SQL in the Cloud

Master Thesis
Daniel Egger

Supervised by:
Prof. Dr. Donald Kossmann
Tim Kraska

Systems Group
Department of Computer Science
ETH Zürich

March, 2009 – September, 2009
Today’s web applications have the choice between two different and competing data storage options. On the one side there are the traditional relational database management systems, whereas on the other side there is a whole new market of different cloud storage systems emerging. The new cloud storage systems promise to provide very high availability and to be very scalable - characteristics that cannot be fulfilled by traditional RDBMS.

The goal of this thesis is to combine those two areas by using a cloud storage system as the storage layer for the MySQL database. We aim at providing SQL in the cloud. By leveraging the scalability of the cloud storage system, we intend to create a highly scalable database. In order to achieve this, an efficient mapping of the internal SQL structures and indexes to the data model of the cloud storage system had to be established.

The results of the benchmarks have shown that, with some exceptions, these goals have been achieved.
Contents

1 Introduction 7
  1.1 Motivation .................................................. 7
  1.2 Problem Statement ......................................... 7
  1.3 Contribution ............................................... 8
  1.4 Structure of this Thesis .................................. 8

2 Cloud Storage Systems 11
  2.1 Cloud Computing ........................................... 11
  2.2 Cloud Storage Systems ..................................... 12
  2.3 Cassandra .................................................. 13
    2.3.1 Overview .............................................. 13
    2.3.2 The data model of Cassandra ......................... 13
  2.4 Cloudy .................................................... 16
    2.4.1 Cloudy Consistency Guarantees ....................... 16
    2.4.2 Atomicity in Cloudy ................................ 17
    2.4.3 Thrift API ........................................... 17

3 Open Source Databases 19
  3.1 Requirements .............................................. 19
  3.2 PostgreSQL ............................................... 20
    3.2.1 General Information ................................ 20
    3.2.2 Evaluation .......................................... 20
  3.3 Apache Derby ............................................. 21
    3.3.1 General Information ................................ 21
    3.3.2 Evaluation .......................................... 23
  3.4 MySQL ..................................................... 24
    3.4.1 General Information ................................ 24
    3.4.2 Evaluation .......................................... 24
  3.5 Decision .................................................. 25

4 MySQL Storage Interface 27
  4.1 MySQL Pluggable Storage Engine Architecture .......... 27
    4.1.1 Features ............................................. 27
    4.1.2 List of different storage engines of MySQL ....... 29
    4.1.3 Create Tables with Different Storage Engines .... 30
  4.2 Storage engine API ....................................... 30
### Contents

4.3 Example queries .................................................. 31
  4.3.1 Full Table Scan ........................................... 31
  4.3.2 Primary Key Query ......................................... 32
  4.3.3 Secondary Index Query ..................................... 32
  4.3.4 Conclusion .................................................. 33

5 Data Model Mapping .................................................. 35
  5.1 Mapping of Cloudy to MySQL ................................. 35
  5.2 Index Structures ............................................... 36

6 Implementation Details ............................................. 39
  6.1 MySQL Storage Layer ......................................... 39
  6.2 Cloudy Modifications ........................................ 40
  6.3 System Setup ................................................. 40
  6.4 Known Issues .................................................. 40
    6.4.1 Configuration and Meta Data Management .............. 40
    6.4.2 Other Issues ............................................ 42
    6.4.3 Problems with Cloudy .................................. 42
  6.5 Lessons Learned from Working with a Buggy Code Base .. 42

7 Performance Testing and Benchmarking ......................... 45
  7.1 Used Data .................................................... 45
  7.2 Microbenchmarking ............................................ 46
    7.2.1 System Setup ........................................... 46
    7.2.2 Queries ................................................ 46
    7.2.3 Results ................................................ 47
    7.2.4 Analysis ............................................... 48
  7.3 TPC-W Benchmark ............................................. 50
  7.4 Discussion .................................................. 50

8 Conclusion and Future Work ...................................... 53
  8.1 Conclusion .................................................. 53
  8.2 Future Work ................................................ 53
1 Introduction

1.1 Motivation

Within the last ten years the new medium Internet has attracted a great many people, with the effect that web applications are used by tens of thousands of users and even millions if the application proves to be successful. With so many users the magnitude of data to store and the amount of requests of data to handle is huge and can quickly become a limiting factor. Traditional relational database systems and older parallel and distributed database systems can often not scale up sufficiently to such demanding scenarios. Traditional database systems intend to provide consistency at all times and when they are distributed try to achieve distribution transparency. This means that they are aimed to give the end user the impression that there is only one single system instead of a number of collaborating systems. They even go as far as rather failing completely than breaking this transparency [7]. This leads to the fact that a traditional database system is not available when one of its partitions or subsystems is not accessible.

From the point of view of the providers of highly demanded web services the most important property they ask for in a storage system is availability and they cannot tolerate down-time in their service. This leads to the interesting trend that providers of highly successful services implement their own cloud storage solutions and abandon the traditional RDBMS for these applications. Examples are Amazon S3 [1], Yahoo P NUTS[9], Google BigTable[8] and many others. These storage systems are designed to reliably store huge amounts of data and to scale up to thousands of machines in order to be constantly available.

1.2 Problem Statement

There are trade-offs to be made as the well known CAP theorem [18] shows. You can have at most two of the properties consistency, availability, and tolerance to network partitions at the same time in any shared data system. The mentioned cloud storage solutions cut back at the consistency often only supporting eventual consistency[30] to maintain the other two properties. But this relaxed consistency is not always desirable for all kind of usage scenarios, while others are fine with it. The problem with the varying consistency needs was the reason for another cloud storage system, called Cloudy, which is being build at the ETH Zürich [19]. Cloudy is based on the cloud storage system Cassandra and extends it by implementing adaptable transaction guarantees inside of the storage service which can be altered for different usage scenarios. Users of Cloudy can decide on a finer grain if they want a higher availability or more consistency per
data item thanks to its adaptable transaction guarantees.

Another disadvantage of the current cloud storage services is the interface they provide to insert and access the data from a user point of view. They often only provide a very simple interface where one can access and insert merely unstructured values stored under a specific key or similar basic methods. If the end users want to get more complex and aggregated data results they have to gather the individual data pieces bit by bit from the storage system and combine them themselves. This is a big step backward from traditional RDBMS which often support a rich query and manipulation language as an interface to its data, enforces the use of a schema and tries to minimize data duplicity with good practices and normal forms. This also brings the added benefit that often data stored in RDBMS is application independent and can be used in different applications whereas the data stored in cloud storage systems is often bound to a certain application.

1.3 Contribution

This leads us to the subjects for this master thesis which tries to merge these two approaches for data storage solutions and intends to implement a SQL-interface on top of a cloud storage system. More concretely we aim at using Cloudy as the storage layer for a traditional database. Through the elasticity of the Cloudy system which can add and remove nodes from its running cluster we would get a highly scalable database system where the scalability comes from the storage layer. By using a traditional database as the underpinning basis we can still retain its SQL layer and reuse the query language and benefit from many internal algorithms.

Our new database system can be accessed by the standard APIs and connection libraries of the database. This enables a lot of existing third-party application to use our system without modification. Additionally, already existing software can be made scalable by switching the underlying storage layer of the database.

1.4 Structure of this Thesis

As the motivation stated our intention is to build a highly scalable database which uses Cloudy as a storage layer. The summary of the following chapters below outlines the steps taken to achieve this goal.

Chapter 2 describes the characteristics of cloud storage systems with special emphasis on Cassandra and Cloudy. It also discusses the consistency guarantees of these systems and how this guarantees affect our project. This chapter also explains the data model Cloudy uses and the API to store and retrieve data.

Chapter 3 lists the requirements a database needs to fulfill to be used as our base system. We compare and evaluate different databases and will finally announce which will be used in the further project.
Chapter 4 presents the MySQL storage interface. Furthermore a number of examples are given on how some common SQL queries access the storage layer.

Chapter 5 illustrates the data mapping from the table based SQL to Cloudy. This chapter will also show how index structures are stored in Cloudy.

Chapter 6 discusses some of the implementation details of this project. It also gives an overview of some obstacles that had to be overcome.

Chapter 7 presents the results of some benchmarks we did. It compares the performance of our project with a standard MySQL distribution. In the end we will analyze the results.

Chapter 8 is the conclusion of this work. It reviews our goals and examines if we achieved them. Thereupon it provides some pointers about possible future work.
2 Cloud Storage Systems

In this chapter we try to give a definition for cloud computing and cloud storage systems. We will demonstrate what the features and capabilities of a “standard” open source cloud storage system are. In the further sections we will go deeper into the features that Cloudy provides, being based on Cassandra and adding adaptable transaction guarantees. A description of the different consistency guarantees and other features of Cloudy will be given. In the end we will explain the data model of Cloudy and the interface it provides.

2.1 Cloud Computing

The concepts cloud computing and cloud storage are closely related. To understand one concept you also have to understand the other. In this section we will start the explanation by defining cloud computing.

For many years the Internet has been represented as a “cloud”, and the term cloud is often used in computer science jargon as a synonym for the Internet [28]. Furthermore the term cloud computing is now used to describe various service offerings over the Internet like Amazon EC2 or Google App Engine. But still it is hard to describe every aspect and try to define what cloud computing really is. Here is a list of the main properties that many of these experts called [17]:

**Internet centric:** As our definition of cloud already suggest, a cloud computing service is available in the Internet and has to offer an API through which a client application can interact with the service.

**Scalability:** A cloud computing service is highly scalable. Users can acquire computing resources depending on their demand to avoid under-utilization but also over-utilization. Some systems even scale up and down automatically.

**Availability:** One of the most important aspects of cloud computing for end users is that the service is almost constantly available.

**On demand and pay what you use:** A user of a cloud computing service only has to pay for the resources he actually uses. The user can acquire and release these resources, like server time and network storage on demand. In a open source service that would be running in house it could also mean that you only have to provide the hardware that is necessary and be able to attach or remove machines from the running system.
The combination of high scalability and high availability requires a cloud computing system to be as flexible as possible. That means that new machines should be able to be added to the running cluster at runtime and without having an impact on the system itself. This property is also known under the term elasticity. It is further essential that a single failing machine, or in a large scale system even a failing data center, does not bring the whole service down.

A thorough definition of cloud computing which also incorporates delivery models and deployment models can be found at [23].

### 2.2 Cloud Storage Systems

Cloud storage systems are a subset, but often also the foundation of cloud computing systems and therefore share the properties listed above. There are a plethora of different cloud storage services around and here again it is very hard to give an exact definition. On one side there are services which enable the end user to store simple files as a backup solution. On the other hand there are a lot of new database systems which are commonly called key/value stores. The expression Key/value store is an umbrella term for services which are often also described as document-oriented, attribute-oriented, distributed hash table, and key/value database [6]. All of these names are a variation on the same theme which emphasize different aspects of these systems. Because of the frequent changes in this field and and due to the fact that every few weeks a new systems appears on the horizon, we will refer the reader to online lists to see a comparison of such systems [3, 20]. Using the term cloud storage system we will refer to one these systems in the rest of this thesis.

One of the key distinction characteristics of cloud storage systems compared to traditional RDBMS is the different and diametrically opposed emphasis on consistency and availability. We have already mentioned the CAP-theorem in the introduction. The older traditional RDBMS are built around the historical basis that consistency is the main aim to achieve in a database system. As a result they often struggle scaling up vertically beyond a small number of server nodes. With many servers, the failure rate increases, and by enforcing consistency the availability decreases.

Cloud storage systems on the other hand are built around the premise to be available almost constantly and to scale out on a large number of nodes. This is necessary to serve the needs of highly demanded web services which operate on a worldwide scale. Yet high availability can only be achieved by cutting back at the consistency guarantee.

Instead of full ACID guarantees as traditional RDBMS most cloud storage systems support BASE properties. BASE stands for basically available, soft-state and eventual consistency [27]. Subsequently, we will discuss the different consistency guarantees and explain what eventual consistency means in 2.4.1.
2.3 Cassandra

2.3.1 Overview

Cloudy is based on Cassandra [2], therefore it makes sense to examine the features and also the shortcomings of Cassandra a bit better before we discuss Cloudy.

Cassandra is a highly scalable, eventual consistent, and distributed storage system that brings together the underlying message and gossiping system from Amazon Dynamo [11] and ColumnFamily based data model from Google BigTable [8]. Cassandra, initially developed by Facebook and open sourced in the summer of 2008, has been recently adopted as an Apache Incubator project. Cassandra is in production use by Facebook where it is responsible for the inbox search feature, which is up to 40 Terabyte of data across 120 machines in two separate data centers.

Cassandra is written in the Java programming language and runs on a standard Java virtual machine, it runs on any system that also provides a JVM.

With the gossiping system from Dynamo, Cassandra is very stable and has no single point of failure. The failure detection is decentralized and every node can learn about the arrival or failure of another node. This means that in a running Cassandra system new nodes can be added and removed at run time, but the system cannot currently add or remove these nodes automatically a feature which is called cloud bursting.

In Cassandra the data gets partitioned among the nodes according to the key of the single data rows. Concretely a partitioner class decides on which Cassandra node the data will get stored based on the hash value of the key. Currently Cassandra provides two partitioners out of the box. A random partitioner which, as the names suggests, distributes the data rows randomly, and a order preserving partitioner which distributes similar keys to the same nodes. The order preserving partitioner is also used to provide range queries to the system. Range queries return a list of data rows whose key is in a certain range. The order preserving partitioner is necessary in such a case so that the system only needs to contact a minimal number of nodes to return the data for a range.

For the moment Cassandra only provides eventual consistency with weak read and non blocking writes. This means that for a read the data is immediately returned when the first value of a replica has arrived and the system does not wait for all replicas. When a write is issued, the system automatically returns and updates the value internally to all replicas without waiting for an acknowledgment from a replica. Cloudy provides a finer consistency model that is controllable by the administrator, subsection 2.4.1 goes into more detail.

2.3.2 The data model of Cassandra

Cloudy shares the data model with Cassandra. We will introduce Cassandra data model in this chapter.

The data model of Cassandra is based on the data model of Google BigTable, but adds another optional dimension. At first sight it can be confusing. It is easiest understood if you imagine it as a four- or five-dimensional hash map. Part of the confusion is coming
Chapter 2. Cloud Storage Systems

from the fact that the names that Cassandra uses for certain data model properties are often not very intuitive in our opinion.

In order to explain the data model we will start by giving you an example of what information you need to provide to access a data value in Cassandra, then we will explain the components. We begin with the case of accessing the value of a so called simple column, which has four access dimensions:

\[ \text{TableName}, \text{Key}, \text{ColumnFamilyName}, \text{ColumnName} \rightarrow \text{Value} \]

Additionally it is possible to give only three dimensions and get as a result a list of tuples in the form of (ColumnName, Value):

\[ \text{TableName}, \text{Key}, \text{ColumnFamilyName} \rightarrow (\text{ColumnName}, \text{Value}) \]

The table has currently no big significance in Cassandra and exists only to group the column families. In the newest version the table is now called keyspace to distinguish it more from a table in a RDBMS. The basic access point is the column family which belongs to a table. The column families have to be defined before the cluster is run. Therefore it is expected that there are relatively few column families in the system. A column family is identified by its name which is a string. Each instance of a simple column family from our access examples can have an unbounded number of dynamically created columns. The key is a string and can be of any size. The key must be unique in a column family. The columns are also identified their names and contain a value which can be an arbitrarily byte stream. The data associated with a key in a given column family is known as a row in Cassandra. What is important to understand is that there can be any kind and any number of columns per row and the column names and values are not predefined and can also vary inside of a single column family.

The visualization in figure 2.1 should help to better understand the concept.

The column families can also be in the form of a super column family. With a column family of the type super, another dimension is added to the data model. A super column family can contain an unlimited number of super columns which themselves can contain an indefinite number of simple columns. If you provide all the five dimensions you can access the value stored under a super column:

\[ \text{TableName}, \text{Key}, \text{SuperColumnFamilyName}, \text{SuperColumnName}, \text{ColumnName} \rightarrow \text{Value} \]

It is also possible to provide only four or three dimensions and get as a result a list of tuples in the form of (ColumnName, Value) or (SuperColumnName, ColumnName, Value) respectively:

\[ \text{TableName}, \text{Key}, \text{SuperColumnFamilyName}, \text{SuperColumnName} \rightarrow (\text{ColumnName}, \text{Value}) \]
\[ \text{TableName}, \text{Key}, \text{SuperColumnFamilyName} \rightarrow (\text{SuperColumnName}, \text{ColumnName}, \text{Value}) \]

This added dimension distinguishes the data model of Cassandra from other key/value stores.
Chapter 2. Cloud Storage Systems

Figure 2.1: An example of the Cassandra data model.
This image illustrates examples for a simple column family and a super column family. All the access dimensions are shown with an example value.
The example shows a possible data model for a mailbox. The simple column family messages contains the data for two messages. A message is identified by a key, which in this case is a number. There are several columns as for example the text column which contains the text data, the sender column which contains the id of the sender and more columns.
The super column shows an example for word index. The super column row with the key “99” contains a reversed index for all the words which are in the messages of the user with the id “99”. The word “hello” is the name for two super columns. One with column name “15” contains the value “2” which indicates that in the message with id “15” the word “hello” exists two times.
Chapter 2. Cloud Storage Systems

It is important to understand the implications and properties of the data model to work efficiently. First, it depends on the key on which data node the column family data is stored. The rows get either distributed randomly among the nodes or range based according to the partitioner which has been chosen. Out of this characteristic comes the limitation that the data for a row which is a key/column family combination must fit on disk on a single machine on the cluster. This also means that all the data for super column row must fit on a single node.

2.4 Cloudy

Cloudy is a cloud storage system developed at ETH Zürich. It is based on Cassandra and adds different features to it. In this section these features are demonstrated. At the end we will talk about the Thrift API which enables Cloudy to communicate with other systems and end users of Cloudy.

2.4.1 Cloudy Consistency Guarantees

Like Cassandra, Cloudy supports eventual consistency as its base consistency level. Eventual consistency is a specific form of weak consistency. It basically guarantees that when an update is made to an object, the system will eventually return the last updated value through every access of the same object. The time in which an update has arrived to the system when the system might still return the old value is called the inconsistency window.

In Cloudy the consistency level can be controlled by the end user and even strong consistency can be achieved by the use of quorum protocols. Strong consistency guarantees that after an update completes, every subsequent access to the updated object will return the updated value. That means that no inconsistency window exists in a system that guarantees strong consistency.

Before further explaining adaptive consistency guarantees and the different consistency levels that Cloudy supports, we will establish the following definitions:

- \( N \): the number of replicas of a data item
- \( R \): the number of replicas the system needs to contact in order to read an item
- \( W \): the number of replicas that need to acknowledge the update of an item before the update completes

In Cloudy the administrator can decide on the number of replicas, the size of the quorum read set \( R \), and the size of the quorum write set \( W \) for each column family that is defined, and even for each individual row separately if he desires.

By changing the size of these sets one can have some interesting properties. In the case when \( R + W > N \) the write set and the read set always overlap and we get strong consistency. In order to achieve good performance and high availability the number of replicas need to be bigger than one. If the system wants to provide good read
performance R can be set 1. In this case only one node needs to be contacted to read a data item. If the write set W is set to 1, only one node needs to acknowledge the arrival of a write which leads to good write performance.

For example, a system that sets \( N = 3 \), \( R = 1 \) and \( W = 1 \), is only eventually consistent and there is a possibility of conflicting writes. On the other hand a system that sets \( W = N \) is strong consistent but with the price that writes will fail if a node fails. This is also a consequence of the CAP-theorem.

So the choice on how to configure \( N \), \( R \) and \( W \) depends on the importance of strong consistency and on whether you want to achieve higher read or higher write priority. The article [30] goes deeper in demonstrating the different consistency levels.

In Cloudy the replication number is often set to 3 and the setting \((N = 3, R = 2, W = 2)\) works well for most use cases by still providing strong consistency. Read or write intensive application could set \( R = 1 \) or \( W = 1 \) respectively. Applications that can pass on strong consistency and want to provide high performance and availability could try the setting \((N = 3, R = 1, W = 1)\).

The basic Cassandra version does not allow to set the size of the quorum write sets and quorum read sets. Only the replication factor can be changed. Additionally in the most current version of Cassandra only supports weak reads and non blocking writes which corresponds to a setting of \( R = 1 \) and \( W = 1 \).

\subsection*{2.4.2 Atomicity in Cloudy}

Atomicity is the “all or nothing” property of a transaction. Atomic transactions guarantee that a transaction either completely finishes or does not have any effect on the system. Similar to the different consistency settings, Cloudy can switch on or off atomic updates on a per column family and even on a per row basis.

Cloudy implements atomic transactions with the 2-phase commit protocol. When a client contacts Cloudy to perform an atomic transaction the contacted node acts as the transaction coordinator. The 2-phase commit protocol has the drawback that it blocks if the coordinator fails. Also, in the presence of atomic transactions the operations cannot complete successfully if the number of available nodes is smaller than the size of the quorum read or write set respectively and will fail.

For more details about the 2-phase commit protocol consult a book about distributed systems like [14].

\subsection*{2.4.3 Thrift API}

The client applications that want to communicate with Cloudy can do it over a remote procedure call interface implemented with Apache Thrift [29].

Thrift is a software framework for scalable cross language service deployment. Its main goal is to enable efficient and reliable communication across programming languages. Thrift supports a interface description language and a compiler which can create server and client stubs out of this IDL into a lot of programming languages like
C++, Java, Python, etc. Thrift also takes care of the underlying communication protocol implemented with TCP sockets and provides an implementation for all its supported programming languages. Originally, Thrift was developed at Facebook. Thrift was open sourced in April 2007 and entered the Apache Incubator in May, 2008.

Cloudy provides a Thrift interface file and implements the client side of this interface. The interested clients can generate client stubs out of this interface file in any language they want.

The following operations are supported to communicate with Cloudy:

- **list<columns> transaction_read(i64 txId, string tablename, string key, string columnFamilyPath)**: This operation reads a list of columns. The columnFamilyPath must be in the form “ColumnFamilyName:columnName” for simple columns or “SuperColumnFamilyName:SuperColumnName:ColumnName” for super columns. The SuperColumnName and the ColumnName can be set to “null” in order to match all columns of a column family. The txId is optional and can be set to 0 if the operation should not run under a transaction.

- **void transaction_write(i64 txId, string tablename, string key, list columns)**: This operation allows to store a list of columns under a running transaction. The columnFamilyPath must be an element of each column in the list in the same form as above. The txId is optional and can be set to 0 if the operation should not run under a transaction.

- **void delete_single(string tablename, string key, string columnFamilyPath)**: This operation allows the deletion of columns.

- **i64 begin_transaction()**: Starts a transaction and returns a valid transaction Id.

- **void abort_transaction(1:i64 txId)**: Aborts the transaction with the given Id.

- **void commit_transaction(1:i64 txId)**: Commits the transaction with the given Id.

- **list<columns> fullTableScan(string tablename, string columnFamilyName)**: This operation returns all the columns which are stored under a given column family. As the name suggests, it serves mainly to support rather fast full table scans for the use of the attached database.
3 Open Source Databases

The aim of this thesis is to take an existing database and replace the storage layer. This chapter first describes a list of requirements a database needs to fulfill to be considered a candidate. Then an evaluation and comparison of the three database PostgreSQL, Apache Derby and MySQL is given. Eventually, the chosen database will be presented.

3.1 Requirements

The first and foremost requirement for a database to fulfill in order to be included in our work is that:

The storage layer must be easily replaceable!

In consequence of this requisite there are a number of further requirements which should be fulfilled and are caused by the need to have a replaceable storage layer:

- Open source license: In order to change the storage layer we must have access to the source code of the database. Furthermore we want to be able to release the source code of our implementation afterward and it would also be desirable to be able to use it commercially without paying license fees.

- SQL: It is important that the database supports the SQL query language. We want to be able to reuse the internal parser and optimizer.

- Programming language: The database must be implemented in a programming language that can communicate with Cloudy and that we are sufficiently proficient in.

- Documentation: It would be helpful if the internal structure of the storage layer was well documented either in the source code itself as comments or also as external documents.

- Internals: The path a query takes in the database should be easy to follow also in the internal source code structures. The location where the actual data is accessed should be replacable. In addition we should be able to control how transactions are managed as we want to delegate the actual handling of the transaction to Cloudy.
According to these mentioned requirements we have chosen the following three database for further examination.


In order to make a qualified statement about the replacability of the storage layer it is not enough to only study the documentation of the database. We had to dig into the source code of the databases and examine especially the parts responsible for storing the data.

In the subsequent sections we will discuss and evaluate every database.

### 3.2 PostgreSQL

#### 3.2.1 General Information

PostgreSQL is an open source object-relational database system with more than 15 years of active development its roots are even older and are based on the database Postgres which was initially developed by Micheal Stonebraker. It runs on all major operating systems and supports almost the whole SQL-92 and SQL-99 standard. It is released under the very liberal BSD license. PostgreSQL is written in the C programming language. The documentation for end users is rather extensive and the official manual contains a whole section about some of the internals of the database see [26].

The liberal license and the general good reputation of PostgreSQL in the database community made it the first candidate we examined further although no information about different storage layer implementations were known.

There exist several connection libraries and a JDBC-driver to access the database from various programming languages.

#### 3.2.2 Evaluation

We start the evaluation by checking which stages a query takes in PostgreSQL:

1. The end user or an application has to establish a connection to the PostgreSQL server and send a query.
2. The query is checked for correct syntax and a query tree is created.
3. The planner/optimizer is responsible to generate an optimal execution plan for the given query. It does so by examining each of the possible plans if this is computationally feasible. The resulting plan tree consists of sequential- or index-scan nodes for the base tables, plus possible join nodes as needed, and sort or aggregate function nodes.
4. In this last step the executor steps recursively through the plan and fetches the rows. Each time a node in the plan is called, it has to deliver a row. The sequential- and index-scan nodes are finally responsible to fetch the data for the rows from the storage layer.

As the final stage revealed, we had to examine the source code for the sequential scan and for index scan, and its interaction with the storage layer more precisely to make a decision if we could use PostgreSQL as our database.

It turned out that PostgreSQL assumes that all data is stored in pages on the file system and to fetch a row it has to first read in the desired page and then in this page it has to look for the actual tuple data to return. The graphic 3.1 shows a simple call graph in the case where a sequential scan has to fetch a row from the storage layer. The places in the source code where the actual data is read is very dependent on the fact that the data is stored in this page layout on an actual hard disk. Often a few pages are read in advance into a buffer and on the layer above it the data is caught by delivering many parameters to indicate which exact page and offset of it is currently needed.

To be able to change the storage layer one would have first to break this dependency in the code and provide an abstract interface to fetch the desired data. This would result in a rewrite of quite a big portion of the code.

### 3.3 Apache Derby

#### 3.3.1 General Information

Apache Derby is an open source relational database management system that is entirely written in Java. In 1997 Cloudscape Inc. which was later bought by Informix Software released the first version of this database under the name of Cloudscape. Eventually IBM acquired the technology in 2001 and donated the source code to the Apache Foundation in 2004. It is now an official Apache DB subproject and maintained by the Apache foundation. It is released under the Apache License Version 2.0.

Apache Derby is a very lightweight database: it only uses about 2MB of class files and about 4MB of the Java heap with the running JDBC driver. Derby is still a full database and implements the SQL-92 standard and many SQL-99 and SQL-2003 extensions\(^1\) that means it implements transactions with commit and rollback, support multiple connections with transactional isolation, and provides crash recovery. Summarized it supports full ACID compliance.

Due to its small footprint, Derby can be either used as an embedded database in the same JVM as the client application. But Derby can also be setup as a standalone database server in a client/server mode where the communication happens through the network. The database is accessed through JDBC which is a database access layer for Java.

\(^1\)Refer to the page in the Wiki (http://wiki.apache.org/db-derby/SQLvsDerbyFeatures) for a list of features which are implemented in the current version.
Chapter 3. Open Source Databases

Figure 3.1: A simplified call graph showing how a sequential access node fetches a tuple from the storage layer.

ExecSeqScan

heapgetpage

heapgettuple

ReadBufferExtended

smgrread

src/backend/executor/nodeSeqscan.c
responsible to return next row

src/backend/access/heap/heapam.c
fetchs a tuple from the page in the heap

src/backend/access/heap/heapam.c
reads in a page from a relation

src/backend/storage/buffer/bufmgr.c
returns a buffer containing the requested block of the requested relation

src/backend/storage/smgr/smgr.c
reads a particular block from a relation into the supplied buffer
There is not much documentation of the internals of Derby. Some information can be found at [13, 12]. But the details are rather sparse.

We were especially encouraged to evaluate Apache Derby further for our purposes by another master thesis written at the ETH[21] whose topic was the modularization of Apache Derby. But up until the version 10.4 of Apache Derby, which was the most current version when we did our evaluation there, was only the standard disk storage option as the data backend for Apache Derby. But with the recent release of the feature version 10.5 they added an in memory storage backend option to the official version.

3.3.2 Evaluation

Similar to the evaluation of the other databases we tried to follow the path from a SQL query in the Derby source code statically and also in a running system in the debug mode. The stages for the query are very similar to the general stages of a query in PostgreSQL. The SQL query gets parsed and optimized and finally Java byte code will be generated for the query. The resulting class is loaded and executed and will start to communicate with the storage layer.

On several occasions the Derby development team announced that there is a clean separation of the storage interface and the storage implementation similar to the Plug-gable Storage Architecture of MySQL like in this blog post [24]. Unfortunately this clean separation is not really present in the current version of the Apache Derby source code. The store layer itself is split into two main parts, the so called access area and the raw area. The raw area provides access to the data on a page granularity. The access layer sits on top of the raw layer and provides a row based interface to the SQL layer. This is exactly the interface we were looking for. But it is very hard to understand what is going on in this access layer. The call stacks to access the data from a simple SQL-query was very large and a lot of methods were called whose intention were not clear from the name and the code. Although containing a lot of comments, the documentation is generally of too low a level. The higher level explanations are not sufficient to understand the interaction of the Database with the storage layer. It was for example very hard to isolate certain crucial information from the access layer like the table names of the involved queries or the key values when the storage is accessed through an index, information we would need to code our own storage layer.

After two weeks of tinkering with the Derby source code it was deemed too time consuming and too risky to code our own storage layer for Apache Derby. Although there seems to be good intention from the Apache Derby developers to provide a pluggable storage architecture it would take a whole master thesis to refactor the code to make this possible.
3.4 MySQL

3.4.1 General Information

The first version of MySQL was released in 1996. It was mostly the work of a single developer, Monty Widenius. He wanted to have a database with an SQL interface but he was not satisfied with the speed that other databases have given him. So he developed his own database. The first early version of MySQL lacked transactions and several other crucial features. But around the year 2000 a new company called MySQL AB was established which hired several developers to provide a SQL-interface to Berkeley DB in order to support transactions. The Berkeley DB storage backend was finally implemented and released with the version 3.23 of MySQL although it was never very stable. But the integration work resulted in various hooks in the MySQL source code which enabled other programmers to develop other storage layers. With the version 4.0 another storage engine called InnoDB was integrated into the main line of MySQL. With this second integration the abstraction of the storage layer with the rest of the system even improved. MySQL with the InnoDB storage engine provides ACID-compliant transactions along with foreign key support. MySQL was bought in February 2008 by Sun Microsystems, which was itself acquired by Oracle in 2009.

The MySQL source code is released under the GNU General Public License and a commercial license. MySQL is mostly written in the C++ language with some parts in pure C. There is a lot of documentation available for MySQL. On one hand there are very extensive reference manuals for each release version available online at http://dev.mysql.com/doc/ and also very extensive documentation printed in book form like [15]. Additionally there is an active forum, which is administrated by the owners of MySQL, where you can ask questions available at [5]. Even the documentation for the inner workings of MySQL is more extensive than all other reviewed open source databases with some highlight like [25, 16]. MySQL support a broad subset of SQL-99.

3.4.2 Evaluation

MySQL already advertises its pluggable storage structure, which is exactly what we are looking for. A deeper look into the source code revealed that they really keep their promise saying that it is reasonably easy to implement a new storage layer. We will go into more detail about this structure in the next chapter. The overall impression of the source code was also good. The code is well documented and with the help of the excellent internal documentation it is rather easy to follow the path a query takes from the connection up to the data fetching.

Yet we also have to consider that we are bound to release the source code of our own storage engine under the GPL in the case we want to distribute it.
Chapter 3. Open Source Databases

Table 3.1: Overview table of the evaluated databases

<table>
<thead>
<tr>
<th></th>
<th>PostgreSQL</th>
<th>Apache Derby</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>License</td>
<td>BSD</td>
<td>Apache License version 2.0</td>
<td>GPL</td>
</tr>
<tr>
<td>Programming language of the system</td>
<td>C</td>
<td>Java</td>
<td>C++/C</td>
</tr>
<tr>
<td>Internal documentation</td>
<td>medium</td>
<td>rather sparse</td>
<td>extensive</td>
</tr>
<tr>
<td>Code structure</td>
<td>good and easy to follow</td>
<td>sometimes hard to understand</td>
<td>good and easy to follow</td>
</tr>
<tr>
<td>Time needed to code storage engine and risk involved</td>
<td>high</td>
<td>high</td>
<td>low to medium</td>
</tr>
</tbody>
</table>

3.5 Decision

We decided to make our own storage layer in the database MySQL. In the end the good internal documentation and also the many different storage engines we could look into as an example has turned the balance toward MySQL. These attributes decrease the risk involved and also the time needed to implement our own storage layer.

Table 3.1 summarizes the decision points we considered in our evaluation.
4 MySQL Storage Interface

The source code of MySQL, especially in the area of the storage management, is very modular and allows an end user to change the storage layer very easily. This is mostly a result of the history of MySQL which is briefly outlined in the last chapter.

The general architecture of MySQL follows the general architecture of most RDBMS: A connection can be made through several connector libraries for a lot of different languages. Then the query is parsed an optimized. Eventually the data is fetched from the storage layer. A high level view of this architecture can be found at 4.1.

In the rest of this chapter we want to discuss what benefits the pluggable storage engine architecture brings to end users. Then we will explain in more detail the concrete API that enables this pluggable architecture. Finally we will show how some example queries are finally translated to calls to the storage engine API.

4.1 MySQL Pluggable Storage Engine Architecture

4.1.1 Features

The pluggable storage engine architecture is a unique feature of the MySQL database. It gives users the possibility to chose from a list of purpose built storage engines which are optimized for various use cases. If a user has a very specific need he can even implement his own storage engine. As the name “pluggable” indicates, the server administration can add or remove storage engines in a MySQL server at runtime as the storage engines can be packaged as plugins.

The different storage engines try to satisfy different needs and there are a lot of different features a storage engine might provide to fulfill certain performance, consistency or other requirements. From a technical perspective there are a lot of variation points for different engines. In the following list we describe what we think are some of the most important points an implementation of a storage layer has to consider.

Transaction properties: Not every application needs transactions. The ones that do might not need full ACID-transaction guarantees.

Scalability and high availability: Our own implementation of a storage engine wants to fulfill this goal. But not every storage engine needs to be highly scalable and be available all the time.

Storage of the data: There are a lot of different ways the data can be stored. One can change the file layout or page sizes of the data tables. The storage of the indexes also has to be considered. The physical layout of the storage might already be
given through the need of another application. This thesis demonstrates that you can even get rid of storing the data on a hard-drive and store it in a cloud storage system.

**Index structures:** Different applications might benefit from different index structures. Ranging from traditional b-tree implementations to specialized data structures for Geo-spatial or other applications.

**Performance:** This could include thread support to use all the cores of a server, bulk inserting of data and a lot of other things.

**Integrity:** A feature that many database supports is referential integrity which are often implemented with the help of foreign key support. This removes the responsibility away of checking every inner-table relationship from the application programmer. But not every storage engine needs these additional integrity checks.

**Caching:** It is the job of the storage engine to decide if and how to implement caching. The decision is highly dependent on other internal factors of the storage engine.

**Miscellaneous features:** The storage engine could need special security restrictions. It could be responsible to gather the data from various places and integrate it in the storage engine for the interested users. The list of possibilities are almost endless.

As the list demonstrates there are many different storage engines conceivable. They might range from a very simple storage engine which can only be used to query logs or data files produced from even a different application in a read-only fashion, but also a full fledged storage engine like InnoDB which supports almost any feature that is
expected from a full-fledged database, like high performance index structures, row-level
locking and full ACID compliance. Even a very different like our own implementation
approach is possible.

4.1.2 List of different storage engines of MySQL

The official MySQL distribution already includes several storage engines which serve
different purposes. Here we list the most interesting and most used ones.

**MyISAM:** MyISAM is the default storage engine if the server is not configured oth-
erwise. The implementation of **FULLTEXT** indexes enables full text search capa-
bilities. MyISAM has no support for transactions and referential integrity. Its
locking granularity for concurrent accesses is on the table level. Still it has the
reputation of having a high performance especially on applications which are read
intensive. Yet another disadvantage to consider is the poor crash recovery, with
data files which can become corrupt after a crash [32].

**InnoDB:** InnoDB is another well known storage engine in MySQL. It supports full ACID
compliant transactions with the four SQL transaction isolation levels, serializable,
repeatable read, read committed and read uncommitted [31], while repeatable
read is the default isolation level. InnoDB ensures data integrity with the imple-
mentation of referential integrity. InnoDB uses row-level locking instead of table
level locking which may increase the performance of write intensive concurrent
applications. InnoDB can be considered the most appropriate choice for most traditional appli-
cations needing transactions. MySQL combined with the InnoDB storage engine
supports almost any feature that is expected from a RDBMS.

**MySQL cluster:** Another interesting storage architecture is the MySQL cluster, inter-
ationally also called the **NBDCLUSTER** storage engine. It supports a shared-nothing
clustering architecture without a single point of failure, when the data is repli-
cated on more than one data node. In a minimal setup you need three nodes to
start a MySQL cluster, one management-node, one SQL-node and one data node,
but at least two data nodes are strongly recommended to have data redundancy.
The so called SQL-nodes run the MySQL-daemon service and are the access points
for the end users. The cluster is designed to run in closed network because the
data communication between the nodes is not secured and the data traffic between
the nodes might be very high. MySQL cluster uses synchronous replication during transactions through the us-
age of the two phase commit protocol [14]. But the clusters only implement read
committed as the isolation level for transaction. The MySQL cluster does not
support foreign keys. Interestingly the most current version of the cluster (version
7.0) even supports adding new data nodes to a running cluster systems under
certain circumstances [4]. However it is not known how well the system scales and
what the limit is on how many data nodes are supported in the cluster.
Chapter 4. MySQL Storage Interface

**CSV storage engine:** The CSV storage engine is a very simple storage engine which stores the tables in a comma separated values text file. It does not support any sophisticated features like indexes or transaction. But its files can be read and even written by spreadsheet applications. The CSV storage engine serves as an educational example for interested programmers who can study its source code.

### 4.1.3 Create Tables with Different Storage Engines

In MySQL every table can be created with a different storage engine. The working of the MySQL server and the communication of the different tables do not depend on the used storage engine. So a lot of interesting setups are possible with different tables which use different storage engines according to its requirements.

The actual engines that are available depend on the MySQL server. To get a list of all the available storage engines the server supports you can issue the `SHOW ENGINES` command in a MySQL client. Here is the output of the command running in our own test server which is based on version 5.1 of MySQL.

```
mysql> SHOW ENGINES;
+---------+-------------+--------+
| Engine  | Comment                  | Trans. |
|---------+-------------+--------+
| EXAMPLE | Example storage engine  | NO     |
| MYISAM  | Collection of identical MyISAM tables | NO |
| InnoDB  | Supports transactions, row-level locking, and foreign keys | YES |
| BLACKHOLE | /dev/null storage engine (anything you write to it disappears) | NO |
| CLOUDY_STORE | Cloud based storage engine which uses Cloudy | NO |
| MEMORY  | Hash based, stored in memory, useful for temporary tables | NO |
| ARCHIVE | Archive storage engine | NO |
| FEDERATED | Federated MySQL storage engine | NULL |
| CSV     | CSV storage engine | NO |
| FOO     | Foo Example storage engine | NO |
| ndbcluster | Clustered, fault-tolerant tables | NULL |
```

To create a table with a specific engine we need to give the engine name as an option to the `CREATE TABLE` command like in the following examples:

```
CREATE TABLE test1 ( ... ) ENGINE = InnoDB;
CREATE TABLE test2 ( ... ) ENGINE = CSV;
```

### 4.2 Storage engine API

The MySQL server communicates with the storage layer through a predefined API. In the MySQL source code this interface is implemented in the abstract class `handler`. The handler class provides abstract methods for basic operations on tables, like opening and closing, full scanning of the table, retrieving rows by indexes and primary keys, inserting, updating and deleting rows.

Each storage engine needs to subclass this handler class and has to implement the abstract methods to translate the low/level storage operations to the engine. The concrete implementations of the handler are instantiated when the table is accessed on
Chapter 4. MySQL Storage Interface

Figure 4.2: A class diagram of handler with some of the most important methods.

a per thread basis. That means that for each connection that is made to a table, a new instance of a storage engine class is created.

The method names that the handler class provides are descriptive and indicate what they do as the image 4.2 demonstrates.

The handler class is only responsible for access methods on a per table basis. But some work needs to be done on a more global level. So each storage engine has access to a Singleton instance of a class called handlerton. The handlerton contains method pointers to methods that apply to the storage engine as a whole. These includes, among other things, methods for transaction handling, like prepare, commit and rollback.

More information about how to implement your own storage engine can be found at [25, 16].

4.3 Example queries

In this section we want to show which methods from the handler instance are called for some example queries. As our test data we take a single table called author which has a primary key and a secondary index and is filled with three records, more details can be found in figure 4.3.

4.3.1 Full Table Scan

As a first example we want to make a full table scan on our author table triggered by the following simple SQL-query:

```
SELECT * FROM author;
```

Once the query is parsed the following methods are called in our handler:

```
info // place to give hints to the optimizer
rnd_init // initiate the full table scan
rnd_next // fetch first row
rnd_next // fetch second row
rnd_next // fetch third row
```
The method responsible for fetching the data and returning it to the underlying server is `rnd_next(uchar *buf)`. The parameter buf is a byte array already preallocated by the system and needs to be filled with the whole row data in the internal MySQL format. The byte array needs to be filled with each column in the order it was defined in the `CREATE TABLE` statement. The source code for the CSV storage engine contains an example of how you can convert the data to the internal representation.

The `info` method that is called in the beginning can be used to set various flags as a hint to the optimizer. It can range from the number of tuples which are expected in the full table scan up to some obscure things which are not very well documented. We had to play around with different settings in this info method until we achieved something which worked well in most of the cases.

### 4.3.2 Primary Key Query

A primary key query is a SQL-query which has an equality operator in the `WHERE` clause on the primary key field defined for the table:

```sql
SELECT * FROM author WHERE a_id = 3;
```

The sequence of calls for this query is very basic:

```sql
info  // place to give hints to the optimizer
index_read  // fetch the row with the data
```

MySQL is aware that we access the table with a primary key which must be unique and therefore issues only a single call to the `index_read` method. The `index_read` has also byte array as a parameter that we need to fill with the data. Additionally it provides the value of the queried index value, the integer 3 in this case, as another parameter to the method. The information on which index we query at the moment is stored in a class variable in our handler subclass.

### 4.3.3 Secondary Index Query

We demonstrate the interactions of a secondary index query with the following query:

```sql
SELECT * FROM author WHERE a_lname = "Muster";
```

We expect that MySQL reads both of our rows matching this secondary index. The sequence of calls is only slightly more complicated than in the primary key query:
This time we have accessed a secondary index. MySQL cannot know in advance how many records are stored with our secondary index, except in the case where the index would be unique. So similar to the full table scan it calls the `index_next` method until the storage engine indicates that all records are read.

### 4.3.4 Conclusion

A query which accesses data can only take two paths, either it makes a full scan on the table and then makes something out of the fetched rows or it fetches only a few rows through indexes including primary keys. So with our three example queries we have covered all possible access paths.

The remaining methods `write_row`, `update_row` and `delete_row` to insert, update and respectively delete rows, give the whole record in the internal MySQL format as a parameter. These methods to manipulate the data are called once for each row in the table that needs to be manipulated.

As a final word we like to give an advice to someone who tries to implement his own storage engine. It turned out that the documentation of the abstract handler class and the example storage engines do not explain all parameter methods in the required depth and do not provide all information about what global settings need to be changed in some situations. Some inner quirks can only be found out by trial and error. But the best documentation of the handler interface is hidden in the source code of InnoDB and can be found in its implementation. Unfortunately this is neither advertised in the online documentation nor in the comments of the handler interface.

<table>
<thead>
<tr>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_id</td>
</tr>
<tr>
<td><em>a_fName</em></td>
</tr>
<tr>
<td><em>a_lName</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a_id</th>
<th>a_fName</th>
<th>a_lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Daniel</td>
<td>Muster</td>
</tr>
<tr>
<td>3</td>
<td>Peter</td>
<td>Muster</td>
</tr>
<tr>
<td>4</td>
<td>Anna</td>
<td>Meier</td>
</tr>
</tbody>
</table>

Figure 4.3: The test data for our query examples
5 Data Model Mapping

An important early step was to define how we want to adapt the relational data model that we get from using MySQL to the data model of Cloudy. This includes two main aspects. The first one is how to store the tables in Cloudy. The second main decision is how to store the indexes, especially the secondary indexes. The decisions were made early but some details were also changed after first testing versions showed how well they worked.

The first section of this chapter demonstrates how the relational tables are mapped to the Cloudy data model. Section 5.2 illustrates how secondary indexes are implemented.

5.1 Mapping of Cloudy to MySQL

It is important to keep in mind the data model of Cassandra and Cloudy as shown in section 2.3.2 to follow this section. We have chosen to store all the data from a relational table in a column family of the type simple. A column family per table is created.

Each row in the column family consists of a single column where we store the entire raw byte stream of the database record in the internal MySQL format. By using directly the internal MySQL byte format we avoid time consuming serializing and deserializing of the data with the disadvantage that we can only access the whole record at once. The Cloudy row-key to access the column will be the byte value of the primary key of the MySQL record encoded in Base64.

In table 5.1 the individual items are listed and the mapping of Cloudy to MySQL is summarized.

<table>
<thead>
<tr>
<th>Cloudy / Cassandra</th>
<th>RDBMS as in MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table / Keyspace</td>
<td>Database</td>
</tr>
<tr>
<td>Column Family of type simple</td>
<td>Table</td>
</tr>
<tr>
<td>Column Family Name</td>
<td>Fully qualified database table name</td>
</tr>
<tr>
<td>Key</td>
<td>Primary Key value in the internal MySQL format encoded as Base64</td>
</tr>
<tr>
<td>Column Name</td>
<td>Constant string e.g. “record_data”</td>
</tr>
<tr>
<td>Column Value</td>
<td>The whole MySQL record in its internal byte format</td>
</tr>
</tbody>
</table>

Table 5.1: Mapping of Cloudy data model to the MySQL data model.

Out of this mapping the following consequences arise:
Chapter 5. Data Model Mapping

By also using the primary key of the MySQL row also as the key in the Cloudy row accesses through the primary key in a SQL statement should be very fast. Because access by a key is also the main access method in Cloudy. A separate primary key index is not necessary.

One thing to consider is that each MySQL table must have a primary key defined, this is not a real drawback.

Furthermore the decision on which Cloudy node the data gets stored is based on a hash of the key so the data of a MySQL table gets automatically distributed among all the Cloudy nodes.

One choice that was made was to store the whole SQL row as a byte stream in the internal MySQL format instead of storing the single attribute values separately. So with this decision the system always has to read or write the whole byte stream when it is accessed to be read or updated. One advantage is that we can just dump the byte stream to all storage API methods without deserializing it, and all of these methods are asking for the whole row data anyway.

In figure 5.1 an example mapping of the author table data to the column family database/author is shown.

5.2 Index Structures

Secondary indexes are mapped to Cloudy with the help of a super column family. The table 5.2 describes the mapping.

In this mapping the index itself stores a copy of the actual MySQL row data. This was done to enhance performance. Early experiments showed that it is not very fast to only store a list of primary key values which belong to a certain index value. If only the primary key is stored the systems needs to first get the list of all primary keys for a certain index value and afterward get the actual data for each primary key.

With the current mapping we have only a single access path to read the data. Also, if we want to write, update or delete a record in a secondary index we can do that by...
### Chapter 5. Data Model Mapping

<table>
<thead>
<tr>
<th>Cloudy / Cassandra</th>
<th>RDBMS as in MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Column Family</td>
<td>All secondary indexes of a single table</td>
</tr>
<tr>
<td>Super Column Family name</td>
<td>Name of the table prefixed with “index”</td>
</tr>
<tr>
<td>Key</td>
<td>Index value</td>
</tr>
<tr>
<td>Super Column name</td>
<td>Index name</td>
</tr>
<tr>
<td>Column name</td>
<td>Primary key value of the record</td>
</tr>
<tr>
<td>Column value</td>
<td>Byte stream of the whole record in the internal MySQL format</td>
</tr>
</tbody>
</table>

Table 5.2: Mapping of a secondary index in Cloudy

![Super Columns](image)

Figure 5.2: Example mapping for a secondary index structure.

accessing only the desired column by using the primary key as a further dimension. But this flexibility comes with the cost of redundant data that we have to store for each index. Each index value contains now the whole record data. Additionally, there is now the limit that the data stored for each index value must fit on a single Cloudy node.

With the use of the index value as the key in Cloudy the data for every index value gets distributed among the Cloudy cluster.

The use of the primary key value as column name may seem redundant, but with its use the storage API methods can quickly access the desired row to update or delete the data.
6 Implementation Details

The implementation of the project basically consisted of two different parts. One part was to code the actual MySQL storage layer, see section 6.1. Another part was to make modifications in the Cloudy code base in order to make the SQL layer work well. These modifications are described in section 6.2. In section 6.3 we will show how a complete system of our database will look like when it is running. Furthermore, section 6.4 contains a discussion of some of the remaining bugs in the system. Unfortunately, the code base of the existing Cloudy system turned out to be quite buggy. In the last section we will discuss some of these bugs and try to give some suggestion on how to avoid these problems in the future.

6.1 MySQL Storage Layer

The new storage layer programmed for this thesis is called CLOUDY_STORE. The storage layer is based on version 5.1.32 of MySQL which was the most current stable version of MySQL at the beginning of this project. As MySQL is programmed in C/C++ our own storage layer is also programmed in C++. The storage layer has been successfully compiled under various Linux flavors and GCC versions.

The source code configuration and the compiling of big C/C++ projects can be a time consuming task. An early challenge was to compile the MySQL source code including our own storage layer with the thrift library. The thrift library relies on some C++ specifics, like exception handling, which are disabled by default in the standard MySQL source code. In the end some obscure compiler settings needed to be turned on to successfully compile the project with all needed libraries. This was very time consuming and many trial-and-error attempts had to be done, especially because the error reports after a failed C++ compilation are often very cryptic and not very helpful.

Apart from these initial problems the storage layer is mostly an implementation of the storage interface as described in chapter 4 with the data model mapping of chapter 5. Once a working implementation was developed a process of constant tuning and refactoring was started. The goal of the refactoring was mostly to improve the performance of the implementation while at the same time implementing more features.

As mentioned in the discussion of the storage interface some of the internal workings were not very well documented. In order to understand some properties we had to study the source code of many other storage implementations. The best insights delivered the source code for the InnoDB storage engine.
6.2 Cloudy Modifications

The initial goal was to treat Cloudy as a black box for our storage layer. We constantly ran some performance tests to see how fast our implementation worked.

Some of these tests revealed that we had issues with full table scans. Our initial plan to make full table scans worked the following way. First, a list of all primary keys belonging to a SQL table was stored. To make a full table scan, the system then had to read the full list of these primary keys for the table. After this the system had to issue a read to Cloudy for each primary key to access the record data.

As the short description indicated this meant a lot of single calls and traffic from and to Cloudy and turned out to be slow for our purpose. So we had to implement a method in Cloudy to quickly return all columns which belong to a column family. The internal Cloudy code already provided some hooks which facilitated this exercise. With this modification we could raise the performance of full table scans through an order of magnitude.

Cloudy being developed in Java, another problem was to find a communication bridge between Cloudy and the MySQL storage layer which is written in C++. To close this gap we used the thrift library which Cloudy already supports. The server side is implemented on Cloudy in Java. The client side was implemented in C++ and gets called from the storage engine. Some tests showed that the thrift library worked well for this task and did not introduce bottlenecks.

6.3 System Setup

To deploy a full running system with our new database we can start a arbitrary number of Cloudy nodes. On each node we can then also start a MySQL instance including the new storage layer. The starting of a MySQL instance is optional. The running MySQL instances are the access points to our system. With this varying number of MySQL instances we avoid having a single point of failure. The database with our setup can then be accessed in all the ways a normal MySQL database can be accessed including all already existing connection libraries.

Figure 6.1 shows a running system with five Cloudy nodes each attached with a MySQL instance.

6.4 Known Issues

This section gives an overview of the problems and shortcomings that we are aware of and are still present in the system.

6.4.1 Configuration and Meta Data Management

For the moment the system does not allow to create and modify SQL-tables while the system is running. All the meta data management has to be done manually before the
system is started.

The following list defines all the steps which need to be taken in order to setup a system which can be started:

1. For all tables that are used in MySQL two column families have to be defined statically in the configuration file of Cloudy, one for the record data and one for the index data.
   At the moment Cloudy does not allow to create new column families while the system is running so these column families must be defined before the system gets started.

2. In a second step the databases and the tables need to be created in each MySQL instance that will be used later. It is very important for the tables to be created in the same way in each instance, with the same indexes and the same data types, so as to avoid problems when the system is running.
   MySQL stores the metadata about the tables separately on the local hard disk.
   To get rid of this limitation we would have to be able to change this metadata dynamically and distribute the changes among the system.

3. As a last step first Cloudy and then the MySQL instance need to be started on each participating node.

All these steps can be done manually or by a script. It is absolutely necessary that the the same meta data is used on each node or otherwise the system will show strange behavior or will even crash without meaningful error messages.
6.4.2 Other Issues

- There remain some issues with the full table scan that we have implemented. In the current version the full table scan only reliably returns the data that is stored on the local node. There is a problem with the messaging system in Cloudy so often the messages with the full table scan data from the other nodes do not reliably get to the node which issues the full table scan.

- For some complex queries that are issued to the database, the optimizer will choose a suboptimal plan. It is possible to force the system to choose a specific index or ignore an index altogether. The exact interaction between the storage layer and the optimizer happens through some specific flags which are not very well explained. But the implementation as it stands now often chooses a reasonable execution plan that performs well most of the time. But from time to time it can happen that the end user has to force specific index usage which can speed up some query executions by a big factor.

- Under some unknown circumstances the result of a query which has an ORDER BY clause will return empty result sets although the same query without the ORDER BY clause will return a full result. This only seems to happen when all attributes from a table are getting asked for, because in this case a slightly different internal method call order is issued. The exact source of this problem remains unknown.

- There is a problem under some setups with attribute fields that are not stored directly in the record but through pointers, like BLOB and TEST fields. If the system is run with multiple nodes, sometimes the system will crash when such a field is accessed from a machine which did not insert the field. However, sometimes the fields can get accessed nonetheless. The error only occurred when the system was run under a 64-bit systems whereas running under in 32-bit system it seemed to work.

6.4.3 Problems with Cloudy

Due to some remaining bugs in the code base of Cloudy, Cloudy itself could not be used as the underlying system for our storage layer. Some of these problems are discussed in the next section. For our benchmarks we had to use the basic Cassandra cloud storage system as backend for our own storage layer. As a consequence we could not use the transactional features of Cloudy in our storage layer and the database itself only provides eventual consistency with weak reads and non blocking writes.

6.5 Lessons Learned from Working with a Buggy Code Base

A considerable part of the time allotted to this thesis was spent with the code base of Cloudy. One of the major tasks was to fix some defects which arised in some particular
Chapter 6. Implementation Details

cases when the system was used. Unfortunately, while we were tackling the known bugs a few other bugs which were even graver appeared. Although we finally failed to fix all the bugs in the system some valuable lessons were learned which we are trying to pass on in this section.

One of the worst remaining bugs in the system at the time we accessed the code was a bug we called the “data loosing bug”. You could insert some data through the standard API and under some circumstances this data could not get accessed anymore. This bug occurred when the data was flushed to disk, an operation which gets executed regularly in the system. So over the time, it seemed like the system constantly lost data. With this bug the system basically ceased to be usable. What is even more astonishing is that this bug remained undetected in the code base for almost six months. During these months the code was frequently updated. After a few weeks of debugging, it turned out that the bug was caused by an incomplete update of some serializing code. The code to serialize some of the internal data structures to disk, holding the column family data, was updated but the correspondent code to deserialize the data remained untouched. So the system responsible to read back the data from the disk interpreted the data wrong and always returned empty data sets. But this wrong interpretation of data was not considered as an error, neither was an exception thrown nor were any log messages issued at this particular point. After months, the origin of this bug was very hard to find. If there had been some automtisms to test the API the source of the bug would have been found and identified much faster!

Another bug that appeared when we took over the source code, was that super columns did not work anymore. Yet all the methods in the Thrift-API for super columns were still present. After looking through the whole Cloudy source code there was not one single test case or any example which used super columns. So over the time the super columns just ceased working even if super column support was still advertised.

Yet another problem with the code was that it was based on an early alpha version of Cassandra. This alpha version still contained a considerable number of bugs. But Cloudy missed the opportunity to gradually incorporate the newer updated versions of Cassandra. This resulted in the fact that many bugs that were fixed in the Cassandra code remained in the Cloudy code and needed to be fixed a second time or still remain undetected in the code. After almost a year while both projects were updated frequently the base version of Cassandra used in Cloudy diverged heavily from the most up to date version of Cassandra. So the task to update Cloudy to the newest version of Cassandra would be a major subproject itself now, which would require a few weeks of work.

A big challenge was to debug the system in a distributed way. The system seemed to work very well when run locally with a single node only. But a lot of errors solely appeared when the system was setup in a distributed manner with multiple nodes. Setting up a distributed system with multiple nodes can take a lot of time and one might be tempted to test new features only on a single node. But this might not reveal all problems and, even worse, could conceal the real reason for the problem when its only detected much later. So it is absolutely necessary that the system can be setup with multiple nodes in a automatic way. You also have to pay attention that the same
version of the software with the same configuration is run on every node of the system. This could be achieved with the help of startup scripts or similar methods and was often successfully applied in this project.
7 Performance Testing and Benchmarking

This chapter presents the results and the analysis of several benchmarks run on different system setups. Section 7.1 shows the data setup used for the benchmarks. As mentioned in the chapter before we needed Cassandra as our underlying storage system because some critical bugs remained in the Cloudy source code.

7.1 Used Data

The data used for the benchmarks come from the TPC-W benchmark [10]. The TPC-W benchmark emulates an online bookshop and simulates users browsing and shopping on its website. The data tables indicate all important entities that are used. Figure 7.1 shows the tables graphically.

For the microbenchmark in the following section we only use a subset of all tables.

![Data Tables of the TPC-W benchmark](image)

Figure 7.1: Data Tables of the TPC-W benchmark
7.2 Microbenchmarking

This section presents the microbenchmark used to compare the performance of different setups of the system. The microbenchmark tests the system with different simple queries which cover all access paths and also insert and delete. The systems are tested under increasing load by concurrently issuing the queries.

7.2.1 System Setup

Different setups are tested and compared:

**Setup MySQL InnoDB:** A single MySQL server instance using InnoDB as the storage layer.

**Setup Cassandra One:** A single Cassandra node with a running MySQL instance using Cassandra as the storage layer.

**Setup Cassandra Five Replica:** Five Cassandra nodes each running a MySQL instance using Cassandra as the storage layer with the replication factor “five” for the data. This means that once the data gets written to the system it is replicated to all nodes.

The setup with a standard MySQL with the InnoDB storage engine is used to compare our own storage layer implementation against the standard MySQL. In this case we use InnoDB because it is one of the most widely used storage engines. The different Cassandra setups are used to show if the number of nodes in Cassandra has an effect on the performance of the systems.

The hardware used to perform the tests are five 64 bit machines with an AMD Dual Core Opteron 280 Processor and 6 GB of RAM memory. Each is running a Linux distribution with kernel version 2.6.18. The machines all run in the same internal network.

7.2.2 Queries

In this microbenchmark a number of simple queries are used. We tried to cover all access paths and also included a set of queries to test the insert and delete performance of the system. Here is a list of all the queries:

**Primary Key Query**

The following query is issued 2000 times in a row per run to get a performance number item:

```sql
SELECT * FROM item WHERE i_id = <random number>
```

The item table contains 10000 items. The random number is a valid id of one of the items in the list. The `i_id` attribute is the primary key for the table and this query basically tests the access path for primary key accesses.
Chapter 7. Performance Testing and Benchmarking

Secondary Index Query

The following query is run 250 times in a row per run:

\[
\text{SELECT count( i_id ) FROM item WHERE i_subject = "<random subject>"}
\]

A secondary index is created on the attribute \textit{i_subject}. The item table contains 10000 items and there are 24 different \textit{subjects} in the system. Each subject should reference about 400 items. The count function in the query enforces that each item of a subject needs to get accessed. With this query we test the access path over secondary indexes.

Full Table Scan

The following query gets run 50 times in a loop per run:

\[
\text{SELECT count( ol_id ) FROM order_line}
\]

There exist 7750 records in the \textit{order_line} table. With this query each record needs to get accessed and a full table scan is issued.

Bestseller Query

The so called bestseller query is used to test a more complex query which accesses a lot of tables. The bestseller query gets the 50 best selling items ordered by the number of orders from the most recent 3333 orders. We do not show the query here and point the interested readers to the TPC-W specification. The bestseller is only run once per run in the tests.

Insert and Delete Query

In order to test the insert and delete performance of the database we insert and subsequently delete the same ten records in the \textit{author} table with the following queries. The author table also contains a secondary index on the \textit{a_lname} attribute. So both insert paths are covered. The insert and delete queries are executed 1000 times per run.

7.2.3 Results

The benchmark was run with the different setups as explained above on the mentioned hardware. All the queries demonstrated in previous section were run. Each measurement point is the aggregated average time of 25 consecutive runs. The results are shown in detail in tables 7.1, 7.2 and 7.3. All measured times are given seconds. In the case where more than one MySQL-instance was running, as in the setup with five Cassandra nodes, the connections to the MySQL instances were equally distributed among all the running instances.

Unfortunately some benchmarks for the Cassandra setup with the insert and delete query could not be measured to the end. The corresponding fields were left empty and are labeled with a *. By running the insert and delete benchmark under this medium
load already beginning with 25 concurrent threads, the Cassandra setups became slower and slower. The benchmarks were aborted when a single run proved to take more than ten minutes. One side effect that could be monitored was that the memory consumption became bigger and bigger until it reached the maximum heap memory allocated for Cassandra, almost four gigabytes of heap memory. When this number was reached the system started to trash and the measured times started to grow exponentially high. In some cases the database crashed, and with it even the server machine. However, the memory needed to store the whole data is comparatively low only a few hundred megabytes at most. This behavior in Cassandra combined with the effect that the memory of the process does not get freed up after the abort of the benchmark seems to indicate a memory leak in the handling of inserts and deletes.

Table 7.2 shows that in the setup with only one Cassandra node the threshold for the read performance is already achieved with about 100 concurrent accesses. InnoDB still works reasonably well with 200 threads, even the insert and delete queries successfully finish under this high load.

7.2.4 Analysis

Scaling of Cassandra with Varying Number of Nodes

Table 7.4 gives a comparison of the running times between the setup with one Cassandra node and the setup with five Cassandra nodes. As the numbers indicate the systems seems to scale and can handle much more load if the number of Cassandra nodes is higher. This is especially apparent in the case when 100 threads are accessing the system at the same time. The setup with one Cassandra node seems to reach a threshold while the setup with five nodes works still well as the whopping difference of 2694 % in run time indicates.

Comparison of Cassandra with InnoDB

Table 7.5 presents the performance of Cassandra compared to a standard MySQL setup with InnoDB.

The almost linear increase of performance of the Cassandra database compared to InnoDB under the growing load up to 100 threads looks promising. However, with the last measurement using 200 and 300 threads the performance of the Cassandra system in comparison to InnoDB already starts to decline again.

Furthermore, this microbenchmark only allows to compare the performance with equal queries. The bottleneck for the Cassandra database lies especially in the insert and delete performance. As explained above, the run time just explodes when too many update queries are issued at the same time. In a mixed setting where all kinds of queries are issued, which is the normal case for a database in the web environment, these update queries just decrease the performance of the whole system.

The numbers for the full table scan have to be considered with a grain of salt. The current implementation of the full table scan with the Cassandra-MySQL database only
Table 7.1: Results for MySQL with the InnoDB storage engine.

<table>
<thead>
<tr>
<th># of threads</th>
<th>1</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>key query</td>
<td>0.658</td>
<td>0.687</td>
<td>1.263</td>
<td>2.971</td>
<td>6.227</td>
<td>35.982</td>
<td>62.230</td>
</tr>
<tr>
<td>secondary index</td>
<td>0.235</td>
<td>0.558</td>
<td>1.474</td>
<td>2.998</td>
<td>6.642</td>
<td>12.750</td>
<td>19.294</td>
</tr>
<tr>
<td>full table scan</td>
<td>0.108</td>
<td>0.315</td>
<td>1.296</td>
<td>3.395</td>
<td>6.643</td>
<td>13.814</td>
<td>21.294</td>
</tr>
<tr>
<td>bestseller</td>
<td>0.097</td>
<td>0.466</td>
<td>0.998</td>
<td>1.888</td>
<td>4.582</td>
<td>7.669</td>
<td>12.901</td>
</tr>
<tr>
<td>insert and delete</td>
<td>1.212</td>
<td>2.968</td>
<td>7.215</td>
<td>17.343</td>
<td>249.687</td>
<td>413.176</td>
<td>501.816</td>
</tr>
</tbody>
</table>

Table 7.2: Results for Cassandra setup with one node.

<table>
<thead>
<tr>
<th># of threads</th>
<th>1</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>secondary index</td>
<td>1.229</td>
<td>4.910</td>
<td>12.929</td>
<td>20.767</td>
<td>46.419</td>
<td>125.669</td>
</tr>
<tr>
<td>full table scan</td>
<td>1.737</td>
<td>7.035</td>
<td>15.777</td>
<td>36.251</td>
<td>72.325</td>
<td>92.463</td>
</tr>
<tr>
<td>bestseller</td>
<td>1.800</td>
<td>4.170</td>
<td>6.121</td>
<td>18.128</td>
<td>25.558</td>
<td>37.482</td>
</tr>
<tr>
<td>insert and delete</td>
<td>1.187</td>
<td>10.943</td>
<td>.*</td>
<td>.*</td>
<td>.*</td>
<td>.*</td>
</tr>
</tbody>
</table>

Table 7.3: Results for Cassandra setup with five nodes and replication factor five.

<table>
<thead>
<tr>
<th># of threads</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>key query</td>
<td>2.639</td>
<td>4.177</td>
<td>7.959</td>
<td>15.241</td>
<td>49.807</td>
<td>79.654</td>
</tr>
<tr>
<td>full table scan</td>
<td>2.737</td>
<td>4.323</td>
<td>7.849</td>
<td>14.888</td>
<td>27.301</td>
<td>56.075</td>
</tr>
<tr>
<td>bestseller</td>
<td>3.706</td>
<td>7.209</td>
<td>12.771</td>
<td>25.850</td>
<td>71.885</td>
<td>97.188</td>
</tr>
<tr>
<td>insert and delete</td>
<td>7.133</td>
<td>.*</td>
<td>.*</td>
<td>.*</td>
<td>.*</td>
<td>.*</td>
</tr>
</tbody>
</table>

Please read the text in section 7.2.3 to get an explanation for the empty results marked with a *.
returns the tuples which are stored in the local node. The potential time needed when other nodes need to send their tuple data over the network is not measured and ignored in this benchmark. In this setup we always return all the nodes from a table, but in a bigger setup where the replication factor is not as high as the number of nodes, the time needed for a full table scan will increase.

7.3 TPC-W Benchmark

In order to test the system with a more realistic workloads than the microbenchmarks, we ran an implementation of the TPC-W benchmark. The implementation used as the TPC-W is the product of another ETH master thesis [22]. TPC-W simulates the workload of an online book store. It defines the data setup and also the look and the access paths on the web site. A total of 14 web pages are described in detail in the TPC-W specification. User are emulated via a remote browser emulator which generates the same HTTP-traffic as a real customer with a browser.

The primary metric of the TPC-W benchmark are WIPS, web interaction per seconds. For every web interaction TPC-W specifies a maximum processing time which a web interaction must fulfill to be counted as a valid web interaction. The benchmark adapts its load to the maximum throughput of the system under test. For more details we refer to the mentioned thesis above [22] and the official TPC page [10].

We performed the TPC-W benchmark with three different setups:

1. A single Cassandra node with one MySQL instance.
2. Three Cassandra nodes each running a MySQL instance with replication factor three.
3. Five Cassandra nodes each running a MySQL instance with replication factor five.

Table 7.6 displays the results we achieved. The numbers underline the impression that the database with Cassandra as the storage layer scales with the increasing number of nodes. Unfortunately we are never as good as a single MySQL instance using InnoDB as the storage engine. On the same hardware with the same data setup, InnoDB was able to achieve more than 250 valid WIPS [22].

7.4 Discussion

In the preceding paragraphs, we were able to demonstrate that our database system with the Cassandra storage layer does scale. Unfortunately, scalability is not the same as performance and throughput. Even in a setup with five Cassandra nodes, our system was not able to achieve the performance and throughput of a single MySQL database with the InnoDB storage engine. Furthermore we have to take into account that MySQL provides full ACID guarantees, while our system only provides BASE.

It is astonishing how well InnoDB adapted to the relatively high workload compared to Cassandra. A single Cassandra node already reached a threshold with 100 threads
Table 7.4: Comparison between Cassandra 1-node and Cassandra 5-node.

<table>
<thead>
<tr>
<th># of threads</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>key query</td>
<td>56 %</td>
<td>97 %</td>
<td>98 %</td>
<td>230 %</td>
<td>2694 %</td>
</tr>
<tr>
<td>secondary index</td>
<td>150 %</td>
<td>188 %</td>
<td>164 %</td>
<td>241 %</td>
<td>497 %</td>
</tr>
<tr>
<td>full table scan</td>
<td>250 %</td>
<td>364 %</td>
<td>461 %</td>
<td>654 %</td>
<td>621 %</td>
</tr>
<tr>
<td>bestseller</td>
<td>112 %</td>
<td>84 %</td>
<td>141 %</td>
<td>137 %</td>
<td>144 %</td>
</tr>
<tr>
<td>insert and delete</td>
<td>153 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

This table tries to show if the Cassandra database scales. It shows the running times of the Cassandra setup with five nodes and replication factor five in reference to the Cassandra setup with one node. The performance of Cassandra with one node is the reference point considered as 100 %.

Table 7.5: Comparison between Cassandra 5-node and InnoDB.

<table>
<thead>
<tr>
<th># of threads</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>200</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>key query</td>
<td>26 %</td>
<td>30 %</td>
<td>37 %</td>
<td>38 %</td>
<td>40 %</td>
<td>72 %</td>
<td>78 %</td>
</tr>
<tr>
<td>secondary index</td>
<td>17 %</td>
<td>21 %</td>
<td>23 %</td>
<td>25 %</td>
<td>26 %</td>
<td>15 %</td>
<td>15 %</td>
</tr>
<tr>
<td>full table scan</td>
<td>11 %</td>
<td>29 %</td>
<td>43 %</td>
<td>46 %</td>
<td>44 %</td>
<td>51 %</td>
<td>37 %</td>
</tr>
<tr>
<td>bestseller</td>
<td>12 %</td>
<td>12 %</td>
<td>14 %</td>
<td>17 %</td>
<td>17 %</td>
<td>10 %</td>
<td>13 %</td>
</tr>
<tr>
<td>insert and delete</td>
<td>16 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Comparison of the performance of InnoDB to the Cassandra setup with five nodes and a replication factor of five. The numbers show the percentage of the performance achieved with the Cassandra database. The performance numbers of InnoDB are the reference points considered as 100 %.

Table 7.6: Results of TPC-W benchmark

<table>
<thead>
<tr>
<th>setup</th>
<th>max. number of valid WIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>one node</td>
<td>8</td>
</tr>
<tr>
<td>three nodes</td>
<td>12</td>
</tr>
<tr>
<td>five nodes</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 7.6: Results of TPC-W benchmark
while InnoDB was still running better under a load of 300 threads than the setup with five Cassandra nodes. We did not have the resources to deploy a system with hundreds of Cassandra nodes, such as the one in production use at Facebook. However, in a system with many nodes, the internal communication traffic would increase with the number of nodes, especially in the presence of full table scans, which is a very common operation in SQL databases.

Still we were able to demonstrate that every single access path in our database works reasonably efficiently. If there was no internal problem in Cassandra with inserts and deletes under a high workload, we assume that the results for the TPC-W benchmark would be better.
8 Conclusion and Future Work

8.1 Conclusion

We started work for this thesis with the aim of building a scalable database with a cloud storage system as its storage layer. The benchmarks in the end have indeed shown that this goal has been achieved.

In particular, we were able to implement a new storage layer for the MySQL database which is based on the Cassandra storage system. A comparison of different open source databases revealed that, using the pluggable storage architecture of MySQL, it is possible to build a new storage layer in a reasonable amount of time. With the use of MySQL as our front database we support all the SQL features that MySQL implements. Every library and third party application, which supports MySQL, can now use our cloud storage based layer without modifications.

In the course of the thesis we tested several possibilities on how to adapt index structures and other important database methods to the data model of Cassandra. Eventually, we have found a mapping that works efficiently for almost all use cases and table structures.

Finally with the results of the benchmark we were able to demonstrate that our database scales, and that a database setup with five Cassandra nodes can handle a much higher load than a setup with only one Cassandra node.

We were also able to identify a bottleneck in Cassandra with insert and delete queries. If we issue a considerable amount of inserts or deletes concurrently, the memory need of Cassandra explodes and the performance of the whole system is brought down. So, although our database scales, it can still not compete with a standard MySQL implementation performance-wise.

8.2 Future Work

Automatic Metadata Management

In the current state, a lot of configurations need to be done manually and it takes a lot of time to setup our database system with multiple nodes. If the system could setup and deploy new nodes automatically, a lot of time could be saved. The required configuration could be centralized in one file belonging to the MySQL process. It would also be very convenient if new tables could be created on the fly and if all the necessary meta information was distributed automatically among all the participating nodes.
Chapter 8. Conclusion and Future Work

Full Table Scans
Currently, a full table scan does not return all the data belonging to a table, but only the data that is stored on the local node where the scan is issued. In a future version the database needs to contact all nodes which are storing data for the desired table. It is important for the contacted nodes to return all the data in one run. It is not very efficient when other nodes can send data only row by row.

Transaction Guarantees and Different Consistency Levels
Cassandra, the system we used as our storage layer, only guarantees eventual consistency with weak reads and non blocking writes. At the beginning we wanted to use Cloudy as the storage layer, which would have provided adaptable transaction properties. Unfortunately, we could not use Cloudy because of some critical errors which still remain in the code base. In a future version it would be interesting to incorporate some of the transaction guarantees into our database through the storage layer.

Leverage Distributed Features
The current implementation of the storage layer treats Cassandra as a block box and does not take advantage of some features which would be possible in the distributed setting. The system could example route the query to the node which stores the data. It would also be very interesting to explore the performance differences of data shipping against query shipping. We could push down predicates or aggregates to remote nodes.
Bibliography


[27] Dan Pritchett. Base: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability.


