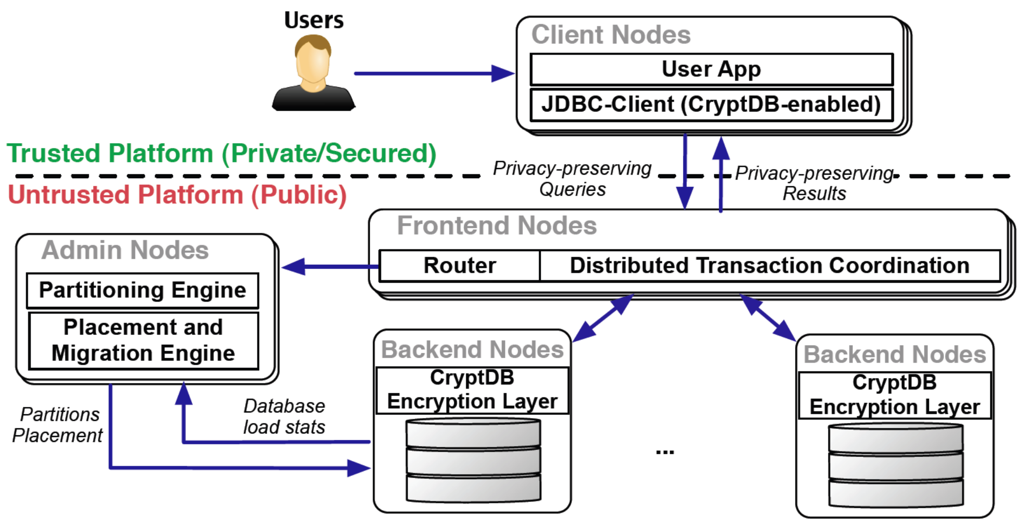
Instead of developing completely new mapping frameworks, existing tools widely used in modern applications can be modified. One of these tools is the famous ORM tool hibernate which is used to map Java Objects into relational databases. The advantage of modifying existing software is, that applications already using it can be migrated without any rewriting. It is also possible to reuse most of the modules in the framework resulting in faster development. As an example of such a modification, in [86] hibernate is used to implement an ORM like mapping framework for the non relational data store riak. Riak is a key-value store managing data in tuples existing of a key and a value field. The key is used to uniquely identify a data set in the database. A special feature of riak is that relations between key-value pairs can be defined. This is possible by defining so called links to the related tuples. The relations can then be resolved by following the links persisted with the key-value pair. Riak already supports operations to perform resolving relations which can be used by applications. For better data management, riak uses buckets to manage tuples. A bucket is simply a group of key-value pairs which is managed as an unit. Special operations are provided to get for example all members of a bucket. The implementation presented in [86] uses one bucket per class and one key-value pair for each object of the class. The fields of the object are serialized as JSON string and persisted in the value field of the corresponding tuple. Links are used to represent relations between objects. To perform this mapping, a new module was developed which was then used in the hibernate components to persist and retrieve data from riak. The most challenging part of modifying hibernate is the session component. The session is used to manage the states of objects in the framework and to perform modifications when necessary. Instead of completely rewriting this component or parsing plain SQL, the existing implementation was modified to use the riak mapping module. The modified hibernate version was then compared to the relational database version. The results show that the riak solution is very fast and scalable in contrast to the RDBMS approach. The reason for this result is that riak can easily be distributed across various servers and uses consistent hashing for managing keys. The use of consistent hashing avoids modifying the complete index when a new entry is persisted, as this the case in the relational databases. The implementation used in [86] supports just a small subset of the features provided by hibernate. To get all features work, some of the modules need to be reimplemented in future versions of the tool.

As we have seen in this section, it is possible to develop a mapping strategy for non relational databases. The approaches presented use existing libraries or APIs and only support one type of data store. In the solution developed in this thesis, a new framework is developed from scratch. The resulting implementation is designed to support different types of databases and also provides a strategy for easily adding new ones.

3.3 Databases in the Cloud

To avoid developing new mapping tools or modify existing ones another solution often described in research papers is to provide special implementations of relational databases for the cloud or to develop a database on top of a non relational data store.

For running a relational database in the cloud, there are basically three challenges that need to be solved: efficient multi-tenancy, elastic scalability and privacy. One solution to solve these problems is presented in [27] where a special transactional cloud framework called the relational cloud is developed. The implementation provided supports Database as a Service (DBaaS) and features known from relational databases. Reducing the number of required machines to get efficient multi-tenancy is reached by using multiple logical databases instead of virtual machines. This means, that there is a single database on each server hosting multiple logical databases leading to a smaller number of total servers. To scale data across the multiple instances, a special workload aware partitioner is used. This component uses graph partitioning to analyse queries and perform automatic data item mapping to the nodes based on there workload. To preserve privacy of the data persisted on the nodes, a special technology called CryptDB was developed. This module uses different protection levels to manage data access and to grant only the required type of control. Queries performed only work on encrypted data and a special JDBC driver is used to hide encryption and decryption tasks from the users. The overall architecture of relational cloud is shown in Figure 3.3. As we can see, the user performs the query with the modified JDBC driver which forwards the request to the frontend node. The routing component of this node then distributes the data over the different backend nodes based on the load stats from the partitioning engine. The backend nodes decrypt the data, perform the query and return the encrypted result back to the frontend node. The frontend node merges the results and returns the merged data to the JDBC driver which decrypts the data for the user. Experiments used to evaluate this approach have shown, that concepts used in relational cloud produce a total throughput reduction of about 40% compared to common databases. But on the other hand, using these technologies provides a scalable and relational database with a high level of security.



Users
Client Nodes
User App
JDBC-Client (CryptDB-enabled)
Trusted Platform (Private/Secured)
Untrusted Platform (Public)
Privacy-preserving Queries
Privacy-preserving Results
Frontend Nodes
Router
Distributed Transaction Coordination
Admin Nodes
Partitioning Engine
Placement and Migration Engine
Backend Nodes
CryptDB Encryption Layer
Backend Nodes
CryptDB Encryption Layer
...
Partitions Placement
Database load stats

Figure 3.3: Relational Cloud Architecture (taken from [27])

Another approach to get relational features in the cloud is to build a relational database on top of an existing data store. An example implementation using Amazon S3 is described in [19]. The basic idea is, that most of the logic used in modern database systems is located on the clients and S3 is used as a shared disk. Because the performance of S3 is higher with bigger chunks of data, pages are used to group records. A so called page manager is used to coordinate read and write requests to Amazon and to buffer pages on the local disk. The record manager manages records on pages and coordinates free space. The main goal of the implementation is to keep the system scalable and available. Any client can fail at any time and no other client should be locked in this case. Therefore, no locks are used to protect pages or records. To provide these features, ACID properties are not supported or only partially. Managing free space in the system is done by using a B-tree. For keeping interaction with S3 low, clients use a local buffer based on a Time To Live (TTL) protocol. The protocols used in the implementation make extensive use of redo logs and undo logs. Redo logs are idempotent to avoid errors when records are applied twice or more often. To support transaction like features, a special commit protocol shown in Figure 3.4 is used. Therefore another Amazon Service called SQS is required. A commit is done in two steps:

- The client generates log records for all the data sets changed or created and sends these logs to SQS.
- The logs submitted are applied on the pages stored in S3. This step is called checkpointing.

Logs are stored in so called Pending Update (PU) queues. One of the most challenging parts using this protocol is, that at no point two clients should concurrently carry out the checkpoints on the same PU queue. The solution presented to solve this problem are so called LOCK queues. This special queues are used to grant locks on the corresponding PU queue.

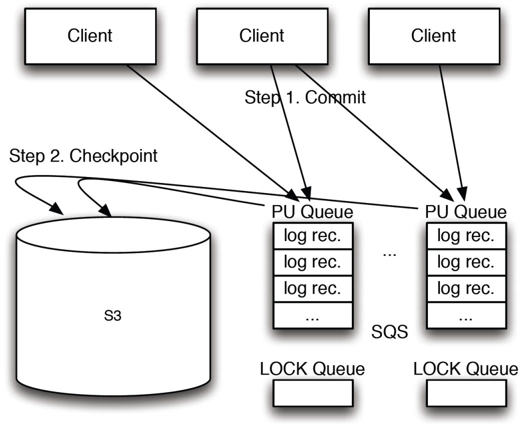


Figure 3.4: Basic Commit Protocol (taken from [19])

Relational Cloud and the S3 database try to bring well known features from relational databases to the cloud. The problem is, that based on the CAP theorem, there can not be full support for all these features without limiting scalability. But this is an important feature for modern web applications and with an increasing number of devices accessing services in the internet, it is getting more and more important. So the focus of this thesis is to use scalable data stores in modern applications without limiting scalability. Therefore, existing databases are used and relational features are omitted.

CHAPTER 4

Design and Implementation

In this section the basic design and implementation of the prototype will be explained and presented. In the first part, the overall architecture will be explained followed by an example used in the next sections to describe the design decisions.

4.1 Architecture

The framework presented in this section is designed to be modular and easy to extend. Nearly all components of the implementation can be replaced with custom solutions and support for new data stores can be provided easily. The overall architecture is shown in Figure 4.1. The components in the dashed rectangle are part of the framework and generally not designed for direct usage by client applications. Programs using the library use the public API to persist or receive data. Therefore the interface is designed to provide database independent methods. The implementation is part of the core component and uses different modules to perform the requested operations in the corresponding data store. The main task of the core is, to connect the other components and to provide a general implementation for the API used by the client applications. To get the information necessary to interact with the databases, a special configuration manager is used. This module parses the configuration file provided by the client software and provides this data to the core. Configuration files are XML files with a special format described later in this chapter. One of the most important parts in the framework are the drivers used to communicate with the different data stores. A driver is responsible to handle exactly one specific database and to abstract the details from the core. Therefore drivers implement a special interface used by other components to perform read and write operations. The used drivers are selected in the client configuration file. This means, that one database can have different driver implementations giving the client the choice to choose one of them.

Data store communication can be done in different ways depending on the provided APIs. Generally there exist two main types:

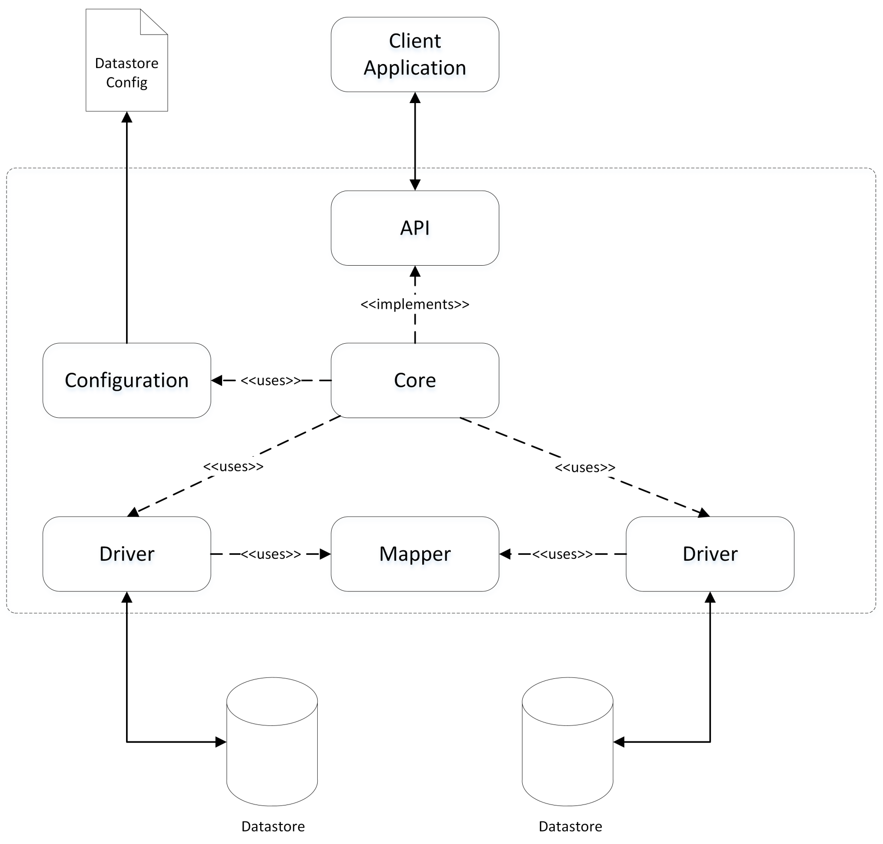


Figure 4.1: Overall Architecture

- Representational State Transfer (REST) APIs
- Native APIs

Today, nearly all of the available NoSQL databases provide a REST interface to manage data. The advantage of this approach is, that normal HTTP Requests are used and no special library is required for communication. Some data stores also provide native APIs in the form of normal Java interfaces. This is especially useful for high performance requirements because no communication overhead is produced. Another important part in data management is to solve the impedance mismatch between the programming language representation and the data model used by the data store. This is done in the mapper component used by the different drivers. To support various database interactions, different mappings are provided . New mappings can be added easily by implementing a special mapper interface used for encapsulation [63].

The general data flow is shown in Figure 4.2. The client calls an operation, in this example save, on the core component using the public API. The core component then requests the configuration management unit to load the required data store configuration, provided by the caller. The

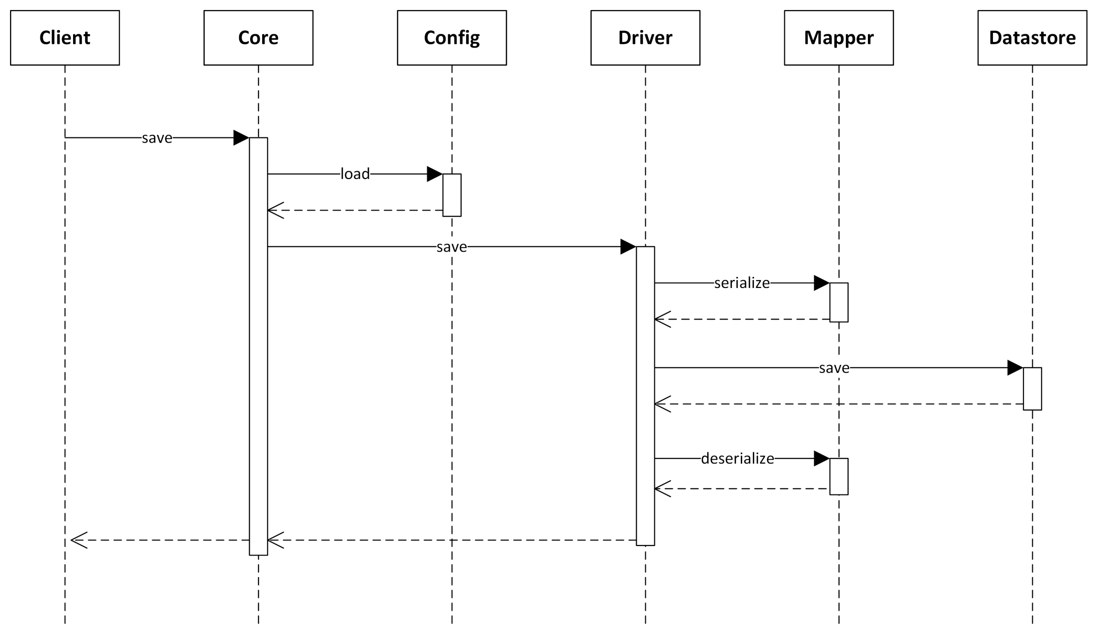


Figure 4.2: Dataflow of a save operation

request together with the configuration is then passed to the driver specified by the client. To persist the data in the data store, the data sets are serialized to the corresponding representation. This task is performed by the mapping unit responsible for the corresponding data model. Now, the data is submitted to the database and the response is deserialized by the mapper used before. The driver now returns the result to the core which then forwards it to the client application. This workflow is similar for all provided operations and data stores.

To explain the individual steps when performing a specific data store operation, we will use the example shown in Listing 4.1

API

To offer support for many different data stores and data models, the provided API only defines methods available on all current databases. This functions are sufficient to read and write data and are designed to be lightweight and easy to use. The currently supported operations are listed in Table 4.1 and can be used with all drivers. Data types used in the persisted objects need to be supported by the corresponding mapping framework, otherwise an exception will be thrown.

Objects passed to the data store API need to provide a special id field for identifying the instance in the data store. The identifier field is a simple string field annotated with a special annotation provided by the library. In our example in Listing 4.1 the field named `id` is used to hold the

```
public class Person {
    @DatastoreId(strategy = IdStrategy.AUTO)
    private String id;
    private String firstName;
    private String lastName;

    public enum Gender {
        MALE, FEMALE
    }
    @Adapters({
        @Adapter(targetClass = JsonElement.class,
           adapterClass = JsonGenderTypeAdapter.class),
        @Adapter(targetClass = HbaseCell.class,
           adapterClass = HbaseGenderTypeAdapter.class)
    })
    private Gender gender;
}

public class Main {
    @DataSource(name = "couchdb")
    private static Datastore datastore;

    public static void main(String[] args) {
        Person person = new Person();
        ...
        datastore.save(person);
    }
}
```

Listing 4.1: Data Store Example

Method	**Description**
void save(Object object)	Persists the given object in the data store .
<T> T find(Class<T> objectClass, String id)	Returns the object with the given id and maps it to the specified type. If no object is found an exception is thrown.
void delete(Object object)	Deletes the given object from the data store.
void update(Object object)	Updates the given object in the data store.

Table 4.1: API methods

data store identifier. Another interesting detail in this code is the strategy parameter used by the `@DatastoreId` annotation. The value of this parameter tells the framework if the corresponding id, used to identify the object in the data store, is generated automatically, by the data store, or is provided by the user. In the shown example `IdStrategy.AUTO` defines that the identifier is set by the database. When setting the field manually `IdStrategy.MANUAL` should be used as strategy. The default behaviour is the automatic identifier generation which will be used when no specific strategy is stated.

So the only requirement for an object, which should be managed by the library, is to have a string field annotated with `@DatastoreId`. This condition holds for all methods provided by the public API.

Core

The component implementing the public API is the core module. This part of the library validates the method calls and forwards the requests to the corresponding driver implementation.

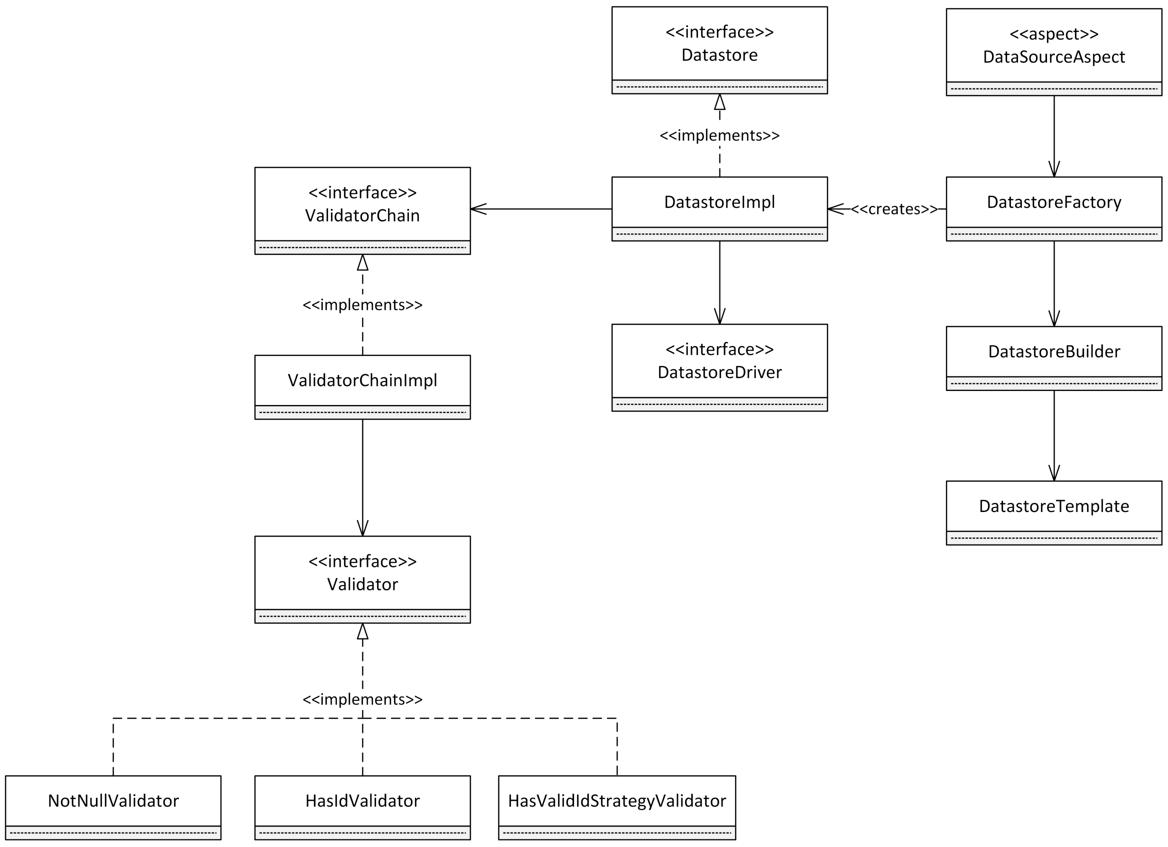


Figure 4.3: Architecture of the Core Component

The architecture of the core component is illustrated in Figure 4.3 showing the connection between the individual classes. The main part of this module is the `DatastoreImpl` class implementing the public API. The different methods are responsible for validating the input parameters and performing the corresponding driver methods.

For validating the different parameters passed to the API methods a special validator chain is used. Basically a validator chain is a simple collection of validators, implementing the `Validator` interface, which are applied successively. If one of the specified validators in the chain fails, an exception is thrown containing the reason. Therefore validators provide a special method returning a readable error message, used to inform the user about the reason. In the class diagram we can see that there are currently three different implementations used by the core component.

After validating the method call, the corresponding driver is called to perform the data store request. To avoid code duplication in the different implementations, the core handles receiving and setting id fields on objects. For both identifier strategies drivers must provide an appropriate method.

Another important task of the core component is to create `Datastore` instances and to provide them to the client. Therefore two different techniques can be used, dependency injection [75] and instance creation via a factory class [83]. When using the first approach, AspectJ [59] is used to create data store instances and to inject them into the annotated client fields using the `DataSourceAspect` aspect. In our example from Listing 4.1 the aspect is applied to the field with the name `datastore` in the class `Main`. With the help of the name parameter in the annotation, the `DatastoreFactory` is then used to create the required instance. An alternative for clients, to avoid aspects, is to directly call the factory class.

An important point to note when working with `Datastore` instances is that they are generally not thread safe. As if working with entity managers in JPA, they should only be used for one unit of work an not be passed around across different objects. This is not necessary because each object can inject an instance of the same data store. To avoid errors when providing instances to the client, templates are used to build the corresponding `Datastore` objects. Templates hold the information required to create `Datastore` instances and are provided by the configuration unit described later in this chapter.

Driver

Drivers are responsible for data store communication and act as an abstraction layer between the core component and the different data stores. One implementation is therefore responsible for communication and management of exactly one specific data store. To hide the details from the upper layers, the core modules, a special interface, the so called driver interface, is used. The connections to the data store itself can be of any type but generally there exist two main types, native and REST drivers. All information required to establish a connection is provided by the core and can be used by the different driver implementations. If special properties are required to control the behaviour a driver can use specific property files with the help of the libraries property management unit explained later in this chapter.

The basic architecture of the driver component is shown in Figure 4.4 illustrating the current implementations. Actually the library has integrated support for three different data stores using native and REST communication. As already mentioned before, each driver has to implement the `DatastoreDriver` interface used for communication with the core module.

Referring to our example in Figure 4.1 we will first explain the architecture of REST based driver implementations. In the last chapter we already mentioned, that the core component

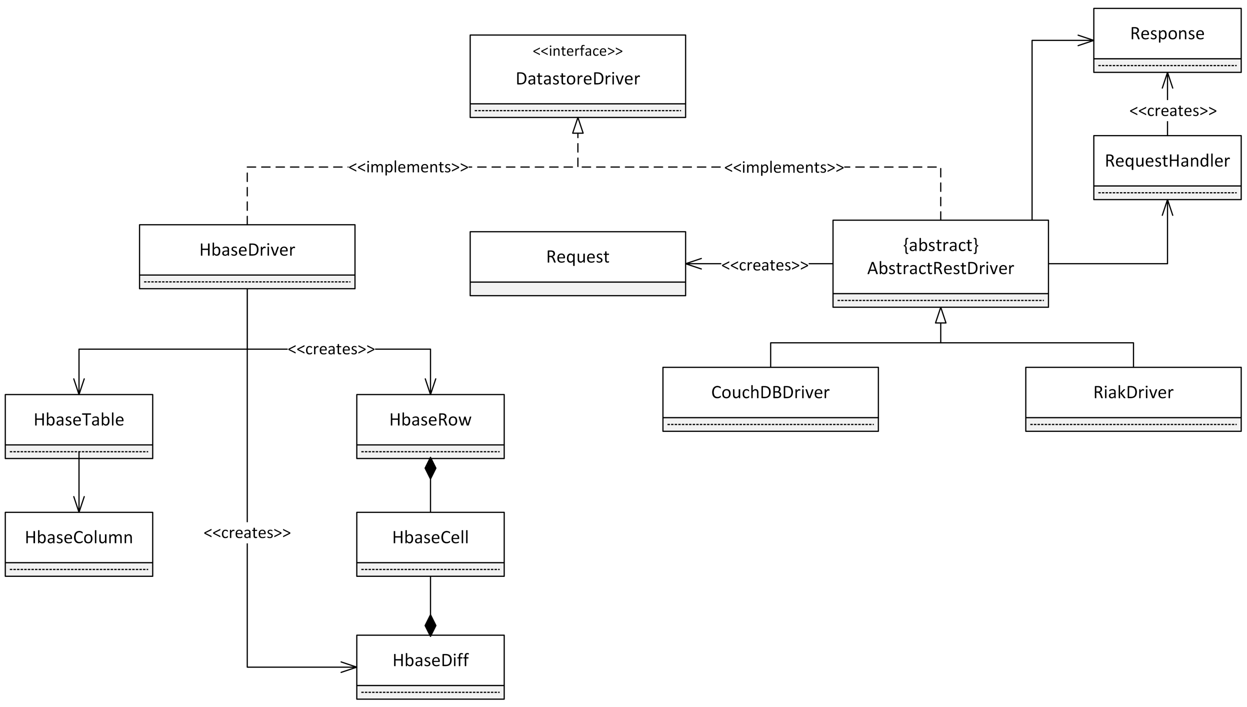


Figure 4.4: Driver Architecture

calls a specific method for each operation on the selected driver. So the first thing to do when designing a driver is to map these function calls to the corresponding data store operations.

Operation	**Type**	**CouchDB**	**Riak**
Save (Generated ID)	POST	/somedatabase/	/buckets/bucket/keys
Save (Manual ID)	PUT	/somedatabase/id	/buckets/bucket/keys/key
Update	PUT	/somedatabase/id	/buckets/bucket/keys/key
Find	GET	/somedatabase/id	/buckets/bucket/keys/key
Delete	DELETE	/somedatabase/id?rev=1582603387	/buckets/bucket/keys/key

Table 4.2: REST operations

Table 4.2 for example shows the mapping form the driver methods to the data store operations for Riak and CouchDB, which are both supported by the library. As we can see both databases use similar functions for the different operations, leading to nearly identical implementations. To avoid code duplication it is therefore reasonable to reuse classes and to abstract different parts of the implementations. So the abstract class `AbstractRestDriver` contains functions used in both drivers. The differences are realized by extending the abstract class as this is the case by `CouchDBDriver` and `RiakDriver`. Requests shown in Table 4.2 are represented by the `Request` class using the builder pattern [83] to build the concrete requests. Handling the different URL patterns is realized by requesting the concrete driver class. So each of the REST drivers has a corresponding method to return the required URL pattern to the abstract class. In our example, the abstract driver will request the `CouchDBDriver` to return the URL for saving objects with automatic id generation. Because patterns for data stores can change over time, as this was already the case with Riak, URL patterns are managed in driver specific property files as templates. When loading the property, the template is then filled with the corresponding data to get the concrete URL. Continuing with our example, the returned URL is then used by the `Request` class to build the request which is then forwarded to the `RequestHandler` class. After performing the desired data store action the response is received and validated. In case of an successful execution the result is forwarded to the core component, while in case of an error an exception containing the reason is thrown. The described steps are similar for both REST drivers and there functions.

The second category of drivers are so called native drivers using third-party libraries for communication with the data store. The `HbaseDriver` class shown in Figure 4.4, used for managing HBase [12, 41] data stores, is such a driver. Data in HBase is managed in tables consisting of different rows which are composed of columns. For abstracting this data model the helper classes `HbaseRow` and `HbaseCell` are use by the driver class. These helpers are used to represent the data stored in the database and to perform the corresponding operations. The details of mapping the data to these classes is explained in the mapping section later in this chapter. Communication is performed with the help of the `HbaseTable` and `HbaseColumn` classes shown on the left side of the Figure. Therefore `HbaseTable` is used to perform operations on tables, while `HbaseColumn` is used for column related tasks. When using our example from Listing 4.1 the driver in this case would first map the `Person` object to the corresponding HBase representation. Now the table helper class is used to lookup the corresponding table or create it if it not already exists. After the required table is found, the column helper is used to create the columns for the row representing the object. A special case in HBase is the update operation because the native library actually does not support automatic update generation. This means, that all actions required to update a given row must be determined manually. For this task the `HbaseDiff` class is used to represent the single steps which must be applied to get the new row.

Adding a new driver is very easy and can be done by simply implementing the driver interface. If the new data store provides a REST based interface, the driver can extend the abstract rest driver to avoid code duplication. By doing this, the new subclass has to provide the URLs listed in table 4.2. In case of a native driver the necessary connection has to be established by the new implementation. A special properties file [55] can be used to manage configuration options without hard coding them. The mapping between the database representation and the programming language has to be done by the driver. For REST drivers the built-in JSON [54] mapper can be used to serialize and deserialize the data sets. The implementation and design of the mapper module, and how to implement a new mapping strategy, will be explained in the next section.

The advantage of the driver architecture is, that it is designed to easily add new drivers without changing any core functions. By providing a general interface, it is also possible to give the client application a configuration option for selecting the driver to use which will be explained in the property management section later in this chapter. On the other side it is also possible, that different drivers contain the same code leading to duplications. To overcome this problem, an abstract rest driver was defined implementing code required by the subclasses.

Mapping

The mapping module is a central component of the framework used to solve the impedance mismatch between the different data store representations and the programming language. Therefore a special mapping architecture was designed to support the various existing data models and to assist re-usability of existing strategies. So proven strategies can easily be used by other drivers to map the data and to avoid reinventing the wheel.

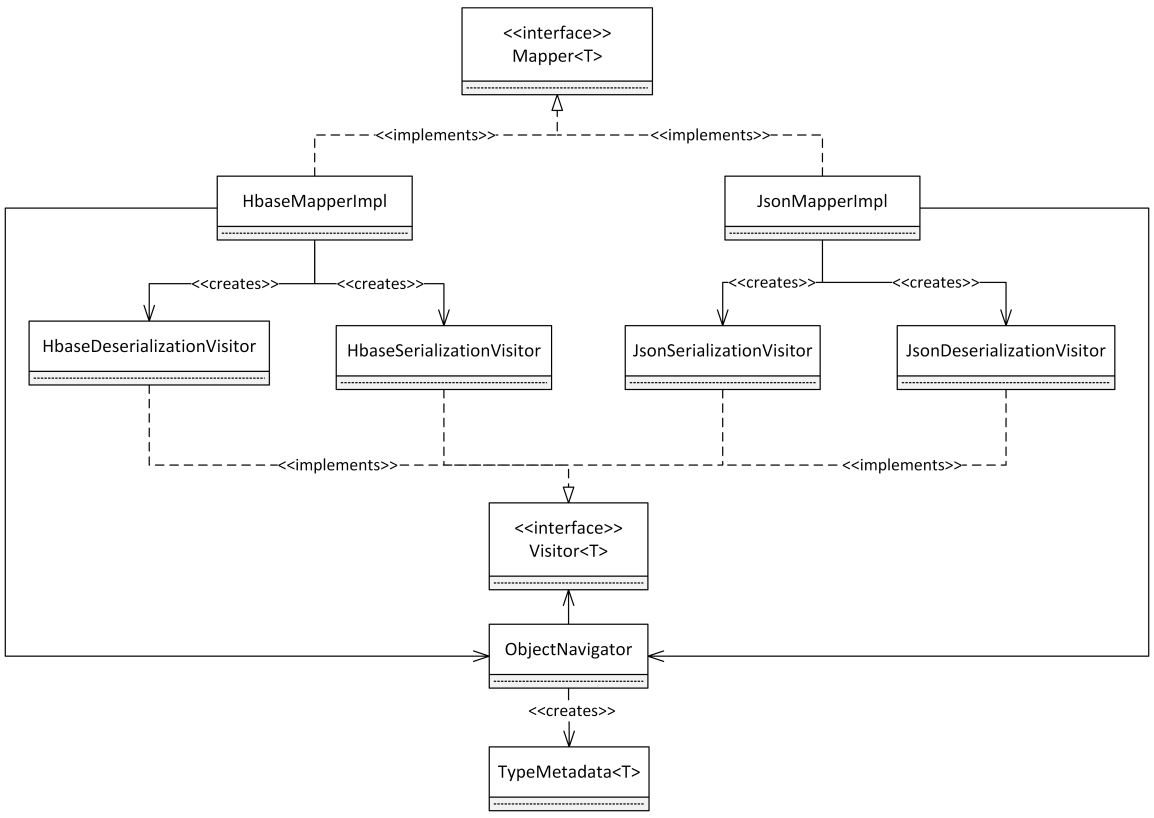


Figure 4.5: Mapper Architecture

The general architecture of the mapping component together with the provided implementations is shown in Figure 4.5. To abstract the details of the concrete mappers, and to provide a general API for the drivers, the `Mapper` interface is used. To support different strategies with various of outcomes the provided methods are generic. The implementations then define the type of the objects they serialize or deserialize.

Using our example from Listing 4.1 we will first explain the design and architecture of the JSON mapper shown on the right side of Figure 4.5. The starting point for the mapping process is in this case the class `JsonMapperImpl` implementing the generic mapper interface. So this concrete implementation is used to map our `Person` object to a corresponding JSON string.

```
{
    "firstName":"John",
    "lastName":"Doe",
    "gender":"MALE"
}
```

Listing 4.2: Expected JSON String for Person

The desired outcome of the transformation is shown in Listing 4.2 where each field of the object, except the id, is represented as JSON element. So transforming a given object to an alternative structure requires iterating all fields and applying a transformation function to each field in the source object. Because this is a common task, needed by all mappers, a general implementation would make sense to avoid code duplication and force clean design. Therefore the library provides the `ObjectNavigator` class to iterate over the defined fields of a given object. To apply specific functions so called visitors, based on the visitor pattern [66], are used implementing the `Visitor` interface. For passing the field type and other related information the `TypeMetadata` class is used. Generally there exist two different types of visitors, one for serializing and another one for deserializing. So in our example the `JsonSerializationVisitor` is used to transform the given object to a corresponding JSON string and the `JsonDeserializationVisitor` for the reverse case. Visitors use so called type adapters to map a specific type to the required representation, JSON elements in our case.

The type adapter architecture is shown in Figure 4.6. As we can see, the `TypeMetadata` class already detects the corresponding adapter for a given field avoiding code duplication in the visitors. A type adapter is simply a class implementing the generic `TypeAdapter` interface responsible for exactly one type of fields and one type of target objects. Each adapter belongs to one factory [83] extending the generic `AbstractTypeAdapterFactory` class. Each of the factories is responsible for managing adapters for one concrete target object specified by the type parameter `T`. To extract functionality used in all implementations the abstract class is used. Theses factories are then used by the `TypeMetadata` class to determine the corresponding adapter for a given field and to create an instance.

Continuing with our example, the object navigator calls the `JSONSerializationVisitor`

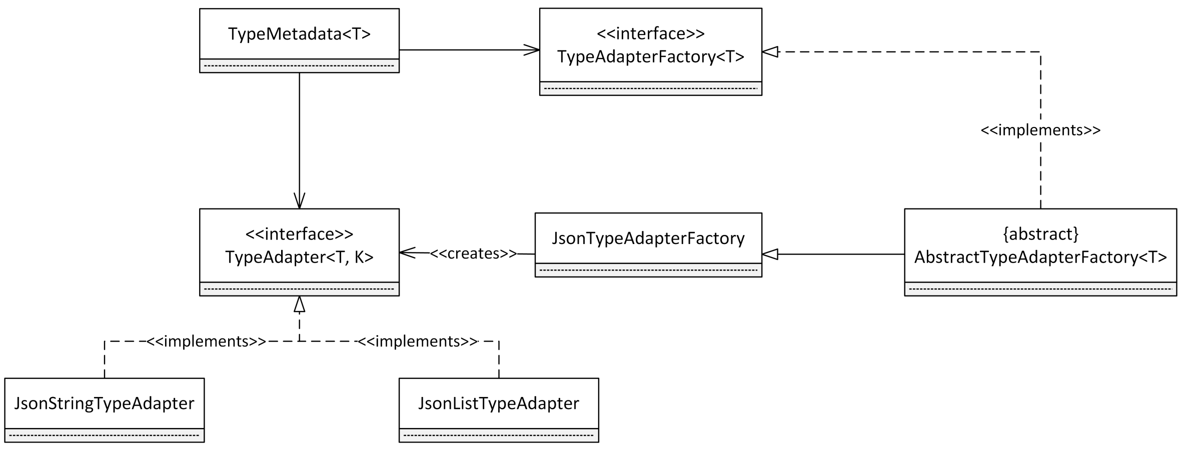


Figure 4.6: Type Architecture

for each field in the `Person` class. The instance of the `TypeMetadata` class, passed as parameter, contains the required information. In case of the `firstName` field, the type adapter determined is the libraries internal `JsonStringTypeAdapter`. This converter is then used to transform the string value into a corresponding JSON representation. The same procedure is repeated for the other fields of the object leading to the final result.

A special situation is the `gender` field of the `Person` class from our example. The annotation `@Adapters` signals the object navigator to use a set of specific adapters instead of requesting the factory classes. In this way mappings can be influenced by users leading to a customized mapping. For each target object a separate mapper has to be provided using the libraries `@Adapter` annotation. The parameters used in this annotation specify the class of the target and the adapter to use.

The implementation of the custom type adapter for transforming a `Gender` enum to a JSON string is shown in Listing 4.3. As we can see, there are actually two methods required for serializing and deserializing a specific type. The first method is used to convert a `Gender` instance to a corresponding `JsonElement`. The `JsonElement` class, used by the adapters, is reused from the Gson [45] library and has the subtypes `JsonPrimitive`, `JsonArray` or `JsonObject`. In our example, the enum is simply converted to its default string representation. For transforming the enum to the HBase data model a separate type adapter, shown in Listing 4.4, is required.

Using the `HbaseMapperImpl` class to transform the `Person` objects works similar. But now the object navigator is using the HBase type adapters to transform the field types. Using our custom type adapter from Listing 4.4, the resulting HBase representation is shown in Figure 4.7.

```java
public class JsonGenderTypeAdapter implements TypeAdapter<Gender, JsonElement> {

    @Override
    public JsonElement serialize(Gender object, TypeMetadata<JsonElement> typeMetadata)
         {
        return new JsonPrimitive(object.toString());
    }

    @Override
    public Gender deserialize(JsonElement element, TypeMetadata<JsonElement>
        typeMetadata) {
        JsonPrimitive jsonPrimitive = (JsonPrimitive) element;
        if(!jsonPrimitive.isString()) {
            throw new DatastoreException("Invalid␣value␣for␣gender␣type.");
        }
        String value = jsonPrimitive.getAsString();
        return Gender.valueOf(value);
    }
}
```

Listing 4.3: Custom Gender Type Adapter for JSON

```java
public class HbaseGenderTypeAdapter implements TypeAdapter<Gender, HbaseCell> {

    @Override
    public HbaseCell serialize(Gender object, TypeMetadata<HbaseCell> typeMetadata) {
        String className = typeMetadata.getParent().getClass().getSimpleName();
        return new HbaseCell(className, typeMetadata.getFieldName(), Bytes.toBytes(
            object.toString()));
    }

    @Override
    public Gender deserialize(HbaseCell element, TypeMetadata<HbaseCell> typeMetadata)
        {
        byte[] value = element.getValue();
        String name = Bytes.toString(value);
        return Gender.valueOf(name);
    }
}
```

Listing 4.4: Custom Gender Type Adapter for HBase

As we can see, the default HBase mapper creates a table for each object persisting each instance in a single row. All columns of the same object are part of the same column family. If for example the `Person` object would contain a field of another class, lets say `Address`, the fields part of this class are part of another column family, address in our case. This means, a single instance of an object can be fetched with one read request leading to a high performance. By using custom adapters the default mapping can be changed to fit the required data model.

Property Management

The property management component generally consists of two parts, the data store configuration management and the driver specific property loaders. The first unit is used to load the XML

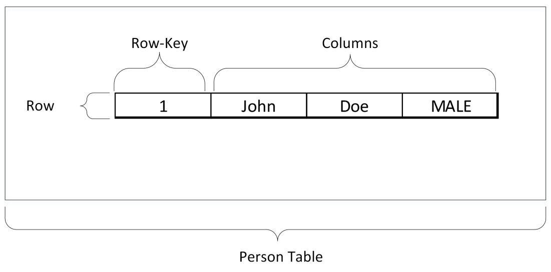


Figure 4.7: Expected HBase Representation for Person

files provided by the user to define the data store connections and settings. Each application using the library needs to provide such a file to create `Datastore` instances.

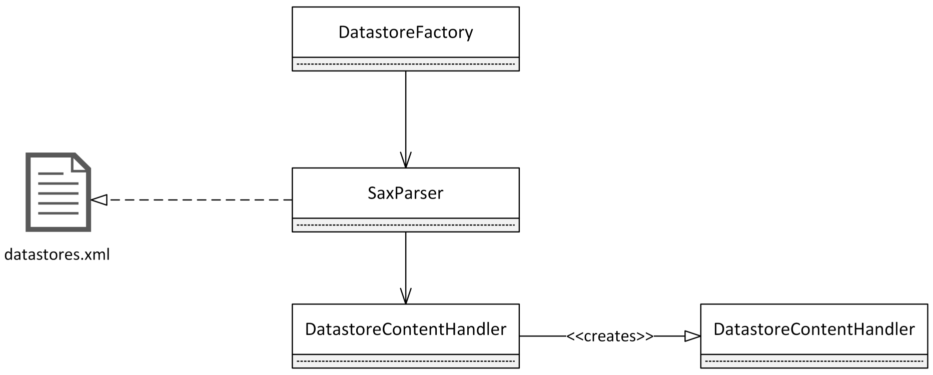


Figure 4.8: Configuration Loading Architecture

The architecture for loading the data store configurations is shown in Figure 4.8. As we can see, the `DatastoreFactory` requests the `SaxParser` to read the datastores.xml file the first time a data store instance is created. The parser reads the configuration file, which has to be placed in the projects META-INF folder, and uses the `DatastoreContentHandler` class to create the `DatastoreTemplate` instances. The factory class can then use the templates to create concrete `Datastore` instances. To avoid reading the complete configuration file each time an instance is created, the factory class caches the templates.

Referring to our example from Listing 4.1, we could for example use the configuration shown in Listing 4.5. Here we can see, that the `name` used to identify the data store in the XML file is the same used in the `@DataSource` annotation. The `name` attribute is required by the factory class to request the configuration for the data store which the client wants to use. The `host` and `port` attributes are used to establish a connection to the data store. The meaning of the `dataunit` property depends on the data model used by the database. In CouchDB for example, the data

```xml
<?xml version="1.0"?>
<datastores>
    <datastore>
        <name>couchdb</name>
        <host>192.168.56.101</host>
        <port>5984</port>
        <dataunit>cloudscale</dataunit>
        <driver>at.ac.tuwien.infosys.cloudscale.datastore.driver.couchdb.CouchDBDriver</driver>
    </datastore>
</datastores>
```

Listing 4.5: Data Store Configuration Example

unit refers to a database, used to store documents, while in Riak it refers to buckets. The last attribute in our example defines the driver to use for handling the data store.

The second part of the property management component is used to give data store drivers and users of the framework the possibility to control different features and behaviours of the system. Therefore a special property loading mechanism is used to load settings used by a specific driver.

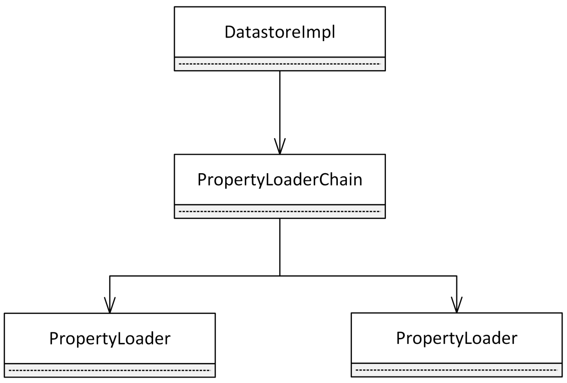


Figure 4.9: Property Loading Architecture

The architecture of the property management is shown in Figure 4.9. Each instance of the core class, `DatastoreImpl`, has a separate property loader chain. The chain itself contains different property loaders to load properties from files or databases. In each chain there is at least one loader, the so called default loader, responsible for handling properties defined in the core implementation. A single chain can have any number of property loaders or none if no special configurations are possible. Loaders only need to implement a special interface and can use any source for loading properties. So it is for example possible to implement a property loader loading its properties from a database. The default source, used in the framework, are simple property files.

An example for a property chain is shown in Figure 4.10. In 4.10(a) we can see a chain without any new loader added. So this property loader chain only consists of the default loader available in each chain. This means, loading `property1` in this case will lead to `value1` received

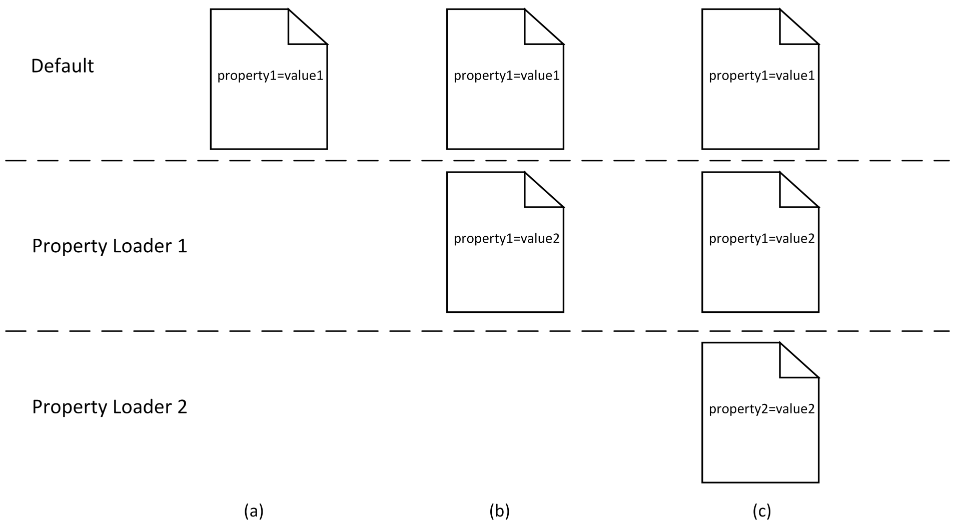


Figure 4.10: Property Loading Example

from the default property loader. 4.10(b) has an additional property loader, resulting in loading `value2` for `property1`. The last example 4.10(c) consists of the default loader and two custom property loaders. But in this case, `property1` will still get loaded from property loader 1 because the resource handled by the last property loader does not contain this property.

Giving developers of drivers and users of the library the ability to introduce new properties, or override defaults, leads to a high configurable solution. This means, that new data stores can be added easily and the corresponding properties can be managed with the help of the provided property management component. Loading values from different sources also makes it possible for users to manage their settings for example in a database.

4.2 Advanced Features

To make the library more attractive for users, some advanced features are provided by the implementation. Basically these features can be used without any additional requirements and are fully supported by all data stores.

Hibernate Integration

One of the most useful features is the hibernate integration. In some cases, users maybe want to persists parts of an entity in a data store. The problem is, that this often requires a lot of additional code to persist and load objects in the data store and all situations of the entity life cycle [78] need to be handled. To help developers with this task, a special hibernate integration was added to the framework.

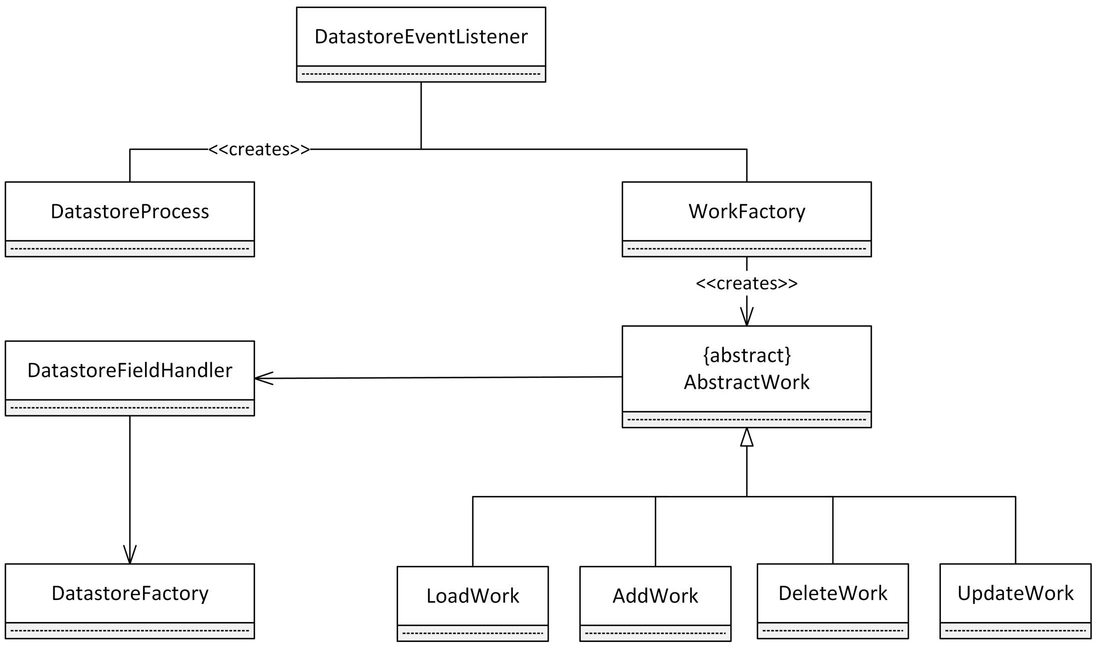


Figure 4.11: Hibernate Integration Architecture

The architecture of the hibernate integration provided by the library is shown in Figure 4.11. To react on changes of the entity, the `DatastoreEventListener` is used to listen for state transitions in the entity life cycle and to create the worker performing the necessary task for the executed Create Read Update Delete (CRUD) operation. To get the instance of the required worker class, a factory is used. Functions required by all worker classes are implemented in an abstract worker class extended by the specific workers. Handling the given fields is done by the `DatastoreFieldHandler`, which responsible for collecting the information necessary to execute the corresponding data store operation. Creating the data store instances is done by the `DatastoreFactory` explained in Section 4.1. For integration of the data store operations into the transaction, started by hibernate, a before commit process is used. This means, that all data store operations are performed within the same transaction leading to a rollback if an error occurs.

In Listing 4.6 we can see an example of a hibernate entity using the data store integration feature. In this case, the author field will be persisted in the data store named `couchdb`. All operations necessary are executed automatically by the library when calling the corresponding hibernate method. To persist the information necessary to load the data store object, a special mapping table is used. Basically this information is used to get all objects in the data store for a given hibernate entity.

```
@Entity
public class Article {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    @Datastore("couchdb")
    @Transient
    private Person author;

    //Getter and Setter
}
```

Listing 4.6: Entity Example

ID	TITLE
1	My First Article

(a)

ID	ENTITYCLASSNAME	ENTITYFIELDNAME	ENTITYID	DATASTOREID
1	Article	author	1	someID

(b)

Figure 4.12: Hibernate Data Model

The database representation of the person entity is shown in Figure 4.12. The table in 4.12(a) is used by hibernate to persist person objects. To prevent hibernate from handling the author field it should be marked as transient [51]. The second table, shown in 4.12(b), is used to map entity fields to the corresponding data store objects. Therefore a single table is created containing the mapping information for all fields of all entities. Mapping entries are created, updated and deleted automatically and should not be changed by the client to avoid errors.

To enable the data store integration in hibernate, the corresponding event listeners need to be configured and the `DataStoreModel` class needs to be added to the persistence unit. Listing 4.7 shows an example hibernate configuration using the data store integration.

Data Migration

Another advanced feature that can users help to manage their data is the data migration support. In some cases, users might want to move their data to a different data store or to an environment

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
    <persistence-unit name="cloud" transaction-type="RESOURCE_LOCAL">
        <class>at.ac.tuwien.infosys.cloudscale.datastore.hibernate.model.DatastoreModel</class>
        <class>at.ac.tuwien.infosys.cloudscale.datastore.model.Article</class>
        <properties>
            <property name="hibernate.dialect"
                            value="org.hibernate.dialect.PostgreSQLDialect"/>
            <property name="hibernate.connection.driver_class"
                            value="org.postgresql.Driver" />
            <property name="hibernate.connection.url"
                             value="jdbc:postgresql://192.168.56.101:5432/cloudscale" />
            <property name="hibernate.connection.username"
                             value="cloudscale" />
            <property name="hibernate.connection.password"
                             value="pass" />
            <property name="hibernate.show_sql"
                             value="false" />
            <property name="hibernate.hbm2ddl.auto"
                             value="create"/>

            <property name="hibernate.ejb.event.post-insert"
             value="at.ac.tuwien.infosys.cloudscale.datastore.hibernate.DatastoreEventListener" />
            <property name="hibernate.ejb.event.post-load"
             value="at.ac.tuwien.infosys.cloudscale.datastore.hibernate.DatastoreEventListener" />
            <property name="hibernate.ejb.event.post-update"
             value="at.ac.tuwien.infosys.cloudscale.datastore.hibernate.DatastoreEventListener" />
            <property name="hibernate.ejb.event.post-delete"
             value="at.ac.tuwien.infosys.cloudscale.datastore.hibernate.DatastoreEventListener" />
        </properties>
    </persistence-unit>
</persistence>
```

Listing 4.7: Configuration Example

of another vendor. Normally a lot of migration code is necessary to convert the data and save it in the new storage system, leading to increasing development costs.

Method	**Description**
<T> void migrate(Datastore to, Class<T> objectClass, String id)	Move the object of the given type with the given id to the given data store.
<T> void migrate(Datastore to, Class<T> objectClass, MigrationCallback<T> callback, String... ids)	Move all objects within the given id range of the given type to the given data store.

Table 4.3: Migration API methods

The library described in this chapter provides the possibility to move data from one data store to another without implementing any additional migration code. Actually there exist two methods to move data across databases shown in Table 4.3. The first method can be used to move single objects of a given type to another data store. To perform the mapping from one representation to another, it is necessary to provide the type of the object to migrate. For moving multiple objects the second method can be used. In this case, a list of ids can be passed to perform the migration of the corresponding objects. This task can run for a very long time and so it should not be a blocking operation. So calling the second method is starting a new thread moving the data from one data store to another. To inform the user when the work is done, an object implementing a callback interface can be passed.

```java
public class PersonMigrationCallback implements MigrationCallback<Person> {

        final Logger logger = LoggerFactory.getLogger(this.getClass());

        public void onSuccess(List<Person> result) {
                logger.info("Successfully migrated {} person objects.", result.size());
        }

        public void onError() {
                logger.error("Error migrating person objects.");
        }
}
```

Listing 4.8: Migration Callback Example

Listing 4.8 shows an example of a simple migration callback for person objects. An object implementing the callback interface has to provide two methods, one for a successful migration and another one if an error occurs. The first methods also gets a list of the migrated objects as parameter.

External Libraries

A feature often required by clients is the integration of external libraries to use data store specific functions. To help the users with this task, a special feature for injecting third-party libraries can be used to create the instance and to establish the connection to the data store.

```java
public class Main {

        @DatastoreLib(datastore = "couchdb", name = "lightcouch")
        private static CouchDbClient couchDbClient;

        public static void main(String[] args) {
            List<Person> persons = couchDbClient.view("_all_docs")
                        .includeDocs(true)
                        .query(Person.class);

            ...
        }
}
```

Listing 4.9: Using external libraries

An example using an external library is shown in Listing 4.9. As we can see, the `@DatastoreLib` annotation is used to inject instances of the `CouchDbClient` class, which is part of the LightCouch [64] library. Therefore the name of the data store and the third-party library to use need to be provided to the annotation. Injected instances can be used without any limitation and have full access to the data store.

```xml
<?xml version="1.0"?>
<datastores>
    <datastore>
        <name>couchdb</name>
        <host>192.168.56.101</host>
        <port>5984</port>
        <dataunit>cloudscale</dataunit>
        <driver>at.ac.tuwien.infosys.cloudscale.datastore.driver.couchdb.CouchDBDriver</driver>
        <libs>
            <lib>
                <lib-name>lightcouch</lib-name>
                <lib-wrapper>
                    at.ac.tuwien.infosys.cloudscale.datastore.ext.LightCouchWrapper
                </lib-wrapper>
            </lib>
        </libs>
    </datastore>
</datastores>
```

Listing 4.10: Data store configuration for external libraries

To inject instances of a third-party library it is necessary to extend the configuration in the `datastores.xml` file shown in Listing 4.10. In the `libs` section of the data store an unbound number of libraries can be added using the `lib` tag. The `lib-name` provided in the configuration is then used by the `DatastoreLibAspect` to get the entry for the corresponding library using the `lib-wrapper` to create an instance of the required class.

```java
public class LightCouchWrapper implements LibWrapper {

    @Override
    public Object getLib(Datastore datastore) {
        CouchDbProperties properties = new CouchDbProperties();
        properties.setDbName(datastore.getDataUnit());
        properties.setHost(datastore.getHost());
        properties.setPort(datastore.getPort());
        properties.setCreateDbIfNotExist(false);
        properties.setProtocol(DatastoreProperties.DEFAULT_PROTOCOL_TYPE.toString());
        return new CouchDbClient(properties);
    }
}
```

Listing 4.11: LightCouch wrapper example

To add support for an external library, it is necessary to create a wrapper class used to instantiate the required class. Listing 4.11 shows the implementation of the `LightCouchWrapper` used in the example configuration in Listing 4.10. Wrappers are simple classes implementing the `LibWrapper` interface containing a single method. The method to implement gets an instance of the data store and needs to return an object of the library to use.

4.3 Implementation

The library explained in this chapter was implemented using the Java programming language version 7 (Java SE7). To avoid reinventing the wheel and to keep the implementation effort low, third-party libraries where used. A list of the used libraries, together with the version number, is shown in Table 4.4.

Library	**Version**
Gson	2.2.2
Apache Commons Lang	3.1
Apache Commons IO	2.4
Apache Commons Beanutils	1.8.3
Apache Commons Codec	1.7
Apache HTTP Client	4.2.2
HBase	0.94.5
Hibernate Core	3.6.10

Table 4.4: Third-party libraries

The most important library for the current implementation is Gson which is used to represent JSON structures. Actually the type adapters used to transform a specific data type use Gson objects to build the corresponding JSON string. For handling REST based data stores the Apache HTTP Client is used to send requests and to handle the response. For the native HBase driver the official HBase library is used to establish the connection and execute the commands. The Apache Commons libs are used to convert data types and to make parameter handling easier. Hibernate integration requires the Hibernate Core library to implement the event listeners and the corresponding transaction interceptors.

Implementation Issues

When implementing the prototype of the described library some unexpected issues, especially in the mapping component, occurred. The problem is, that mapping Java data types to HBase or JSON and vice versa requires detailed information of the current type. Generally this is no problem because Java provides the required information at runtime making it possible to dynamically load the corresponding adapters. Unfortunately this does not hold for generic types because of the type erasure feature [79] introduced in Java 1.5 to support genericity. In this case, the compiler deletes the explicit type information making it impossible for developers to get the generic type information during runtime.

For simple generic types, this problem can be solved collecting the required type information while iterating over the fields of a class. So the object navigator gets the type informations and adds them to the `TypeMetadata` instance passed to the adapters. Therefore reflection, using the code snippet shown in Listing 4.12, is used.

```java
ParameterizedType parameterizedType = (ParameterizedType) field.getGenericType();
typeParameterTypes = convertTypeArguments(parameterizedType.getActualTypeArguments());
```

Listing 4.12: Java Generics

The collected information is then used to determine the adapter for transforming the field. If the type parameter is user defined, a special adapter is used to call the `ObjectNavigator` to convert this objects.

Another problem occurring during mapping is the distinction between user defined data types and Java internal types. This is necessary because objects created by the user need to be handled recursive to completely map them.

```java
public class Person {
    @DatastoreId(strategy = IdStrategy.AUTO)
    private String id;
    private String firstName;
    private String lastName;

    public enum Gender {
        MALE, FEMALE
    }
    @Adapter(GenderTypeAdapter.class)
    private Gender gender;

    private Address address;
}

public class Address {
    private String street;
    private String city;
    private String country,
    private String zipcode;
}
```

Listing 4.13: Data Store Example with additional field

Listing 4.13 shows an extended version of the example from Listing 4.1 with an additional field address. As we can see, the address is a user defined type. This means, that mapping a `Person` object also requires mapping the corresponding `Address` object. Structures of this form can be nested multiple times requiring a recursive mapping function to completely transform them. Therefore it is sufficient to call the object navigator again for each user defined type adding the result to the current mapping. This means, that it is necessary to distinguish between Java internal types, like `String`, and custom ones like `Address`. Unfortunately Java currently does not provide a function to determine the type membership.

```java
public static boolean isJavaInternalType(Class<?> clazz) {
    ClassLoader classLoader = clazz.getClassLoader();
    if(classLoader == null || classLoader.getParent() == null) {
        return true;
    }
    return false;
}
```

Listing 4.14: Type membership detection

The current mechanism used for this task is shown in Listing 4.14. Generally internal types are loaded by the bootstrap class loader resulting in a null value when requesting it from the class. This means, that classes returning null in the first line of the method where loaded during bootstrap and therefore belong to the Java internal types.

CHAPTER 5

Evaluation

In this section we will evaluate the framework described in the previous section and analyse how it can help developers to manage their data, using different data stores. Therefore, in the first part of the chapter, a scenario will be described, used to perform the analysis of the prototype. After analysing the example, the library will be compared to other state of the art technologies, and the native implementations provided by the data store vendors.

5.1 Evaluation Scenario

The main goal of the framework is to support polyglot persistence and to reduce development effort when using different data stores to manage data. Therefore, we will use an example requiring different types of data to evaluate whether the provided prototype is suitable to fulfil the requirements. In our case, we will use a small web store to test the library and to analyse different functionalities.

The basic use cases of the web store are shown in Figure 5.1. There are basically two different types of actors, customers and administrators. A customer can list the products available in the store and display the details of a specific product if he is interested. Buying an article results in adding it to the order list of the user. The list of actual orders can be examined by the customer at each point of time, also providing the possibility to finish shopping and paying the products. To check their personal information, users can also show their profile listing the persisted data. Administrators are basically customers with additional functions to manage products and users. Therefore, they can create new products or edit existing ones, and create user profiles for new customers or administrators. Advanced features, such as rating different products or adding comments, will be omitted to keep the example simple and to reduce development effort.

From the described uses cases we know that there are in general 4 different types of data listed in Table 5.1. The shown types basically have a fixed number of fields, except products which

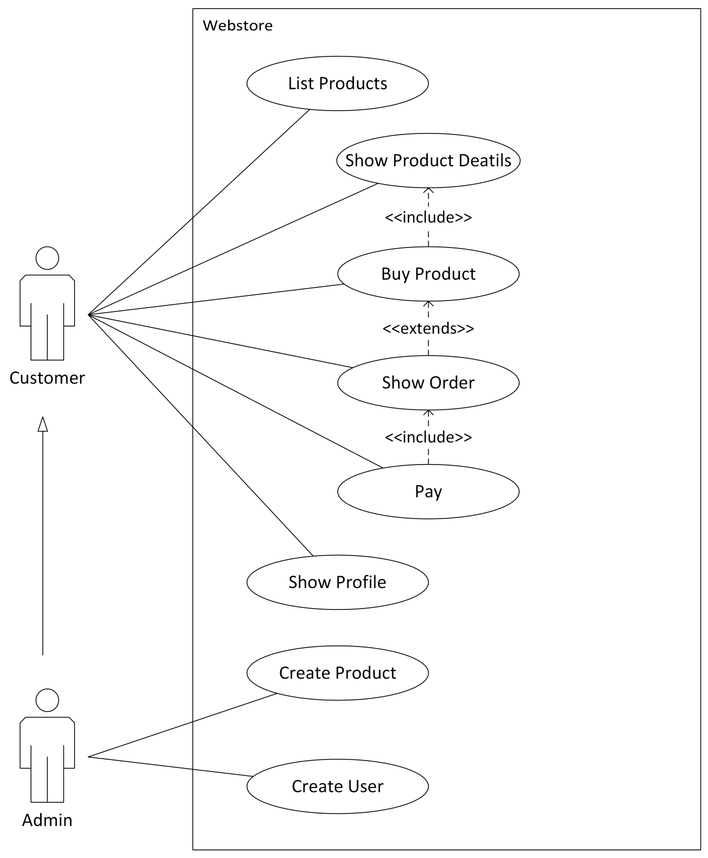


Figure 5.1: Web Store Use Cases

Type	**Number of Fields**	**Access**
User Data	fixed	fast
Product Data	variable	fast
Order Data	fixed	very fast + frequently
Payment Data	fixed	rarely

Table 5.1: Web Store Data Types

can have various properties used for description. Access to the persisted information should generally be fast to avoid slow responses to user requests. Information related to orders has a short lifetime and will be accessed and updated very frequently, in contrast to payment data which will be accessed rarely.

5.2 Analysis

Using the functional and non-functional requirements from the evaluation scenario, we can now analyse the implementation of the example. Therefore, we will first describe the architecture used and then discuss the data model and management.

Architecture

The architecture of the web store is designed to provide a clear separation of concerns [60] and to make traceability of data flows as easy as possible.

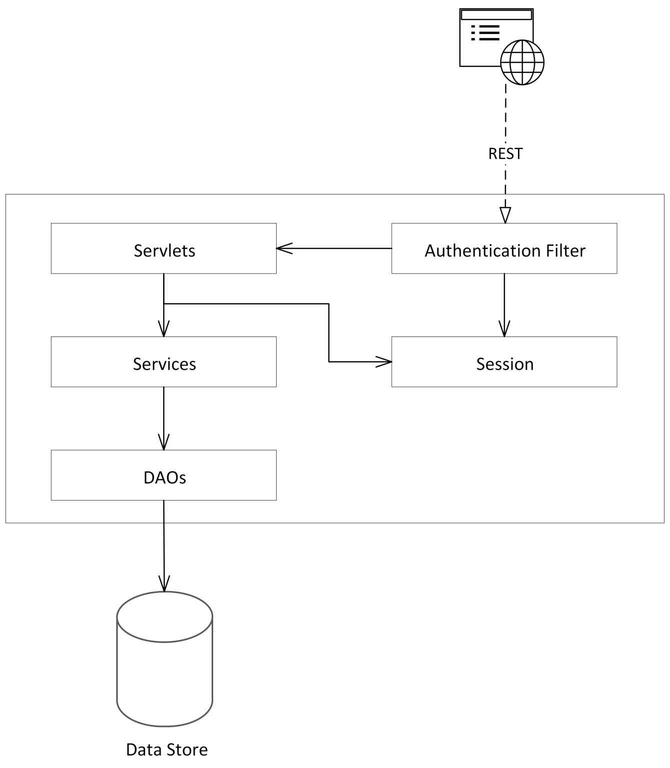


Figure 5.2: Web Store Architecture Overview

The overall architecture of the web store is shown in Figure 5.2. As we can see, clients trigger actions or request information using REST calls. Each incoming request is filter by the so called `AuthenticationFilter`, checking if the user is currently logged in. Therefore, the filter is looking up the `Session` to check if the user is already authenticated or not. If no corresponding entry was found an error response will be returned. Otherwise, the request will be forwarded to the `Servlets`. The servlet handling the request will then unmarshal the payload and call the service, which is responsible for executing the action. To keep track of current orders, the identifier used to find the corresponding order data is stored in the users session. This means, requesting a users current order or adding products to it requires looking up the session to get the identifier. Each service can access the data stores using so called Data Access Object (DAO)s [26].

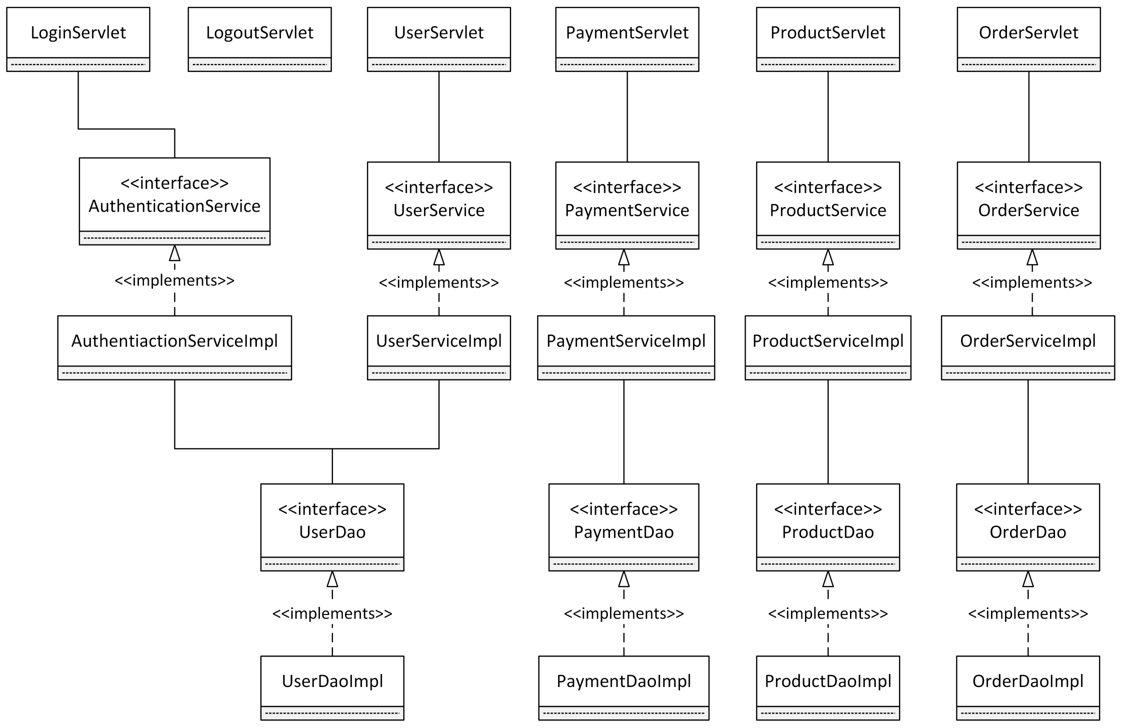


Figure 5.3: Web Store Architecture

A detailed overview of the architecture showing the available servlets, services and DAOs is shown in Figure 5.3. As we can see, there exists exactly one class for each purpose in every layer. Therefore it is possible to easily find the classes responsible for executing an operation and to follow the data flow through the application. An exception is the `LogoutServlet`, used to invalidate a users current session and forcing a login on the next request.

To avoid forwarding internal data to the users, so called Data Transfer Object (DTO)s [28] are used to transfer information between services and the users. Therefore, a special mapping library, the dozer mapper [36], is used to map entities to transport objects and vice versa. This means, that no information used for internal purposes can be forwarded to the caller without explicitly adding it to the transfer object.

Data Model

One of the most interesting parts in the web store example is the data model used to manage the different types of data handled by the application. When defining the data structure, it is necessary to consider the different requirements and to think about the references between the various entries.

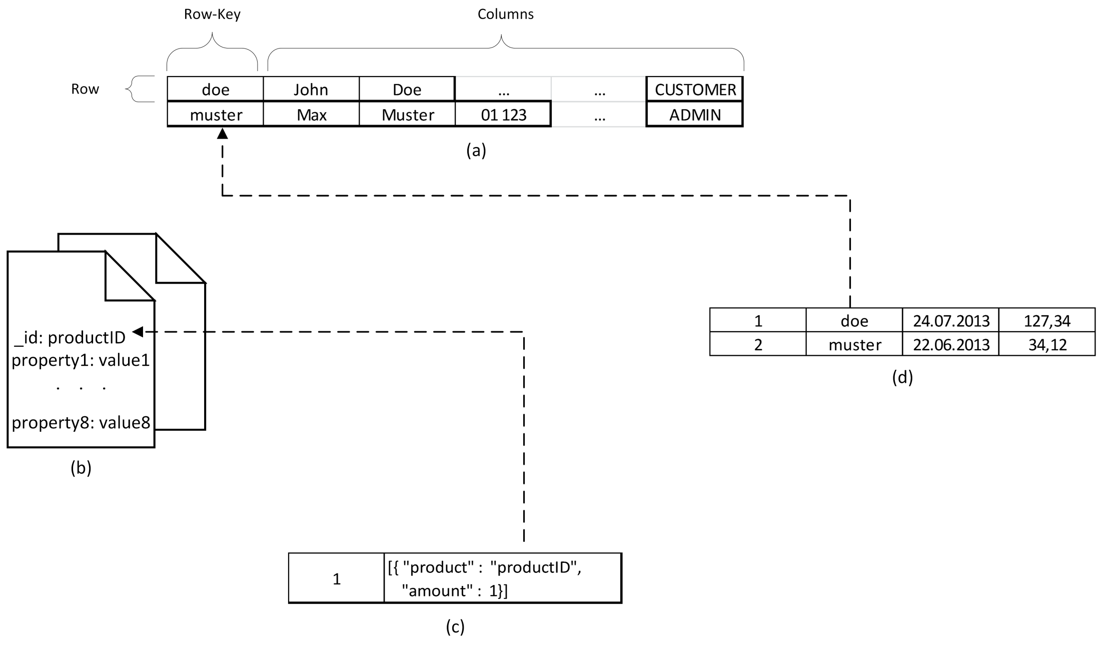


Figure 5.4: Web store data model

The complete data model of the web store is shown in Figure 5.4. As we can see, there is exactly one type of data store for each type of data in the application. To start the implementation we first need to configure the different databases, using the libraries XML configuration file.

The configuration for the web store is shown in Listing 5.1. For each database, managed by the library, there is a separate entry defining the connection properties and the name used as reference in the application. For accessing CouchDB there is also a third-party library, named `lightcouch`, configured. The resulting XML file together with the configuration for hibernate, used for the relational database, is placed in the applications META-INF folder. After finishing the necessary configurations we can now start to implement the entities, used to represent the data persisted in the data stores.

Based on the requirements listed in Table 5.1 we will first create the Java class used for user related information. Since we have fixed number of properties we can simply start to define the related fields in the class. To use the username as identifier, we add the data store id annotation setting the strategy to manual.

The final user entity is shown in Listing 5.2. For mapping the enumeration used to represent the users role, a custom type adapter is used. From the data model we know that user information is

```xml
<?xml version="1.0"?>
<datastores>
    <datastore>
        <name>couchdb</name>
        <host>192.168.56.101</host>
        <port>5984</port>
        <dataunit>cloudscale</dataunit>
        <driver>at.ac.tuwien.infosys.cloudscale.datastore.driver.couchdb.CouchDBDriver</driver>
        <libs>
            <lib>
                <lib-name>lightcouch</lib-name>
                <lib-wrapper>
                    at.ac.tuwien.infosys.cloudscale.datastore.ext.LightCouchWrapper
                </lib-wrapper>
            </lib>
        </libs>
    </datastore>

    <datastore>
        <name>riak</name>
        <host>192.168.56.101</host>
        <port>8098</port>
        <dataunit>test</dataunit>
        <driver>at.ac.tuwien.infosys.cloudscale.datastore.driver.riak.RiakDriver</driver>
    </datastore>

    <datastore>
        <name>hbase</name>
        <host>192.168.56.101</host>
        <driver>at.ac.tuwien.infosys.cloudscale.datastore.driver.hbase.HbaseDriver</driver>
    </datastore>
</datastores>
```

Listing 5.1: Webstore Configuration

persisted in a column oriented database, HBase in our case. This means, that it is only necessary to implement a type adapter for HBase in this case.

The code of the role type adapter, used in the user class, is shown in Listing 5.3. As we can see, the enum is serialized using the string representation of the value and deserialized using the `valueOf` method. In Figure 5.4(a) we can see the final data store representation of the user object. Using a column oriented database in this case avoids persisting null values and provides a scalable solution for managing user related information.

The next kind of data to consider is the product information used to manage the articles in the web store. From Table 5.1 we know that different products can have various fields, representing the properties. To realize this data model we will use a list to persist the specific properties, while simple fields will be used to represent the common properties. The identifier for this entity will be auto generated to keep the development effort low.

The implementation of the product entity is shown in Listing 5.4. We can see, that the only requirement is to add an identifier used to manage the object in the data store. By providing a

```java
public class User {

    @DatastoreId(strategy = IdStrategy.MANUAL)
    private String username;

    private String password;
    private String firstName;
    private String lastName;
    private String address;
    private String city;
    private String email;
    private String phone;

    @Adapters({
        @Adapter(targetClass = HbaseCell.class, adapterClass = RoleTypeAdapter.class)
    })
    private Role role;

    //Getter and Setter
}
```

Listing 5.2: User Entity Example

```java
public class RoleTypeAdapter implements TypeAdapter<Role, HbaseCell> {

    @Override
    public HbaseCell serialize(Role object, TypeMetadata<HbaseCell> typeMetadata) {
        String className = typeMetadata.getParent().getClass().getSimpleName();
        return new HbaseCell(className, typeMetadata.getFieldName(),
                             Bytes.toBytes(object.toString()));
    }

    @Override
    public Role deserialize(HbaseCell element, TypeMetadata<HbaseCell> typeMetadata) {
        byte[] value = element.getValue();
        String name = Bytes.toString(value);
        return Role.valueOf(name);
    }
}
```

Listing 5.3: Role Type Adapter for HBase

list of properties it is possible to persist different articles without any changes. As already shown in Figure 5.4(b), a document oriented database will be used to manage the products of the web store.

A special case is the order data shown in 5.4(c) using a key-value data store, Riak in our example. The order id is used to identify the corresponding entry in the database and is persisted in the users active session. In case of the order items it is necessary to persist a reference to the corresponding entry in the product data store, indicated by the dotted arrow. The problem is that the current implementation of the library does not provide a utility for defining references between objects, handled by different databases.

```java
public class Product {
    @DatastoreId
    private String _id;
    private String name;
    private Double price;
    private List<Property> properties;

    //Getter and Setter
}

public class Property {

    private String name;
    private String value;

    //Getter and Setter
}
```

Listing 5.4: Product Entity Example

```java
public class JsonProductReferenceTypeAdapter implements TypeAdapter<Product,
    JsonElement> {

    @DataSource(name = "couchdb")
    private Datastore datastore;

    @Override
    public JsonElement serialize(Product object, TypeMetadata<JsonElement> typeMetadata
        ) {
        String productID = object.get_id();
        return new JsonPrimitive(productID);
    }

    @Override
    public Product deserialize(JsonElement element, TypeMetadata<JsonElement>
        typeMetadata) {
        JsonPrimitive jsonPrimitive = (JsonPrimitive) element;
        if(!jsonPrimitive.isString()) {
            throw new DatastoreException("Invalid␣value␣for␣product␣reference.");
        }
        String productID = jsonPrimitive.getAsString();
        return datastore.find(Product.class, productID);
    }
}
```

Listing 5.5: Product Reference Type Adapter

Another solution for this use case is to define a custom type adapter creating and resolving the references, shown in Listing 5.5. So when serializing an object the adapter simply returns a JSON element holding the id of the corresponding product. When deserializing, the identifier is used to load the article from the product data store and resolving the reference. Because products can only be ordered when they are persisted, and they can not be deleted, there is no possibility of not finding the referenced product using the type adapter.

The payment data shown in Figure 5.4(d) is persisted in a relational database, using an automatic generated primary key. The dotted arrow signals the reference form the user name, used in the payment table, to the corresponding entry in the data store. Because entities saved in relational databases are managed by the JPA entity manager, we can not define a custom type adapter to resolve this reference automatically. When using hibernate, the type conversion feature [52] can be used to define the custom mapping and to handle the reference automatically. To keep the example simple we will manage this reference manually, setting the corresponding values in the service layer.

Data Management

The different data stores used in the web store example are generally accessed using two different ways, the data store library and native libraries. In the most cases the library developed in this thesis is used, except loading the available products. This operation requires executing a CouchDB view which is not supported in the data store library. The reason is that executing a view is a specific feature not provided by all databases. Using the frameworks advanced mechanism to inject external libraries, we can easily solve this problem by using a native library in this case. All data store operations are encapsulated in the corresponding DAOs, offering a public API to abstract the details. This means, that the layers above do not require to know where the data is persisted when accessing it.

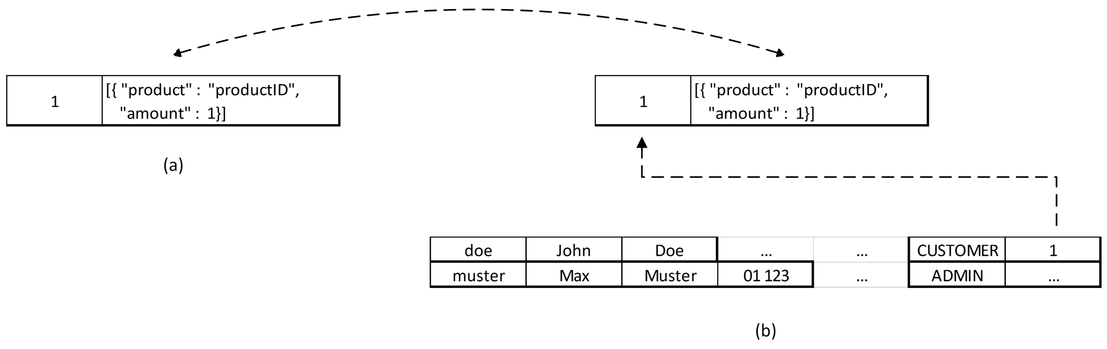

1
[{ "product" : "productID", "amount" : 1}]
(a)
1
[{ "product" : "productID", "amount" : 1}]
doe
John
Doe
...
...
CUSTOMER
1
muster
Max
Muster
01 123
...
ADMIN
...
(b)

Figure 5.5: Web store order migration

To demonstrate a practical example of the data store library's migration feature, we can add an additional use case allowing users to save their orders and to continue later. We assume that order data persisted in the Riak database will be deleted when the users session is invalidated. This means, it is necessary to save the data to another data store before the corresponding entry is deleted.

Figure 5.5 shows how the migration of the users order data works. In 5.5(a) we can see the data persisted in the Riak data store, which will be deleted when the session is invalidated. Therefore the users entry will be migrated to another database, HBase in our case, indicated by the dotted arrow to 5.5(b). No additional code is necessary to perform the data migration, except adding the users order id to his personal information. This is necessary to assign the order data to the user, which is normally done by the session. When the next log in occurs the migrated data will be transferred back to the Riak data store, restoring the users order list and removing the order id from the personal information adding it to the current session.

5.3 Comparison to Other Libraries

In this section, we will compare the data store library with other state of the art technologies. Therefore, we will first compare the features provided by the different libraries followed by a comparison of the performance.

Functionality

In the first part, the features and functionalities provided by the different libraries will be compared providing an overview of the current development. Actually, there do not exist many frameworks supporting multiple NoSQL data stores and most of them are still under heavy development. Projects like Hibernate OGM [53] try to add support for non relational databases to ORM tools used for traditional RDBMS. The HBase plug-in [30] for the JDO framework Datanucleus [29] also belongs to this category, reusing existing frameworks to support new types of data stores. For our comparison we will use the libraries provided by the different data store vendors and Datanucleus because other frameworks only support databases not handled by the library developed in this thesis.

Feature	Data Store Lib	LightCouch	Riak Library	HBase	Datanucleus
Mapping	✓	✓	✓	✗	✓
Custom Types	✓	✓	✓	✗	✓
Associations	✗	✗	✗	✗	✓
Queries	✗	limited	✗	limited	✓
Extendible	✓	✗	✗	✗	✓
Migration	✓	✗	✗	✗	✗
External Libraries	✓	✗	✗	✗	✗
Hibernate Support	✓	✗	✗	✗	✗

Table 5.2: Supported Features

An overview of the different features and the support in the various libraries is shown in Table 5.2. A cell containing a "✓" signals that the feature is supported by the corresponding library while a "✗" means the converse.

As we can see, nearly each library, except HBase, supports mapping of Java objects to a corresponding representation in the data store. For conversion to column oriented schemas there actually does not exist any library, while for JSON there are different third-party libraries available. So frameworks supporting data stores using JSON often use external tools to map objects, as this is also the case in the data store library developed in this thesis.

Supporting custom types to get a flexible mapping architecture is another important feature, provided by nearly all of the listed frameworks. As HBase actually does not support automatic object conversion new types need to be mapped manually using the libraries helper functions. In all other cases it is only necessary to implement an interface or extend an object to add support for a new data type.

A feature known from ORM tools, used for relational databases, is the mapping of associations between different objects. Special annotations are used to define the representation of an association between two or more objects in the data store. Datanucleus, which can also be used for relational databases, also supports this mechanism using modern NoSQL data stores. In all of the other libraries this problem can be solved using a custom converter to map an association requiring additional development effort.

Executing queries to select a specific set of data is a functionality often required by different applications. Basically, non relational databases do not provide a query engine with a special language to select data sets. Some vendors offer additional features, like map reduce or other technologies, to perform operations returning entries with specific properties. CouchDB, for example, has a special feature called views to filter data based on some criteria. The LightCouch library provides an API to dynamically create views or executed persisted ones, but ,actually, views can not fully replace queries and the API is limited. HBase also supports filtering data using a scanner API to select the specific entries. Also this technology can not replace traditional query languages at the moment. In Datanucleus, on the other hand, it is possible to select data sets using JDO Query Language (JDOQL). The only thing to consider when using this framework is that some of the queries are executed in the client's memory, because not all operations are supported by the data store. This can lead to bad performance caused by transferring large sets of data to the client.

Another important factor when comparing different libraries is the possibility to add support for new types of data stores. The different frameworks provided by the database vendors typically lack this functionality. As we can see in Table 5.2, only the data store library from this thesis and Datanucleus offer methods to add support for new types of databases. In case of the data store framework, it is only necessary to implement an interface, while Datanucleus uses OSGi [5] for adding new extensions.

Data migration between different data stores is actually not supported by any of the third-party libraries. When moving data between databases of the same type, all libraries can be used because no mapping to a different data model is necessary. The advantage of the migration feature is that entries can be transferred between all data stores supported by the library without any limitations.

Supporting external libraries is also a feature only implemented by the data store framework developed in this thesis. The benefit of this functionality is that different libraries can be used while keeping the configuration of the data stores in a central place. This means, that for example, changing the data unit or the port of a database only requires to change the corresponding entry in the data store frameworks XML file. External tools can be injected and used easily when special functionality is required.

Hibernate integration is another feature helping developers to reference data in data stores from their hibernate entities using annotations. Actually this technology is not supported by any of the external frameworks listed in the table. This feature should not be confused with associations where users can define the cardinality of the relation. In case of the hibernate support, each instance of an entity persisted will create a new data store entry in the referenced database. This means, that it is currently not possible, that two instances of an entity point to the same data store entry.

Performance

After comparing the features of the different libraries, we will now measure the performance of the frameworks listed in Table 5.2. All measurements were carried out on a MacBook Air, with a 2 GHz Intel Core i7 running OS X version 10.8.4. The different databases used for the measurement were hosted in a virtual machine running Ubuntu 12.10. The test machine was using the standard Oracle Java Runtime Environment (JRE) version 7, and all code required was compiled using `javac` version 1.7.0. During the test execution no other programs where executed to influence the results.

To get a good overview of the performance, three different types of objects where used for the measurement:

- A simple object containing only fields of Java internal types like String and Integer.
- An object containing a generic field, an array list in our case.
- An object having a field of a user defined type.

For all of the listed objects the basic CRUD operations where executed measuring the required time, using the `StopWatch` [10] object from the Apache Commons library. Each task was executed 100 times taking the average value to avoid fluctuations in the result.
As we can see in Figure 5.6, the native libraries are much faster performing the create operation for all of test objects. The reason for this circumstance is that these frameworks only focus on one specific type of data store while the library developed in this thesis and Datanucleus try to provide a more general solution. Therefore, both nearly have the same performance, except the object containing a user defined type. The problem in this case is that the default behaviour of Datanucleus is to use one table per class. This means, that in the last example there are two tables created, one for the object and another on for the sub-object, requiring two write

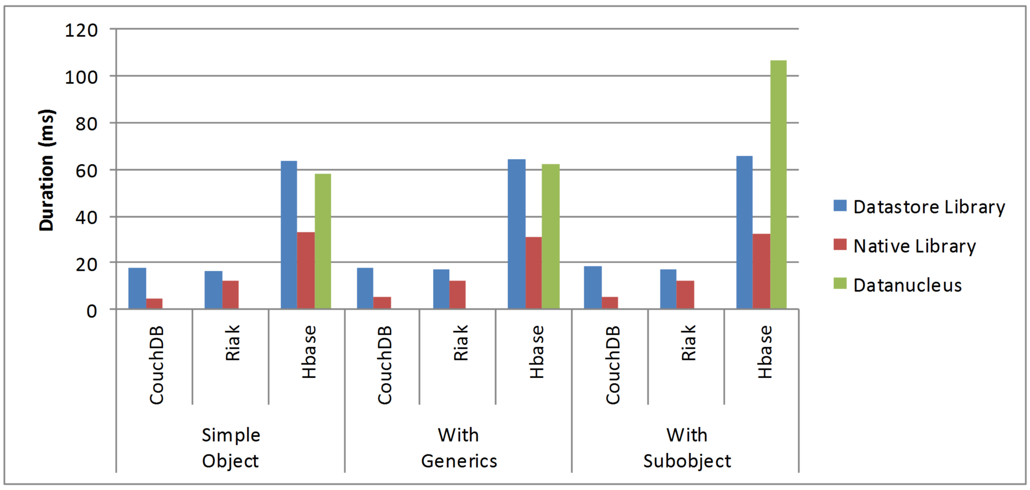


Figure 5.6: Performance Create Operation

operations. When comparing the HBase native library we also have to consider that in this case a manual mapping is necessary, because no automatic object conversion is provided. Another interesting observation is that the data store library nearly has the same performance for all of the provided test objects.

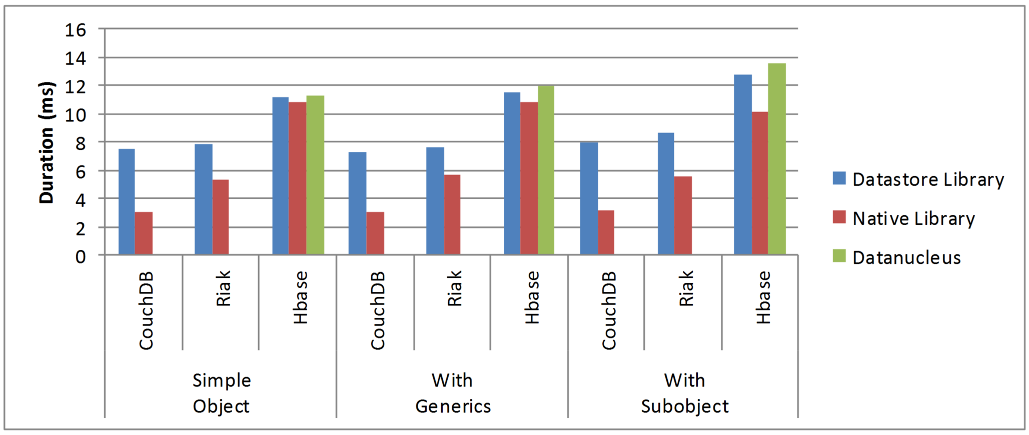


Figure 5.7: Performance Read Operation

In Figure 5.7, the read performance of the libraries is shown. As the create operation before, the native solutions provide a lower duration reading the different objects from the database. In this case, the structure of the persisted data does not play an important role leading to nearly identical results for all test objects. One thing to consider is that the performance of Datanucleus could be improved using a caching method, explained in the official documentation [31].

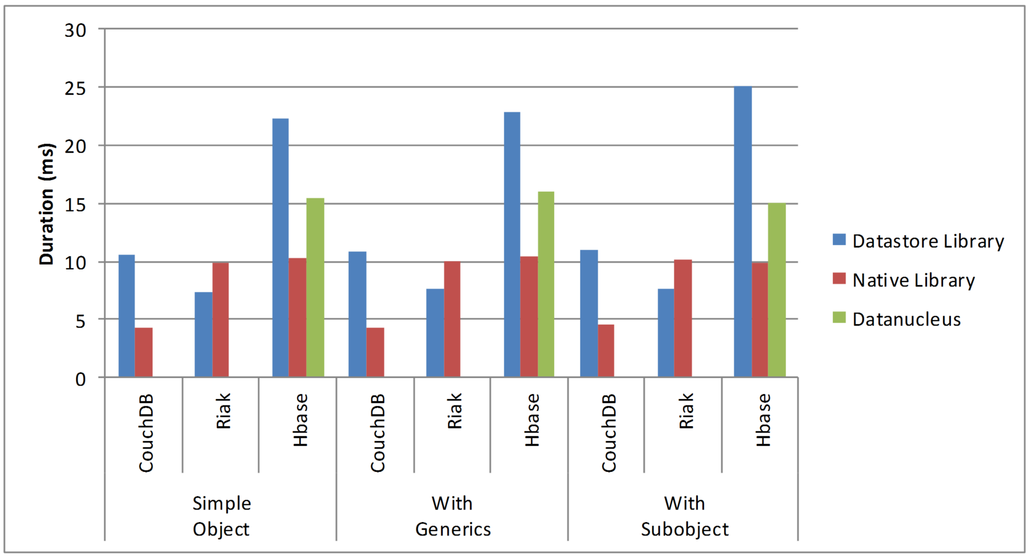


Figure 5.8: Performance Update Operation

The performance comparison of the update operations is shown in Figure 5.8. As we can see, in case of the HBase data store the library of this thesis has the longest duration updating the different test objects. This is the case because performing this type of operation results in creating a special update procedure detecting the correct task for each column in the database. The results of the performance measurements have shown, that this task can be very time consuming and should therefore be refactored in future releases. One possibility to solve this problem is to compare the object to update with the currently persisted one, to get the set of fields which have changed. Reloading the saved object is already necessary when using Datanucleus to add it to the current persistence context.

Figure 5.9 shows the results of the performance measurements for the delete operations. As we can see, the native libraries are again faster than the more generic solutions, except the HBase framework. This is the reason because delete operations are faster using a bulk operation and Hadoop Distributed File System (HDFS) [49], which should be used in production, instead of performing single requests and using common file systems [35]. The lower performance of Datanucleus is resulting from reloading the object from the database, before deleting it, and the fact that data needs to be removed from two different tables.

After measuring the performance of the different data store libraries using basic CRUD operations, we will know execute a more complex scenario. Therefore we will use parts of the data model known from the web store example, discussed earlier in this chapter. The use case for this measurement includes the following steps:

- Create 100 objects each having 100 different properties

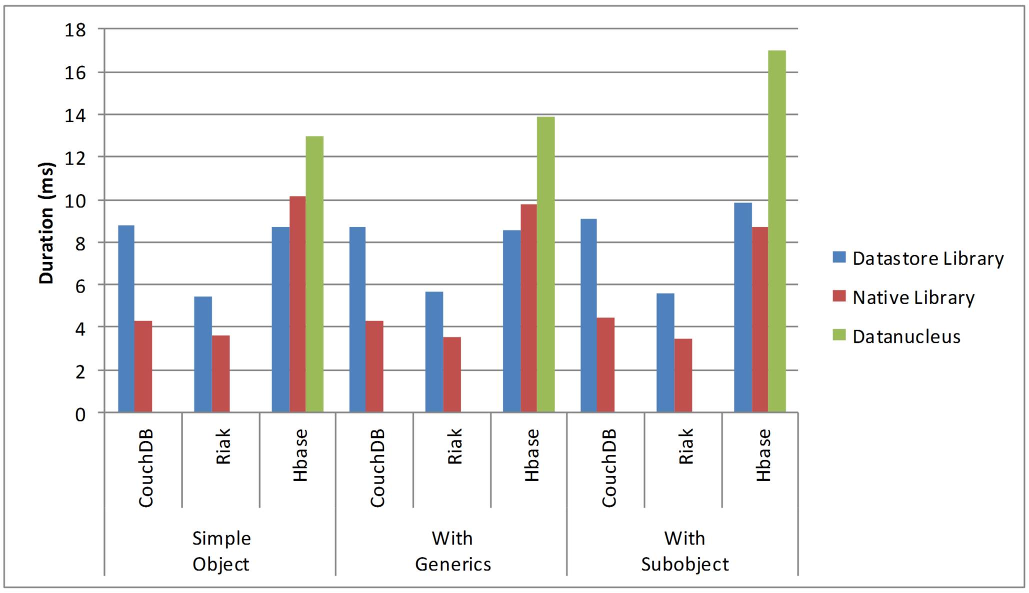


Figure 5.9: Performance Delete Operation

- Create 100 orders consisting of 100 articles
- View the persisted orders
- Update one item in each order
- Delete all orders
- Delete all products

To compare the different frameworks, we will use the duration required for performing all operations listed above.

Figure 5.10 shows the results of the performance measurement for the scenario. In this example, the native libraries are still faster but the difference is smaller than in the examples before. One thing to consider is, that the vendor frameworks do not provide the possibility to easily control the mapping by using annotations. Especially in the case of HBase the development effort is higher using the included Java classes. The overhead resulting from the mapping strategy is getting smaller when using large sets of data. In contrast to Datanucleus, the data store library is still faster handling the given scenario.

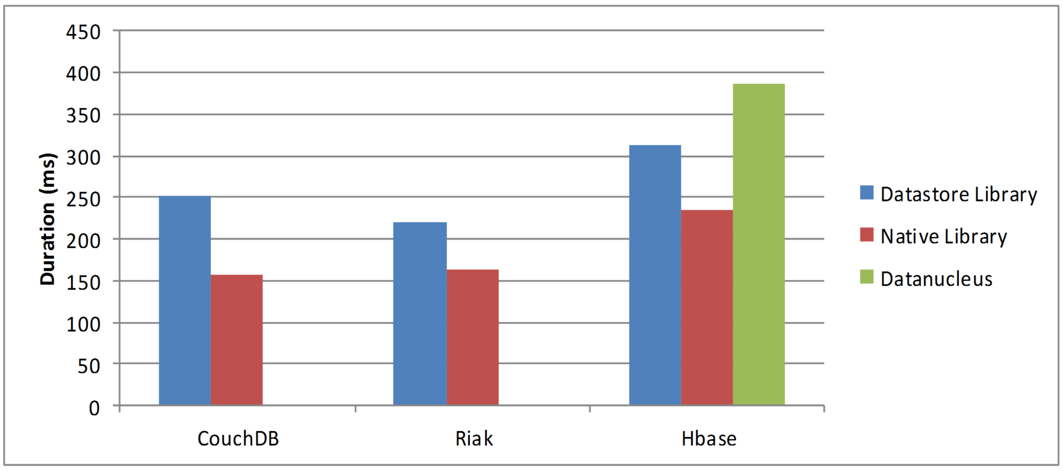


Figure 5.10: Performance Scenario

5.4 Summary

After analysing the evaluation scenario used for the framework developed in this thesis, we will now use the results to answer the questions from Section 1.2.

- Is it possible to create a simple interface for accessing different data store types ? Is it also possible to get access to specific libraries to use special functionality provided by the database ?

  As we have seen in the web store example, the current implementation offers a public API which can be used to access different database types. All of the provided functions are supported by each data store type and there is no limitation using these methods to manage data. Using external libraries to access data is also possible by simply injecting an instance of the required class, using an annotation of the framework.

- How can different databases be used together to support polyglot persistence ?

  In the evaluation scenario we have used different types of data stores to implement a simple web store. The example has shown that developed framework perfectly supports polyglot persistence. The databases to use can be configured using the libraries XML configuration mechanism and can then be injected using the provided annotation.

- How can objects be moved from one data store to another without writing a lot of migration code ?

  Migrating a single object or a list of objects can be done by using the migration function provided by the frameworks public API. This means, that no additional code is necessary to move data between different types of data stores. Migrating large sets of objects can be done asynchronously by providing a callback object.

The prototype implemented in this thesis already provides the basic functions required for using different data stores to manage data. All of the provided methods can be used for each type of data store, without any limitations. The current version has built-in support for three different types of databases and new ones can be added easily implementing the required interface. The advantage of the provided solution is that is built modular, making it possible to replace different parts. By injecting external libraries users get the possibility to use vendor specific functions without much effort. This option makes the framework very flexible and can help developers using different libraries to manage there data. Another interesting feature is the hibernate integration used to mark data which should be persisted in a data store. This way the corresponding data store operation is executed automatically when manipulating the entity.

Despite all the advantages, the current solution still has some limitations and disadvantages. One of the main problems is that there is actually no way for mapping associations and relations between different objects. This means, that all references need to be managed manually leading to a higher development effort. As objects often have relations this is an important feature which is currently missing. Another functionality, often required by applications, is the possibility to filter data. To solve this problem it is necessary to find a solution which can be used for all data store types and to provide a general filtering API. One feature also known from ORM tools is the possibility to control the mapping by using annotations. The current implementation does for example not provide the possibility to simply control the field names used in the data store representation. However, this problem can actually be solved by using custom mappers to control the output. Another problem shown in the performance measurements is, that the current implementation is sometimes much slower than the native libraries. One reason therefore is the actual mapper used to transform objects and data store results. Compared to other libraries supporting multiple databases, like Datanucleus, the current implementation nevertheless provides a good performance.

CHAPTER 6

Future Work

The current version of the data store library already provides basic features for managing data in different databases. However, the current development is not considered to be finished and there are still some important features missing. The next function to implement would be the handling of associations between objects. This is necessary, to provide the possibility for representing associations in the different data stores and to persist more complex data structures. Another improvement that would be important for the mapping component is the configuration, based on annotations. Like modern ORM tools, the framework should provide the possibility to control the mapping using annotations. This will help users to customize the representation in the data store and therefore provide a more flexible solution.

Besides providing new features, the performance of the existing functions should be improved. A higher performance is making the library more attractive for users, currently working with native solutions. Therefore, it is necessary to improve the mapping component, especially the field handling and object loading. Currently, a lot of reflection functionality [38] is used to convert objects to the corresponding data store representation and vice versa. JPA and JDO use byte code manipulation to overcome this problem and to limit the number of reflection methods used. This results in limiting the runtime and moving parts of the mapping logic to the compile time. Another possibility to increase performance is the usage of caches, to limit the number of time consuming operations. This means, that for example the object navigation could be extended to use a caching mechanism for accessing frequently used fields.

Furthermore, the hibernate integration and the support for external libraries should be improved to make these features more attractive for developers. The hibernate support for example should support references to data store objects and lazy loading. Adding new libraries can be improved by making the current wrapper implementation more adaptable and support easier integration of external features. Another important task is to add more features to the migration utility, used to migrate data between different data stores. A useful feature in this case would be the possibility to add custom transformation functions, to control the migration process and its result.

Adding support for new data stores is also an important task for feature versions. New data store drivers can help users to use new databases, without increasing development effort. A very useful tool in this case would be a general query language, to filter results and find special data sets.

APPENDIX A

Acronyms

RDBMS	Relational Database Management Systems
NoSQL	not only SQL
BASE	**B**asically **A**vailable, **S**oft state, **E**ventual consistency
ACID	**A**tomicity, **C**onsistency, **I**solation, **D**urability
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
ORM	Object-Relational Mapping
ER	Entity Relationship
JPA	Java Persistence API
JDO	Java Data Objects
ER	Entity-Relationship
REST	Representational State Transfer
SLA	Service-level Agreement
GFS	Google File System
SQL	Structured Query Language
JSON	JavaScript Object Notation

DDL	Data Definition Language
SQL	Structured Query Language
DBMS	Database Management System
API	Application Programming Interface
OCCI	Open Cloud Computing Interface
CDMI	Cloud Data Management Interface
POSIX	Portable Operating System Interface
DISC	Data Intensive Scalable Computing
DBaaS	Database as a Service
TTL	Time To Live
PU	Pending Update
CRUD	Create Read Update Delete
DAO	Data Access Object
DTO	Data Transfer Object
JRE	Java Runtime Environment
HDFS	Hadoop Distributed File System
JDOQL	JDO Query Language

APPENDIX B

Configuration Schema

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <!-- datastores root element -->
    <xs:element name="datastores">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="datastore" minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <!-- datastore element -->
    <xs:element name="datastore">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="host" type="xs:string"/>
                <xs:element name="port" type="xs:positiveInteger" minOccurs="0"/>
                <xs:element name="dataunit" type="xs:string" minOccurs="0"/>
                <xs:element name="driver" type="xs:string"/>
                <xs:element ref="libs" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <!--external libs root element -->
    <xs:element name="libs">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="lib" minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <!-- external lib element -->
    <xs:element name="lib">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="lib-name" type="xs:string"/>
                <xs:element name="lib-wrapper" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Listing B.1: Datastore Configuration Schema Definition

APPENDIX C

Project Setup

Setting up a project from scratch to use the data store library is very simple and will be explained in the following sections.

C.1 Add Maven Dependency

To start using the library, we first need to add the required dependency to maven. Therefore add the following lines to the pom.xml file of the project:

```xml
<dependency>
      <groupId>cloudscale</groupId>
      <artifactId>cloudscale.core</artifactId>
      <version>0.1.0-SNAPSHOT</version>
</dependency>
```

Listing C.1: Maven Dependency

C.2 Configuration

To inform the data store library about the available data stores and to configure the different connection properties and drivers, a configuration file needs to be added. Therefore create a file named datastores.xml in the projects META-INF folder.

```xml
<?xml version="1.0"?>
<datastores>
    <datastore>
        <name>riak</name>
        <host>192.168.56.101</host>
        <port>8098</port>
        <dataunit>cloudscale</dataunit>
        <driver>at.ac.tuwien.infosys.cloudscale.datastore.driver.riak.RiakDriver</driver>
    </datastore>
</datastores>
```

Listing C.2: Datastore Configuration

Listing C.2 shows an example of a basic configuration with one data store named riak. A detailed description of all possible configurations can be found in the schema definition shown in Listing B.1.

C.3 Using Data Stores

After adding the required dependencies and the necessary configuration the project is ready to use the data store library. There are basically two ways of getting access to a data store. The first method uses dependency injection with the help of AspectJ while the second one does not require any additional libraries.

To use dependency injection for data store access, it is necessary to configure maven to apply the corresponding aspects. Therefore add the following lines to the projects pom.xml file:

```
<plugin>
     <groupId>org.codehaus.mojo</groupId>
     <artifactId>aspectj-maven-plugin</artifactId>
     <version>1.4</version>
     <configuration>
          <source>1.7</source>
          <target>1.7</target>
          <complianceLevel>1.7</complianceLevel>
          <verbose>true</verbose>
     </configuration>
     <executions>
          <execution>
               <configuration>
                    <XnoInline>true</XnoInline>
                    <aspectLibraries>
                         <aspectLibrary>
                              <groupId>cloudscale</groupId>
                              <artifactId>cloudscale.core</artifactId>
                         </aspectLibrary>
                    </aspectLibraries>
               </configuration>
               <goals>
                    <goal>compile</goal>
                    <goal>test-compile</goal>
               </goals>
          </execution>
     </executions>
     <dependencies>
          <dependency>
               <groupId>org.aspectj</groupId>
               <artifactId>aspectjrt</artifactId>
               <version>1.7.0</version>
          </dependency>
          <dependency>
               <groupId>org.aspectj</groupId>
               <artifactId>aspectjtools</artifactId>
               <version>1.7.0</version>
          </dependency>
     </dependencies>
</plugin>
```

Listing C.3: Applying Aspects

Now the data stores configured in the datastores.xml file can be accessed using the `@DataSource` annotation.

```java
public class Main {

    @DataSource(name = "riak")
    private static Datastore riak;

    ...
}
```

Listing C.4: Using Dependency Injection

When not using injection configured data stores can be accessed using the `DatastoreFactory`. The returned object can be used exactly as with injection.

```java
public class Main {

    public static void main(String[] args) throws InterruptedException {

        Datastore riak= DatastoreFactory.getInstance().getDatastoreByName("riak");

        ...
    }
}
```

Listing C.5: Using Factory

C.4 Adding Data Store Objects

To manage an object in a data store it is necessary to add a string field annotated with `@DatastoreId`. The data store library uses this field to store the id assigned to the corresponding object in the data store.

```java
public class MyComplexResult {

    @DatastoreId
    private String id;

     ...
}
```

Listing C.6: Data Store Object

Now instances of `MyComplexResult` can be saved, updated, searched or deleted in a data store using the library.

Bibliography

[1] Daniel J Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull*, 32(1):3–12, 2009.

[2] Ado.net entity framework. `http://msdn.microsoft.com/en-US/data/ef`. Accessed: 11/01/2013.

[3] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ado.net entity framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 877–888, New York, NY, USA, 2007. ACM.

[4] Andre Eickler Alfons Kemper. *Datenbanksysteme*. Oldenbourg, 6 edition, 2006.

[5] Osgi Alliance. *Osgi Service Platform, Release 3*. IOS Press, Inc., 2003.

[6] Amazon dynamodb. `http://aws.amazon.com/dynamodb/`. Accessed: 11/01/2013.

[7] Amazon elastic compute cloud (ec2) documentation. `http://aws.amazon.com/documentation/ec2/`. Accessed: 11/01/2013.

[8] Amazon simple storage service (s3) documentation. `http://aws.amazon.com/documentation/s3/`. Accessed: 11/01/2013.

[9] Rajagopal Ananthanarayanan and Karan Gupta. Cloud analytics: Do we really need to reinvent the storage stack. *... on Hot Topics in Cloud ...*, pages 1–5, 2009.

[10] Apache commons stopwatch. `http://commons.apache.org/proper/commons-lang/javadocs/api-2.6/org/apache/commons/lang/time/StopWatch.html`. Accessed: 24/07/2013.

[11] Apache couchdb. `http://couchdb.apache.org/`. Accessed: 11/01/2013.

[12] Apache hbase. `http://hbase.apache.org/`. Accessed: 11/06/2013.

[13] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[14] A. Bedra. Getting started with google app engine and clojure. *Internet Computing, IEEE*, 14(4):85 –88, july-aug. 2010.

[15] Catriel Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5(4):353 – 382, 1990.

[16] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1–12, New York, NY, USA, 2007. ACM.

[17] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[18] L. Bonnet, A. Laurent, M. Sala, B. Laurent, and N. Sicard. Reduce, you say: What nosql can do for data aggregation and bi in large repositories. In *Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on*, pages 483 –488, 29 2011-sept. 2 2011.

[19] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 251–264, New York, NY, USA, 2008. ACM.

[20] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

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

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[23] Peter Pin-Shan Chen. The entity-relationship model toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[24] Cloud data management interface (cdmi) v1.0.2. `http://snia.org/sites/default/files/CDMI%20v1.0.2.pdf`. Accessed: 11/01/2013.

[25] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, January 1983.

[26] Core j2ee patterns - data access object. `http://www.oracle.com/technetwork/java/dataaccessobject-138824.html`. Accessed: 24/07/2013.

[27] Carlo Curino, Evan PC Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: A database-as-a-service for the cloud. 2011.

[28] Data transfer object. `http://martinfowler.com/eaaCatalog/dataTransferObject.html`. Accessed: 24/07/2013.

[29] Datanucleus. `http://www.datanucleus.org/`. Accessed: 24/07/2013.

[30] Datanucleus hbase support. `http://www.datanucleus.org/products/datanucleus/datastores/hbase.html`. Accessed: 24/07/2013.

[31] Datanucleus jdo caching. `http://www.datanucleus.org/products/datanucleus/jdo/cache.html`. Accessed: 24/07/2013.

[32] Umerhwar Dayal and Four Cambridge Center. Processing queries over generalization hierarchies in a llultidatabare system. 1983.

[33] db4objects object oriented database. `http://www.db4o.com/`. Accessed: 11/01/2013.

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[35] Deleting from hbase. `http://hbase.apache.org/book.html#perf.deleting`. Accessed: 24/07/2013.

[36] Dozer mapper. `http://dozer.sourceforge.net/`. Accessed: 24/07/2013.

[37] Jim R. Wilson Eric Redmond. *Seven Databases in Seven Weeks*. The Pragmatic Bookshelf, 2012.

[38] Ira R. Forman, Nate Forman, Dr. John Vlissides Ibm, Ira R. Forman, and Nate Forman. Java reflection in action, 2004.

[39] M. Fowler. Polyglot persistence. `http://martinfowler.com/bliki/PolyglotPersistence.html`. Accessed: 11/01/2013.

[40] Simson L. Garfinkel and Simson L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3, and sqs. Technical report, Center for, 2007.

[41] Lars George. *HBase: the definitive guide*. O'Reilly Media, Incorporated, 2011.

[42] Oliveira Rui Carlos Mendes de Gomes Pedro, Pereira JosÈ. An object mapping for the cassandra distributed database. 2011.

[43] Google app engine. `https://developers.google.com/appengine/`. Accessed: 11/01/2013.

[44] Google docs. `docs.google.com`. Accessed: 11/01/2013.

[45] Google gson. `https://code.google.com/p/google-gson/`. Accessed: 11/06/2013.

[46] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 29 –38, 2002.

[47] Pascal Hauser. Review of db4o from db4objects. *University of Applied Sciences Rapperswil, Switzerland*, 2011.

[48] Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7):9, July 2008.

[49] Hdfs architecture guide. `http://hadoop.apache.org/docs/stable/hdfs_design.html#Introduction`. Accessed: 24/07/2013.

[50] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341, 2011.

[51] Hibernate. `http://www.hibernate.org/`. Accessed: 11/01/2013.

[52] Hibernate custom types. `http://docs.jboss.org/hibernate/orm/3.6/reference/en-US/html/types.html#types-custom`. Accessed: 24/07/2013.

[53] Hibernate ogm (object/grid mapper). `http://www.hibernate.org/subprojects/ogm.html`. Accessed: 24/07/2013.

[54] Introducing json. `http://www.json.org/`. Accessed: 11/06/2013.

[55] Java properties. `http://docs.oracle.com/javase/tutorial/essential/environment/properties.html`. Accessed: 11/06/2013.

[56] Sam Johnston. Diagram showing overview of cloud computing including google, salesforce, amazon, axios systems, microsoft, yahoo and zoho. `http://upload.wikimedia.org/wikipedia/commons/b/b5/Cloud_computing.svg`, 2009. Accessed: 11/01/2013.

[57] C Keene. Data services for next-generation soas. *SOA WebServices Journal*, 4:12, 2004.

[58] Won Kim. Research directions in object-oriented database systems. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS ’90, pages 1–15, New York, NY, USA, 1990. ACM.

[59] Ramnivas Laddad. *AspectJ in action: practical aspect-oriented programming*, volume 512. Manning Greenwich, 2003.

[60] P.A. Laplante. *What Every Engineer Should Know about Software Engineering*. What Every Engineer Should Know. Taylor & Francis, 2007.

[61] N. Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12 –14, feb. 2010.

[62] Neal Leavitt. Is cloud computing really ready for prime time. *Growth*, 27(5):15–20, 2009.

[63] K.J. Lieberherr and I.M. Holland. Assuring good style for object-oriented programs. *Software, IEEE*, 6(5):38–48, 1989.

[64] Lightcouch. `http://www.lightcouch.org/`. Accessed: 06/07/2013.

[65] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[66] Robert C Martin. The visitor family of design patterns. *The Principles, Patterns, and Practices of Agile Software Development*, 2002.

[67] Thijs Metsch, Andy Edmonds, R Nyrén, and A Papaspyrou. Open cloud computing interface–core. In *Open Grid Forum, OCCI-WG, Specification Document. Available at: http://forge. gridforum. org/sf/go/doc16161*, 2010.

[68] Mongodb. `http://www.mongodb.org/`. Accessed: 11/01/2013.

[69] Elizabeth J. O'Neil. Object\ relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1351–1356, New York, NY, USA, 2008. ACM.

[70] Oracle database application developers guide - object-relational features. `http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14260/toc.htm`. Accessed: 11/01/2013.

[71] Oracle toplink. `http://www.oracle.com/technetwork/middleware/toplink/overview/index.html`. Accessed: 11/01/2013.

[72] Swapnil Patil, Garth A Gibson, Gregory R Ganger, Julio Lopez, Milo Polte, Wittawat Tantisiroj, and Lin Xiao. *In search of an API for scalable file systems: Under the table or above it?* Defense Technical Information Center, 2009.

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

[74] Martin Fowler Pramod J. Sadalage. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Longman, 2012.

[75] Niko Schwarz, Mircea Lungu, and Oscar Nierstrasz. Seuss: Decoupling responsibilities from static methods for fine-grained configurability. *Journal of Object Technology*, 11(1):3:1–23, April 2012.

[76] John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W. T. Lin, and Eugene Wong. Multibase: integrating heterogeneous distributed database systems. In *Proceedings of the May 4-7, 1981, national computer conference*, AFIPS '81, pages 487–499, New York, NY, USA, 1981. ACM.

[77] Michael Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11, April 2010.

[78] The hibernate object life-cycle. `http://learningviacode.blogspot.co.at/2012/02/hibernate-object-life-cycle.html`. Accessed: 11/06/2013.

[79] The java language specification. `http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html`. Accessed: 11/06/2013.

[80] The sql-92 standard. `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt`. Accessed: 11/01/2013.

[81] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[82] Toby Velte, Anthony Velte, and Robert Elsenpeter. *Cloud Computing, A Practical Approach*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[83] John Vlissides, R Helm, R Johnson, and E Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49, 1995.

[84] Dean Wampler and Tony Clark. Guest editors' introduction: Multiparadigm programming. *Software, IEEE*, 27(5):20 –24, sept.-oct. 2010.

[85] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38 –49, march 1992.

[86] Florian Wolf, Heiko Betz, Francis Gropengießer, and Kai-Uwe Sattler. Hibernating in the cloud–implementation and evaluation of object-nosql-mapping.

[87] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010. 10.1007/s13174-010-0007-6.