Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 34

Systems Group, Department of Computer Science, ETH Zurich

Database Logging System

by
Martin Alig

Supervised by
Prof. Dr. Donald Kossmann

March 2012 – September 2012

inf | Informatik
Computer Science

Abstract

In most computer systems log records are an important source for failure detection, monitoring, statistics and various kinds of analytics. Simple log file generation as it is often implemented is not sufficient anymore for large computer systems generating hundreds of gigabytes of log data per day. Those systems need a more sophisticated log storage platform which is able to store hundreds of thousands of log entries per second and offering high performance query capabilities. This thesis proposes a solution using open-source components like Apache HBase and Apache Solr and presents benchmarks of concrete implementations using real data from Amadeus, world's leading travel transaction operator.

Acknowledgements

I would like to thank my mentor Prof. Dr. Donald Kossmann for giving me the opportunity to work on this interesting thesis and for his guidance and support. I would also like to thank the people from Amadeus, especially Thomas Pare and Paul De Schacht, for giving me an introduction to the logging system at Amadeus in Nice and providing us with real world data which was crucial for the completion of the thesis. Finally I would like to thank Georg Polzer for the good collaboration comparing the implemented system to Splunk.

Contents

Chapter 1

Introduction

Software log records are an important source for monitoring, failure tracking and different kinds of analysis. In most simpler software systems log records are just stored in plain text files. Having larger and more complex systems and especially distributed systems, this becomes a problem. The logs are distributed over multiple machines and split into multiple files. Searching through the log data gets very cumbersome, time consuming and mostly requires some custom tools. Also, these systems tend to produce a large amount of log data, which can be in the range of gigabytes to even terabytes per day.

The goal of this thesis is to explore a solution to this problem: A scalable database system for logs which supports high throughput and a wide range of query capabilities. To achieve this, open source components such as Apache Hadoop, Apache HBase and Apache Solr are used, whereas HBase acts as the main storage platform for the log data and Solr adds search functionality to the unstructured contents of the log records.

1.1 Use Case: Logging at Amadeus

Amadeus is the worlds leading technology provider to the travel industry providing marketing, distribution and IT services worldwide. The current systems produce several hundred thousand log records per second. For their business, the logs are very important to respond to user problems, react on issues, monitor the current state of the system and as a source for statistics. As many different software components generate log entries, the entries are very diverse and contain only little common structure. See appendix A for some examples. As the entries are produced at a very high rate, it is not feasible to extract the complete structure from the log messages. Also thousands of different message grammars exist which makes it almost impossible to store the messages in a completely structured format.

1.2 Contribution

The thesis proposes a system consisting of open-source components which is able to handle large amounts of data and high throughput. The thesis shows different implementation approaches and for the experiments concrete implementations have been contributed. Various experiments show that the performance and capabilities can be compared to a commercial tool like Splunk.

1.3 Thesis outline

Chapter 2 covers all theoretical fundamentals of HBase and Solr which are important for the subsequent chapter. In chapter 3 we discusses the design of the solution, findings that were made during the progress of the thesis, alternative solutions and concrete implementations. Then chapter 4 contains experiments and benchmarks using real data from Amadeus and also a comparison with the commercial log analysis platform Spunk. In chapter 5 we draw a conclusion and give some ideas for future work.

Chapter 2

Theory

2.1 Apache HBase

HBase is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable [6, 2]. It runs on top of the Hadoop Distributed File System (HDFS) and is part of the Apache Hadoop project. Simply said, HBase could be described as a big multi-dimensional sorted map.
The development of HBase dates back to the release of the Bigtable paper in 2006. After that, a first HBase prototype has been implemented as a Hadoop contribution and in 2008 HBase became a sub project of Hadoop.

2.1.1 Architecture

HBase consists of three components: one master, multiple regions servers and a client. Further HBase needs an existing HDFS setup for storage and also a ZooKeeper quorum for coordination. Figure 2.1 shows a high-level view over the different components. In the following sections, each of the components is explained and some functionality and features of HBase are presented.

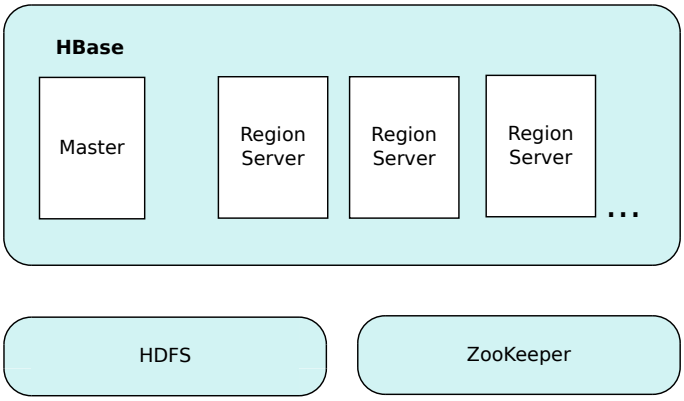

Figure 2.1: HBase Overview

Master

The master server does not store any actual data, but it is responsible to assign regions to the region servers. By assigning regions to region servers it takes care that the load is balanced and single servers are not overloaded. Further it provides all administrative tasks such as table creation and schema changes. When the master goes down, a running cluster can still function. The master does not hold any information that is needed for the cluster to work. Nevertheless, it should be restarted as soon as possible.

Region

A region is the basic unit of scalability and load-balancing in HBase [4] and is responsible to store the actual data. A region always contains a contiguous range of rows stored together. Clients communicate directly with the region servers to write and retrieve data. A single region is always served by exactly one server but a server can have multiple regions. Regions have a configured maximum size and therefore, when a region grows too large, the region server splits the region at the middle key into two parts.

Client

To get in contact with a HBase cluster, the clients have to know the address of the ZooKeeper server. From the ZooKeeper instance the client receives the location of the so called "-ROOT-" table[1]. The root table contains information about the ".META." tables[2] which in turn store where a certain region is placed and what row range the region stores. Then clients can look up the needed meta table and receive the location of the required region. For a more efficient look up, clients usually cache the region locations, the location of the meta tables and the location of the root table.

HDFS

HDFS is the default file system when deploying HBase, but theoretically it could be replaced by any other file system. The advantage of HDFS is that it is scalable, fail safe and also offers automatic replication. Also Hadoop MapReduce fully supports HDFS and therefore it is a good choice as the underlying file system.

ZooKeeper

Apache ZooKeeper is an open-source project and its aim is to provide a highly available distributed coordination service [12]. Simply said, it offers a service to clients with directories and files that can be used to store data, register services or watch for updates. HBase uses ZooKeeper for electing the master (if multiple master servers are configured), for storing the location of the root table and as a registry for the region servers. Also the master uses ZooKeeper to communicate cluster state changes to the region servers.

[1] Subsequently, the -ROOT- table is referred as *root table*.
[2] Subsequently, the .META. table is referred as *meta table*.

2.1.2 Data Model

HBase can be described as a sparse, distributed, persistent, multidimensional sorted map. Each single value or cell is indexed by a row key, column key and a timestamp (see figure 2.2). The row key identifies the whole row and based on the row key the data is stored in a region. The column key itself consists of a column family and a column qualifier. Column families is a concept to group data together, therefore values from the same column family are stored together and can also be compressed together. The last part of the key is the timestamp which is used to identify versions. Usually the timestamp is automatically generated by the region server but can also be specified by the client.

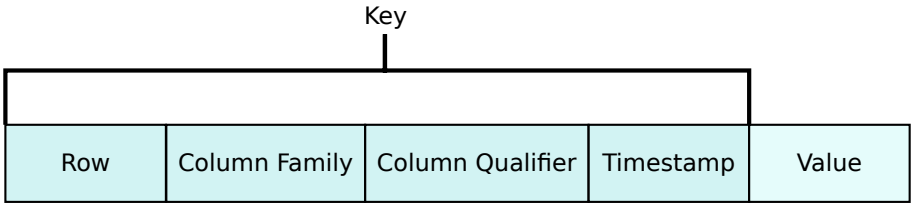

Figure 2.2: HBase Key [4]

2.1.3 Storage

Write

The way how the data is stored to disk is similar to how a Log-Structured Merge Tree works [9]. First, when the data arrives at the region server it is put on the write-ahead log (WAL). The WAL guarantees atomicity and durability. After an entry has been put to the WAL, the region server writes the data into the memory store. Using the WAL which writes directly to disk, it is made sure that no operations are lost on a server crash. The memory store itself is configured to a maximum size and when reaching that threshold, the data is flushed to disk. A flush from memory store to disk creates multiple files, called store files. To keep the size and the number of those files under control, a compaction is triggered from time to time. A compaction is the process of merging multiple store files together and there exist two different types of compactions:

- Minor compaction: The minor compaction is usually triggered after a flush based on a selection logic. It is responsible to merge multiple files into a bigger one. The selection logic of the minor compaction works as follows: a file is selected for compaction if the size of the file is smaller than the sum of all smaller files multiplied by a configured factor.
- Major compaction: This compacts all files of a region into one single file. The major compaction usually does not run very often, e.g. once every 24 hours. A major compaction normally increases the performance as it tries to improve data locality by moving the data blocks to the server where the region is hosted. However, in larger and loaded systems, major

compactions should be managed manually as they rewrite all the stored data which could influence the rest of the system.

As Hadoop mainly follows the sequential read/write pattern, the original HDFS file formats were not suited for HBase's random and realtime access patterns. New storage files have been implemented called HFile. The HFile has been designed after the *SSTable* format from Bigtable but since the beginning it has undergone various changes. Generally speaking, an HFile is an immutable, persistent and ordered map from keys to values where an index is stored at the end of the file. When a file is opened the index is loaded into memory so that afterwards look-ups can be performed with a single disk seek. For a more complete overview how the files are designed, we refer to [4] and especially [1] for a complete overview of all versions.

Read

As seen before how a write is performed in HBase, a row can be at two places: in the memory store or on disk. Therefore, for each read operation, it is first checked if the row is in the memory store. If not, the row is loaded from disk. When the data is loaded from disk, it could be spread over multiple files. Hence HBase actually performs a scan over the store files until it has collected all the needed data.

Delete

Rows are never directly removed from HBase. When a client issues a delete, a marker is written to the row. On future reads, the row with the delete marker will not be returned. The next major compaction will then finally delete the data. Further, column families in HBase can be configured to have a time to live. This makes HBase automatically delete rows when the expiration time is reached.

2.1.4 Write-Ahead Log

As discussed before, write operations are not directly stored to the underlying HDFS but are kept in memory. This is very efficient but introduces the possibility of a data loss. Thus HBase comes with a write-ahead log. Every operation is first written to the log, which resides directly on HDFS. The WAL file is just a list of operations and each region server has exactly one active WAL at every point in time. Each edit operation in the WAL file has a unique sequence id to guarantee ordering. Now when the memory store is flushed to disk, the highest sequence id is also stored as meta information. This allows HBase after a failure to look for the highest sequence id in the persistent store and then replay all operations from the WAL file with a larger sequence id.

2.1.5 Coprocessors

Coprocessors are a feature added to HBase after the Bigtable feature has been presented on a Google presentation [3]. The idea is to have parallel computations where the data resides. In HBase there are two variants of Coprocessors implemented:

- Observer: These behave similar to triggers from conventional databases. User code gets executed when certain events occur. For example before insert, after insert, before read, and so on. Right now three different types of Observers exist: *RegionObserver*, *WALObserver* and *MasterObserver*.
- Endpoint: An endpoint is user code that can be executed on the region server on user request, e.g. a dynamic remote procedure call endpoint or in the context of traditional databases a stored procedure.

One point to note is that Coprocessors in HBase run in the same process as the region server. This is a fundamental difference to the Coprocessors proposed by Bigtable, where they run as a separate process.

2.2 Apache Solr

Apache Solr is an open-source search platform from the Apache Lucene project. It is a server application built around the Lucene full text indexing and search engine core and adds features like distributed indexing, distributed search, index replication and more [10]. As Lucene is just a Java library which has to be integrated, Solr does that. It wraps Lucene inside a server application that can be accessed using HTTP. Further it leverages the need to interact directly with the Lucene internals by being configurable through XML files and a web-based administrative interface. As Solr is accessible through HTTP it can integrated in almost any existing system in any language and for Java there already exists an easy to use client library. Internally, Lucene stores data in a flat structure consisting of documents, which in turn contain several fields as key value pairs. As for our implementations we only used small parts of Solr and Lucene's capabilities, we do not go into more details about the architecture and design of the two.

2.3 Log Files

Current applications usually produce or can be configured to produce log files. Log output is used at various stages of software development or deployment. Developers use log output for debugging or profiling. Later on deployed applications, the log output is used to monitor the application, for failure detection and identification, or just for some information about the current state of the application. The reasons to generate log output can be almost anything and in the end its the developers decision what to log and what not.
Log files are mostly endless growing files as long as the application is running. The structure inside the files can be very complex. Usually there is a fixed structure for the single entries which contains attributes like timestamps, identifiers or other application specific, recurring attributes. But the actual content or the message of the log entry is mostly unstructured text. That could be a developer defined message, application variables, application generated contend like XML or other formats, error messages like stack traces or even binary content. Of course one could define a structure for log entry messages, but as systems grow, different applications work together and logs from multiple applications are collected, multiple formats come together.

Chapter 3

Design and Implementation

3.1 Overview

As seen in the previous section, log entries usually have a structured part which can be decoded easily and also stored in a structured way. Then there is an unstructured part. Assuming that the system to construct has to perform under very high loads, meaning hundreds of thousands of log entries per second, it is not possible to decode every single log entry message without deploying large amounts of computational power. Therefore, a generic way has to be found to store the unstructured content and still have it accessible.
In figure 3.1 we introduce a simple format for a generic log entry which will be used throughout the thesis. As can be seen, the log entry has a unique identifier, multiple fixed attributes and a message part which is unstructured and one has to assume that it can contain any content. None of the attributes is mandatory and also the message can be empty in some cases.

Figure 3.1: Log Entry

3.1.1 Basic Storage

HBase offers high throughput for write and read operations and has built-in replication. This makes it an ideal candidate for the main storage system. The schema proposed to store the log entries can be seen in figure 3.2. The row key is the unique identifier of the log entry. Then there is one column family defined named "l". The column family contains all fixed attributes and also the log message. The qualifier for the attributes is simply their name, the qualifier for the message is just "m". Short column family names and qualifier names should be preferred, as they are always part or the value key and therefore also stored.

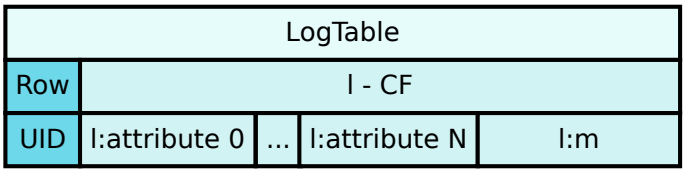

Figure 3.2: Log Table Schema

3.1.2 Secondary Indexes

After the basic storage of the log entries has been constructed, we can look at the current retrieval capabilities. Each entry can directly be accessed using the unique identifier. But the structured attributes can not be queried directly and a scan has to be performed. HBase has no built-in support for secondary indexes on columns, but adding additional index tables can overcome that. See figure 3.3 for the secondary index table schema. For each index we want to construct on an attribute, we define an additional index table. This index table maps the attribute to the row key of the main log data table. The single column family is named "i" and the qualifier for the single column is named "k".

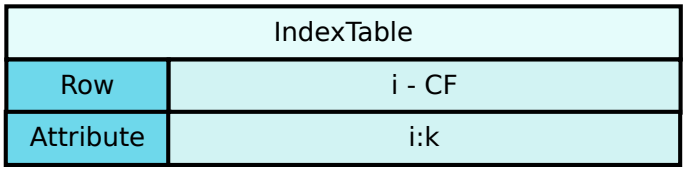

Figure 3.3: Index Table Schema

3.1.3 Full Text Indexing

So far we can access the data through the unique identifier and the attributes which are indexed in the secondary index tables. But the log message, the most interesting part, is not indexed at all. HBase does not offer any searching capabilities on stored values. That is the point where we make us of Solr. The idea is to build a full text index on the log messages using Solr. This index can then be searched and Solr returns a reference back to HBase, which is just the row key to the corresponding log entry. See figure 3.4 for an illustration. Steps one to three describe how to search a value through the full text index. First a query is sent to Solr. Then the result possibly contains row keys which can be used to retrieve the log entries from HBase.

3.1.4 Compression

Because the log entries are usually in text format, it makes sense to apply compression before storing. HBase can be configured to store the data compressed and it is also the recommended way when deploying HBase. Thus for our case it is reasonable to use the built-in compression mechanisms and disclaim any

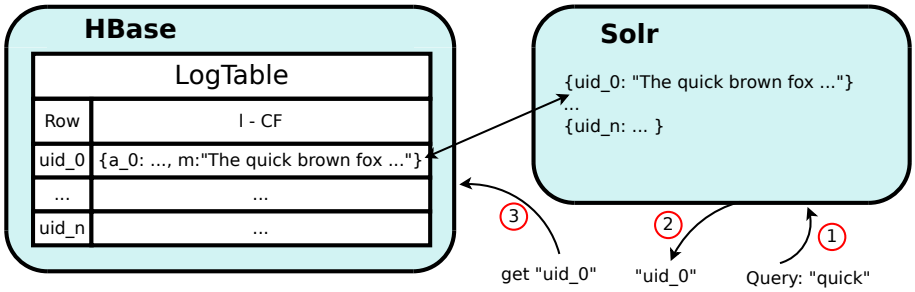

Figure 3.4: Solr Index

custom implementations. For some comments about the applied compression see section 4.5.

3.2 Design Variants

Having the above ideas for the implementation of a database logging system, various techniques are possible. This section shows some variants how the different ideas could be implemented.

3.2.1 Client

There are not many constraints set upon the client. Interaction with HBase can be accomplished in many ways: simple Java clients using the provided client library, Hadoop MapReduce jobs reading and writing data or through Thrift from almost any other language [11].

3.2.2 Secondary Indexes

Basically there are two ways how secondary indexing can be implemented. One way is to let the client do everything. This means next to inserting the log entries into the database, the client also writes the index. The drawback of this solution is that the client is more involved in the indexing process. On the other hand, the client might already have more knowledge about attributes of the log entries and therefore indexing is simpler as the attributes do not have to be extracted once more. The second way to implement secondary indexes in HBase is to make use of the Coprocessor functionality. One would need to configure a Coprocessor to be triggered after each log table insert operation. The Coprocessor could then extract the required attributes from the inserted values and construct the secondary indexes in the index tables. A drawback of the second solution is that the Coprocessor is triggered upon insert, even if there might not be an attribute to index. And if there is an attribute, the value has to be extracted once more.
An issue that arises in both mentioned methods to generate secondary indexes is that HBase does not offer transactions. Therefore we can not guarantee that we always have a consistent state of log entries and secondary indexes as one of the operations could have failed. For some applications it might be tolerable if

the secondary indexes are not fully consistent as long as the data was inserted. A simple approach to solve the issue could be to insert the secondary indexes first and then the log entry. In case something fails, there are just some additional index entries that point to a non existing row in the data table which could be cleaned out by a scheduled job. The concrete implementations in the course of the thesis have not implemented such a failure handling.

3.2.3 Full Text Indexing

Getting the log messages into Solr can be accomplished in multiple ways. As already described in the previous section about secondary indexes, the same techniques could be used for the full text index. Let the client send the log messages to index directly to Solr or use a Coprocessor to extract the messages and send to Solr. Both methods are pushing the data to Solr. Another way could be to let Solr pull the data from HBase. The last method is therefore elegant, as the client and the HBase cluster do not need to have any knowledge about Solr. On the other hand, one has to implement a technique that lets Solr automatically pull the log entries that have just been inserted, so the index is near to real-time. A different approach could be to execute the indexing process to Solr in a more batch like fashion for example using a MapReduce job.

3.2.4 Partial Indexing

Up to now it was assumed that all indexes are eagerly written for all log data inserted into the system. Another option could be to partially index the data at insert time and then later on generate indexes as needed. The benefit of such a method is that a complete index generation at insert time might be too expensive in terms of computational power. Also, an index over the full dataset might not even be required all the time. Thus to allow higher throughput, one could do a partial indexing at insertion, and then lazily index the data as needed afterwards.

3.3 Implementation

One of the main criteria for the success of such a logging database is the throughput. If the systems is not able to scale and to process hundreds of thousands of log entries per second, the goal is not reached. Therefore, for a definite implementation we are looking for the variant that performs best. Which one of the above variants performs best, was not clear at the beginning. Thus multiple variants have been implemented and tested. This section gives an overview over the tested implementations and shows which of them performs better in respect to other variants. Concerning partial indexing, all implementations perform a full indexing on the data because from the use case definition no strategy could be derived how this should actually be implemented.

3.3.1 Client

The client has been implemented using the provided HBase Java client library. The library is easy to use and comes with built-in support for write buffering

and region batching. It has already mechanisms for error handling built-in which facilitates client programming. In addition to the standalone client a MapReduce client implementation has also been done. In terms of performance both methods perform equal, but using MapReduce it is easier to parallelize the insertion so that the HBase cluster is actually fully loaded.

3.3.2 Secondary Indexes

For an evaluation both variants have been implemented. Secondary index creation through the client and through a Coprocessor. As the later variant was not as good as the first, it has been dropped. See appendix B for a comparison between the two.

3.3.3 Full Text Indexing

The connection between HBase and Solr has been the most challenging part. The simple solution to use a Coprocessor to forward the inserted data to Solr over HTTP resulted in very low throughput. There were two main reasons. First the Solr HTTP interface overhead and second the current HBase Coprocessor architecture seems to slow down when larger dependencies are loaded and more involved computations are done. As Solr only provides HTTP interfaces to push data for indexing, we implemented a solution where Solr loads the data directly from the log files or from HBase. This resulted in much better indexing performance but loading the log entries from HBase lead to the next problem: how should Solr know which data to load from HBase. To this problem two different solutions are proposed:

Timestamp based

Each inserted value in HBase is assigned a timestamp. HBase offers a feature to execute scans on tables with a timestamp filter. This allows clients to set a start and an end timestamp and only retrieve the values that lie in that range. Using this feature, Solr servers could continuously scan elapsed time ranges and import the data that got recently inserted into HBase. As the system is distributed there can be multiple Solr server. To make sure that each time range is only full text indexed once, the Solr servers have to share some knowledge about what has already been indexed by one of the servers. As usual, there are multiple ways to solve this problem. We decided to use ZooKeeper. The system already contains a ZooKeeper quorum and ZooKeeper is perfectly fine for coordinating processes of distributed applications [7]. Using ZooKeeper we can have a shared variable over all Solr servers. This shared variable, we call it t_{max}, represents the maximum upper bound of all time ranges that have been indexed by any Solr server. The algorithm that has been implemented to read data from HBase and index in Solr using time range scans can be seen in algorithm 3.1.

ZooKeeper does not support atomic read-modify-write operations nor transactions. Therefore a distributed lock is implemented and shared between all Solr servers. The import routine must first acquire the lock and then it is allowed to read and update the shared variable t_{max}. Having computed a time range, which can not end in the future, the import routine can execute the

Algorithm 3.1 Solr Import

```
1: function IMPORT
2:     while (!importDone) do
3:         lock.acquire()
4:         t_max ← getTMax()
5:         t_start ← t_max
6:         t_end ← t_max + Δt
7:         if t_end > now then
8:             t_end ← now
9:         end if
10:        setTMax(t_end)
11:        lock.release()
12:        scanAndIndex(t_start, t_end)
13:    end while
14: end function
```

scan and index the data. For an example of how a distributed lock can be implemented in ZooKeeper see [7, 12]. This technique makes sure that all data from HBase over a certain time range can be imported and most important it is only indexed by one Solr server.
The proposed timestamp based solution is simple and due to the existing ZooKeeper infrastructure easy to implement. The major drawback is the access pattern of the time range based scan. Assuming that the inserted entries appear in a random fashion, e.g. not ordered by the key, the scan is not able to perform well. As the rows that get scanned are distributed over all regions of the HBase cluster, the scan can not be executed efficiently. To prevent this, one could make sure that the entries are inserted with an ordered key, for example a timestamp. That forces the entries of a given time range to be stored in the same region and therefore can be more efficiently retrieved by a time range scan.

Range/Prefix based

As seen in the previous section, when retrieving data from HBase it is important to have an efficient scan method. The most efficient scan can be done using a row range, e.g. having a start row and an end row or doing a scan using a row prefix. Therefore we also implemented an extension to Solr which executes table scans based on a given range of rows. The problem here is to import ranges, which are not modified anymore. Otherwise the Solr index would miss some data. This could be accomplished by prefixing the row keys in HBase where the prefix is elected based on the current time, for example the current hour of the day. All insertions of that hour would have the same prefix and therefore be ordered together. After that hour has passed, Solr could then efficiently scan all the rows with the prefix of that passed hour of the day. Of course, to have the full text index more up to date, the prefix could be chosen more fine grained. Again using this method, ZooKeeper can be used to have a shared variable which stores the current state of the indexed data.
The proposed prefix solution is more complicated than the previous timestamp based one. We have to think about how the prefix could be determined, also we

have to consider that having a global prefix for a certain duration at insertion time, chances are high that a single region server gets "hot spotted", e.g. all inserts go to the same server. This has to be avoided, otherwise the cluster capacity can not efficiently be used. That could be solved by having a multi-leveled prefix, where the first part is determined by the current time interval and the second is a randomly chosen number between 0 and the maximum number of region servers. Then one could pre-split the tables at creation, so that after start up all regions for all time intervals and for all secondary prefixes are generated. Those regions can then be distributed over all region servers and chances for a "hot spot" are minimized. An illustration of how the data is inserted and how it is retrieved by Solr can be seen in figure 3.5. The illustration shows the state of the current time being in the interval $[t_2, t_3)$ which results in prefix c. Therefore the client inserts row keys which are prefixed by c, followed by a number chosen at random between zero and the number of region servers. The Solr servers continuously scan prefixed ranges which are not used anymore, e.g. where the time interval of the prefix has passed. To coordinate which prefix has been scanned by whom, ZooKeeper could be used again, similar to the timestamp based version.

Discussion

As we have seen the design of the connection between HBase and Solr is not fully clear. Using Solr's HTTP interface is the most convenient and most simple way to implement an automatic full text indexing for the log messages. For requirements with lower throughput in terms of arriving log messages, this method can be preferred. But as already mentioned, for the course of the thesis we try to find a solution which performs as good as possible. Therefore we came up with solutions that let Solr pull the data directly. Looking at the timestamp based and the range based variants, one of the major drawbacks is that for both methods to work well and fast, the row key has to be replaced by a constructed row key. Doing so, we loose some retrieval capabilities, namely we can not retrieve the log messages by their unique identifier anymore. Depending on the use case and on the data, this could be a problem, nevertheless one could just add another secondary index.
For our experiments we mainly wanted to show if Solr's indexing mechanism is able to keep up with HBase. For that reason we implemented two variants where the first reads the compressed log files directly from the local hard drive and the second imports a given row range from HBase. The second method does not work automatically, it is more a prototype to show how the data can be retrieved from HBase and indexed.

3.3.4 Solution Architecture

Figure 3.6 shows the high level setup of the proposed solution. There is a master node which runs all the master services from the different systems: the HMaster (HBase master), the ZooKeeper server and the HDFS namenodes. Then there are multiple slave nodes, where each of them runs: A region server, a Solr server and a datanode from HDFS. Another setup option would be to separate the region servers and the Solr servers to minimize the contention between the two

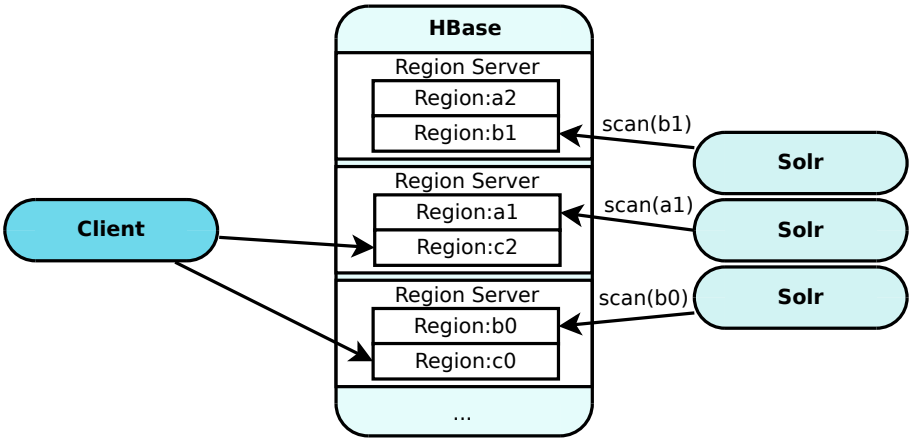

Figure 3.5: Prefix Based Scan

applications. Also in a productive environment one would setup the ZooKeeper cluster on separate nodes. But for non critical experiments we can neglect that fact.

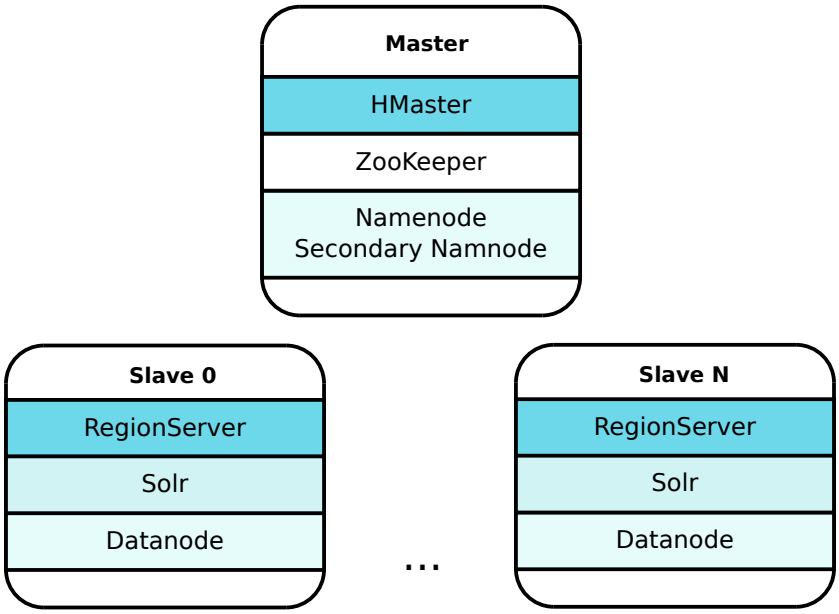

Figure 3.6: Solution Architecture

Chapter 4

Experiments and Measurements

4.1 Setup

4.1.1 Input Data

Amadeus provided us with real log data from their current logging infrastructure. We obtained around 90 GB of compressed data. The data consists of many compressed files between five and ten MB. For testing purposes we divided the whole dataset randomly into splits of five GB. The data is compressed by a factor of roughly seven, this results in a total data size of 630 GB. In all the experiments the mentioned data size refers to the compressed data.

Here is an example of a typical log entry (the message is not shown):

```
2012/05/15 04:32:42.274686 sitst201 srvT2M-838059 Trace name: all0302
Message sent [con=3428618 (FE_EXT_TCIL-I9735_PPPK-005_PPPK-REQ),
cxn=1498681843 (172.31.51.5:21100), addr=0x1db583a8, len=1336,
CorrID=58616D59, MsgID=E#HM7L5ON38KGOFI9Q81O8]
[...]
```

Listing 4.1: Example Log Header

The log entries have an average size of one kilobyte and in general have the following structure:

- The first line always starts with a timestamp
- The second line contains message attributes, like type of message and various identifiers.
- The next n lines contain the actual message.
- An empty line separates single log entries.

The custom parser that has been implemented reads a single log entry and separates it in various attributes and the log message. In listing 4.1 we see for example attributes like: *con*, *cxn*, *addr*, *len*, *CorrId*, *MsgID*. Each attribute is

stored in its own column. As unique row key, a concatenation of *MsgID* and the timestamp extracted from the first line is used.
The attributes have also been used to construct a secondary index. If nothing else is stated, each experiment was configured to construct four additional indexes.

4.1.2 System Configuration

All experiments were run on a cluster at the Systems Group of ETH Zürich. The machines had the following configuration:

- 2 QuadCore AMD Opteron 2376 2.3 GHz Processor
- 16 GB DDR2 667 MHz ECC Memory
- 1 TB S-ATA II 7200rpm Disk Drive
- 10/100/1000 Gigabit Ethernet

The system has been implemented using the following software versions:

- Hadoop 1.0.2
- HBase 0.94.0
- Solr 4.0 Beta

4.1.3 Client

For insertion a client has been implemented that reads the compressed log files, decompresses the archives on the fly, parses the log file and submits the log entries to HBase. The client is configurable for various run modes like batching, caching and region batching and is highly multithreaded. Usually the client has been deployed on multiple separate client nodes to make sure that the system can be saturated.
A MapReduce client has also been implemented reading the log files from HDFS and inserting the log entries into HBase. In terms of performance both implementations behave the same, except that it its easier to fully load the cluster using the MapReduce client, because it runs automatically distributed and in parallel. For the experiments the standalone client has been used, because the MapReduce client requires to have the MapReduce infrastructure running next to HDFS and HBase. This was not practical on the given hardware as it could have stressed the single nodes to much.

4.2 Insertion

An important criteria for the final system is the ability to sustain high loads of arriving log messages. Therefore we focused on insertion and indexing performance when implementing the different components. The following experiments uses different datasets and also varies the number of machines. This should give us an idea, how HBase and Solr can cope with increasing data and how the overall performance can be increased by adding more machines.

Concerning the datasets a 20 GB, a 40 GB and an 80 GB dataset were used. The number of machines was varied between four, eight and twelve. Besides that the experiment consists of three different measurements. The first is the time used to insert the data into HBase, the second is the time used to pull the whole dataset from HBase to Solr and the the third is the time used to load the log entries directly from the files into Solr. The reason to have two measurements for Solr is that loading the log entries directly from files performed best and could therefore be seen as the upper limit for the Solr indexing throughput for the given environment.

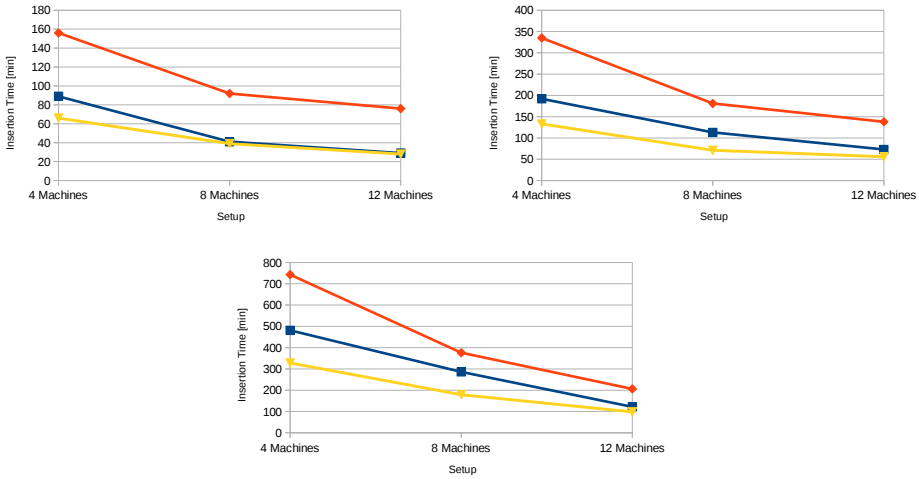

Figure 4.1: Insertion: HBase (blue □), Solr from HBase (red ⋄), Solr from File (yellow ▽)

Figure 4.1 shows the times used for insertion for the three different datasets over the varying number of machines. The results show that the insertion time decreases mostly linear when increasing the number of machines. In the case where Solr reads directly from the files, this is obvious as the input is just evenly split in front between multiple instances. Another finding is that Solr reading directly from the files is comparable to HBase and even faster at four and eight machines.

To have a closer look on how many logs per second can be indexed on the given configuration figure 4.2 shows the number of log entries that are inserted per second per machine over all the different setups. In general, the numbers are in the same range for a specific system and do not vary to much changing the number of machines or the dataset size. In the case of HBase it can be observed that adding more machines lowers the throughput per machine. This could be explained by the effort that is needed for the distribution or by some imbalance between the region servers, so that the additional resources can not be used perfectly. This also coincides with findings from [2] where it is stated that there is a significant drop in per-server throughput when going from 1 to 50 servers. Looking at the HBase results for the 80 GB dataset we see that there is a significant increase of the throughput on twelve machines compared

to eight or four, which is in conflict with the previous statement. The effective throughput for 80 GB at twelve machines is still in line with the other results, therefore it does not necessarily have to be an outlier. An explanation could be that dividing the dataset over twelve regions servers positively affected the throughput more than having an increased number of machines.

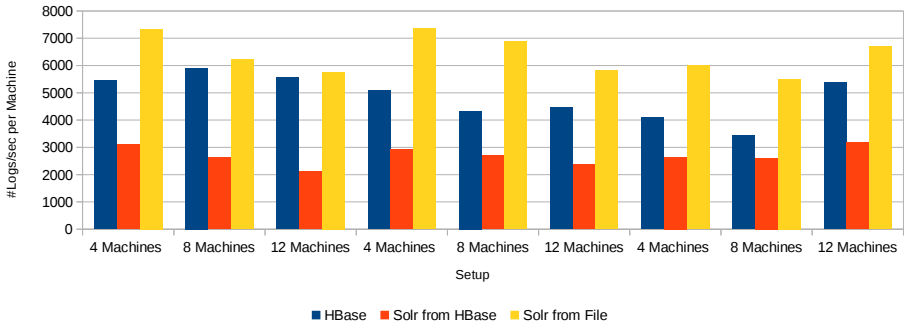

Figure 4.2: Logs per Node

4.3 Retrieval

Amadeus provided us with real world data but unfortunately it was not possible to get any use cases for queries during the course of the thesis. We had to came up with different types of queries and ran those on the same configurations as the insertion experiment, namely using a 20 GB, a 40 GB or an 80 GB dataset on four, eight or twelve machines. The following queries have been implemented and measured:

- *Query 1: retrieve by key*
  Get the full log message from HBase using the row key.
- *Query 2: retrieve by secondary index*
  Look up log messages using a secondary index.
- *Query 3: range scan on index table*
  Scan a given range on an index table, and look up all associated log messages.

All the queries in this section focus on the data retrieval capabilities of HBase. For queries that also include Solr see the Splunk comparison in section 4.6.

Before the results are presented, some comments to the measurements. We tried to execute the queries directly after the insertion process finished to make sure that the cluster is always in a similar state. Then, when we repeated the experiments, we made sure to clean out the caches first to have similar conditions across the measurements.

In figure 4.3 we see the results for the first query. Each line represents the query

times including standard deviation for a specific dataset size over the different number of machines. The query consists of multiple get requests whereas the row keys were randomly chosen from the dataset. Therefore, the single requests should not profit from any caching effects. In the result graph one can see that the measured query times are quite close together and changing the number of machines or increasing the dataset size does not have a big influence on the execution time. Comparing the 80 GB dataset to 40 GB and 20 GB at twelve machines, the query time is not decreasing. An explanation for this behavior could be that in the 20 GB and 40 GB case, the systems benefits from distributing the data over more machines, whereas this effect is not noticeable in the 80 GB case. Unfortunately we could not perform the experiment on more machines to see if this explanation holds.

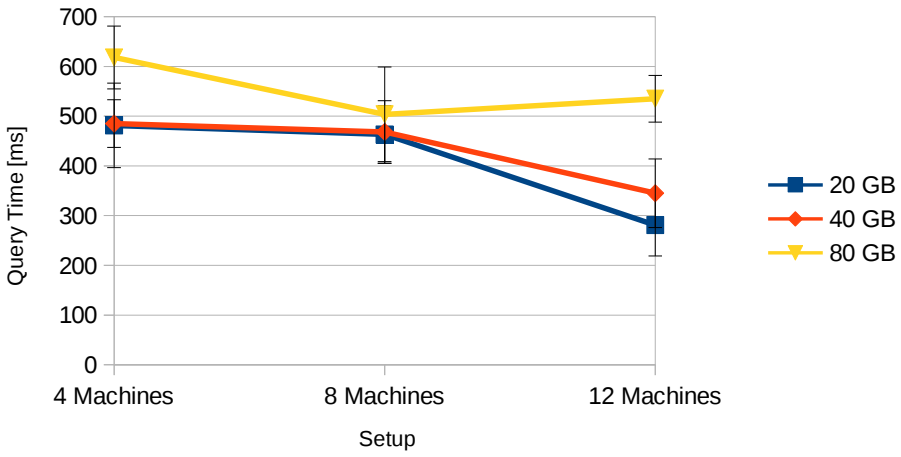

Figure 4.3: Query 1

The results for query 2 are shown in figure 4.4. The second query looks up the log message row keys using the different secondary index tables. As the secondary index might map to multiple row keys in the log table and in general more rows are requested from the log table, the caching mechanisms of HBase have more effect on this query, therefore the standard deviation is higher than in the first query. The measured time includes the request on the index table and the requests on the log table to retrieve the log entries. The results in the graph are as expected and similar to the first query. It is to note that the look-up on the secondary index table is usually faster because the index tables can be smaller and contain much less data, e.g. only two columns.

Figure 4.5 plots the results for the third query. This query performs a range scan on a secondary index table, namely on the timestamp index table. The scan uses a prefix filter and is configured to scan a certain time range. For all row keys that are retrieved scanning the timestamp index table a look-up is done on the log data table. For the experiment the scan was configured to scan a time range of one second. This returned around 2500 results which were

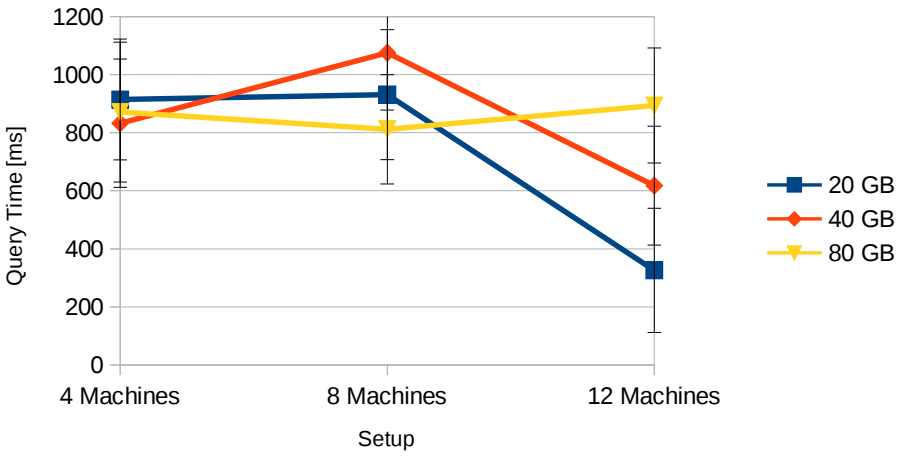

Figure 4.4: Query 2

looked up in the log data table. The data shown in the graph is the average look-up time for a single result from the scan. In this case a timestamp is mostly unique and therefore a look-up consisted of two requests. One from the scan and one to load the data from the log data table. Compared to the previous queries the average look-up time seems to be quite low. First we have to note that the scan on the timestamp table is extremely fast compared to single retrievals. Once the first value has been retrieved, the others come almost for free due to the scan cache which transmits multiple rows in a single remote procedure call. Second, as we look up many values in the log data table, chances are higher that we again hit the caches in the region servers. Between the three datasets there is again no big difference in the measurements so that we can conclude that HBase can scale well over increased data size without loosing of its retrieval performance.

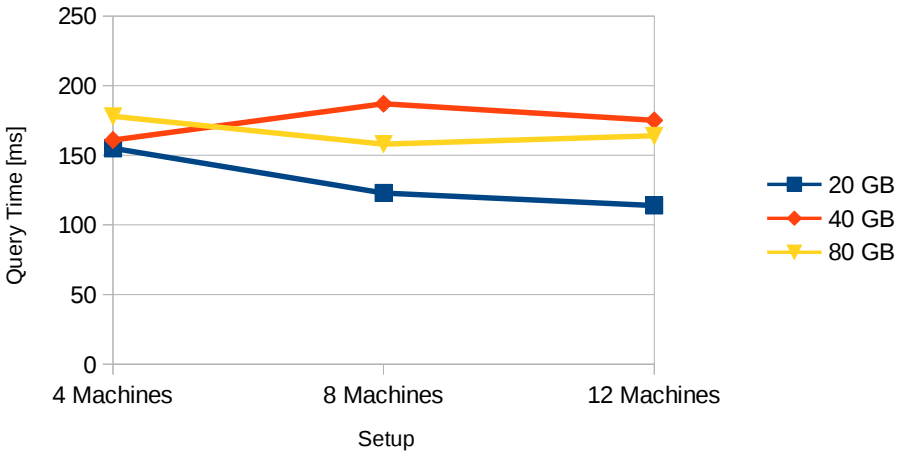

Figure 4.5: Query 3

4.4 MapReduce

Because HBase runs on top of HDFS it is obvious that is comes with support for MapReduce. We implemented a simple MapReduce program which does a table scan over the whole log data table and reads out two attributes, namely *con* and *addr*. In the end the program counts the occurrences of *con* and *addr* pairs. The mapper simply extracts both attributes and emits them to the reducer as a concatenated string. The reducer which receives all single occurrences just sums all up and writes the final sum to the result output. To have a comparison, we also implemented the same logic in a MapReduce program that directly reads the compressed log files from HDFS. The two variants were run on a five node setup, one master machine and four slave machines. The input datasets were 20 GB, 40 GB and 80 GB in size. The results of both variants are presented in figure 4.6 which shows the time needed for execution for the three different datasets.

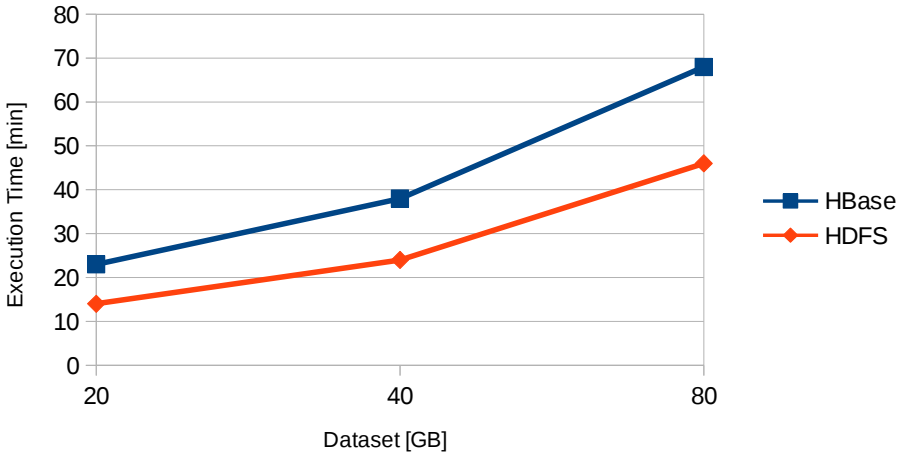

Figure 4.6: MapReduce

As expected, for all inputs the HDFS implementation is faster as this is a very typical use case of batch processing where a standard MapReduce implementation usually excels. A specific reason for the difference is that executing the MapReduce job over the log files on HDFS the cluster can be saturated very well as many input splits exist. In the case of HBase the input is split in regions but the number of regions for all input datasets has only been around 20 to 40. Therefore the job consists of large, single tasks and the work can not be distributed over all worker nodes equally, resulting in less utilization on certain nodes. Especially in the end of the job, we observed that only few task were running on one or two of the nodes and the other nodes were not used anymore. This issue could be overcome by splitting the regions manually before running the job so that the data is distributed over more regions of equal size. This becomes even more important when having a larger cluster with more region servers.

4.5 Secondary Indexing

As secondary indexes are not a default feature of HBase and therefore manually implemented, we wanted to find out how insertion performance is affected by the additional operations that are needed. The experiment lets the client create four secondary indexes which results in having four additional tables for the indexes. The HBase cluster was deployed on five machines, one master node and four region server nodes. Two different datasets have been inserted, one with 10 GB and another with 20 GB in size. The first result set compares the insertion rate of the client with and without secondary indexes. The second result set compares the total disk space that is used by HDFS in both modes. The HDFS cluster has been configured to replicate the data by a factor of two.

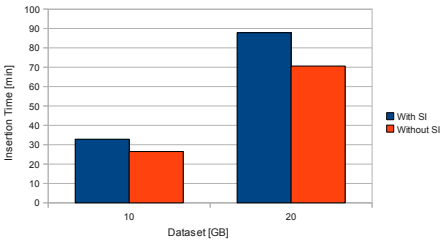

Figure 4.7: Insertion Time

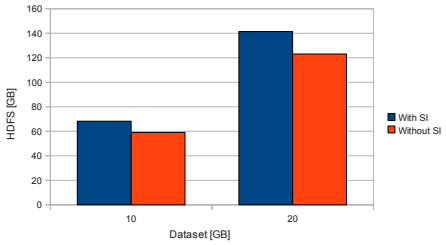

Figure 4.8: HDFS Disk Space

For four secondary indexes on the given data, the insertion time increases by 24% on average (see figure 4.7). The HDFS disk space usage is increased by 15% in average (see figure 4.8). Of course this highly depends on the data and the values chosen for the secondary index. Nevertheless regarding the added retrieval capabilities, it shows that for the given data additional indexes are a worthwhile option.
Note that the source dataset is compressed using *gzip* which uses the *Deflate* compression algorithm. We configured HBase to use *Snappy* [5] as compression algorithm. *Snappy* is a compression/decompression library that aims for very high speed at reasonable compression. In contrast to *Deflate* which aims for maximum compression. Therefore the space used on HDFS is at least twice the size of the input data, because of a replication factor of two. It turned out, that *Snappy* compresses the log data by a factory between three and four whereas the original data is compressed by a factor of roughly seven.

4.6 Comparison with Splunk

During the course of this thesis, Georg Polzer, student at the Systems Group of ETH Zürich, evaluated the commercial log analysis platform Splunk for his Master thesis. Therefore we decided to carry out a comparison between the two systems to get an idea, how the system performs. The benchmarks consists of two parts: indexing performance and data retrieval. For a qualitative comparison see appendix C.
Both parts of the benchmark are based on two factors. First factor is the number of machines where three levels are considered: One machine, four machines

and an eight machine setup. In the case of the HBase/Solr setup, one additional machine is added as the master node. The second factor is the dataset size. Here we used a 20 GB, a 40 GB and an 80 GB dataset.
As the two systems are quite different in terms of the architecture, different setups of the HBase/Solr system have been deployed for the indexing part of the benchmark. The first setup was HBase only where the data is inserted and indexed in HBase. The second setup, which comes closest to Splunk, lets Solr read the log files directly from disk. This means the log files where distributed over the machines and each Solr instance indexed a subset of the dataset. In this case, the data is loaded and the index is written to the same disk. This is also the setup used for Splunk. The third setup lets Solr read and index the log data from HBase.
Note that for the single machine, 80 GB dataset experiment no results are presented because it took exceptionally long to run. This could be an indication that for the given hardware the 80 GB is an upper limit on what can be indexed on a single machine.

4.6.1 Indexing

The first experiment compares the indexing performance between the different setups and systems. To give a good impression how the systems scale out, each figure in 4.9 shows the time used for indexing for a fixed dataset over one, four and eight machines.

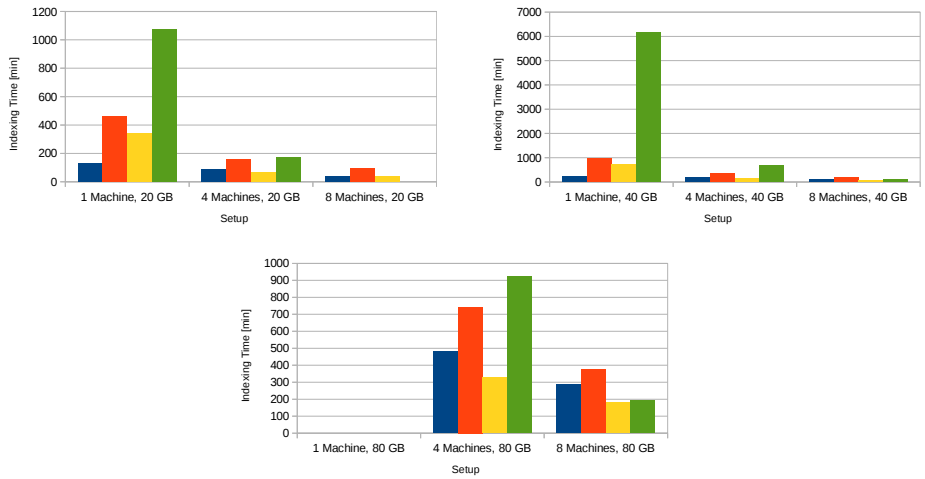

Figure 4.9: Indexing comparison: HBase Only (blue), Solr from HBase (red), Solr from File (yellow), Splunk (green)

Except for the single machine setup, one can see that the different setups and systems are in the same range in terms of indexing performance. Also, all setups scale well over an increasing number of machines. Comparing the HBase/Solr implementation to Splunk, it seems that Splunk does not scale very well over the dataset size, meaning that given a data size more than 10 GB per machine, Splunk needs much more time for indexing.

Comparing the Solr from File setup to Splunk, we can see that both systems are almost equal, except that Solr seems to scale better given a data size more than 10 GB per machine. Nevertheless, as it is not exactly known how the internals of Splunk work and what exactly Splunk does during the indexing period, it is nearly impossible to draw a conclusion. To have a more precise comparison between the two, we would need a more precise use case definition. The HBase only setup compared to Splunk is usually in the same time range. But with increased dataset size and more machines, HBase performs slower. This could be explained by the overhead introduced by the distribution and also by the data replication that is configured by the underlying HDFS file system. Since Splunk reads the already distributed data files directly from disk, this is a clear advantage.

When performing the experiments we also measured the used storage space that is needed by the different setups. The results can be seen in figure 4.10. Each plot shows the needed storage for a fixed number of machines over the three different datasets. For HBase the numbers shown measure the used space with no replication.

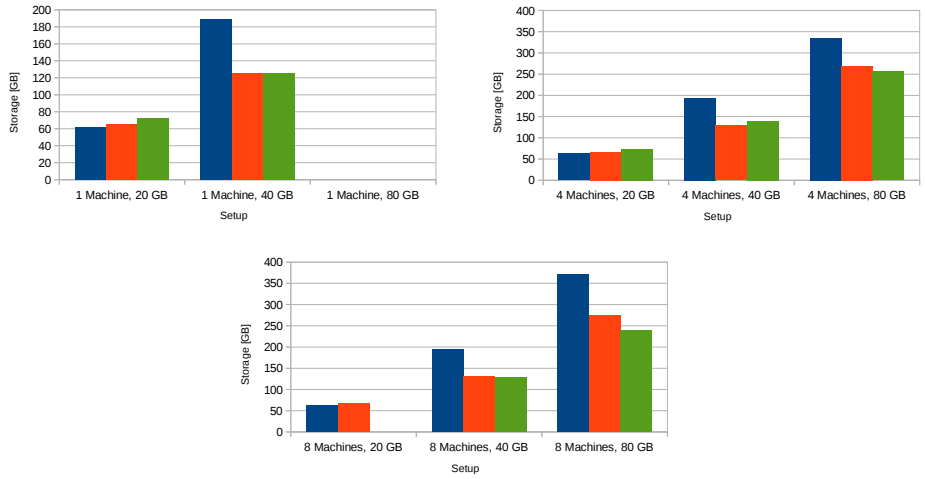

Figure 4.10: Storage comparison: HBase Only (blue), Solr (red), Splunk (green)

One has to note, that if HBase and Solr are deployed together, the numbers of HBase and Solr shown in the graph have to be added up. Comparing that to Splunk, the needed capacity is twice as big. Besides that, we can see that the needed disk space by both system is in the same range and seems to scale almost linearly over the dataset size. However, the numbers of HBase and Solr have to be taken with care. As explained in chapter 2, HBase performs compaction tasks. These tasks usually decrease the size of the needed storage, as multiple files are merged together. Therefore the numbers measured can vary depending on the state of the system. The same holds for Solr. Solr has an option to *optimize* the index. This option also merges the index files together and usually results in less disk space usage. For the experiments, we tried to perform the measurements directly after the insertion procedure to have comparable measurements.

4.6.2 Retrieval

In the second part of the benchmark we compared the retrieval capabilities of both systems. As Solr is comparable to Splunk in terms of architecture, setup and capabilities, the following experiments compare only Solr and Splunk. As we were not provided with real-world queries by Amadeus, we came up with four different queries, ranging from full-text search to more complex queries using some of the query language features of both systems:

- *Query 1: full-text search*
  Run a full text search over all indexed log messages searching for the string "0742489654788".
- *Query 2: indexed field search*
  Search for the string "CAYKK1MPFX3Y1U1AIFJBQJ" in the indexed field "MsgID".
- *Query 3: wildcard search*
  Do a wildcard search in the indexed field "cxn" for the string "*172.24.37.116:9001*".
- *Query 4: numeric range search*
  Search for all log entries where the attribute "len" is in the range of 13'000 and 13'100.

Before presenting the results, we have to consider some limitations of the benchmark. Solr's web interface which was used for executing the queries usually returned only the first 10 results. Trying to retrieve the full result set timed out on longer running queries. Therefore the results shown for Solr usually measure the time to retrieve the first 1000 results. Another point is that the Splunk retrieval engine is not publicly disclosed and therefore not all results can be fully explained.

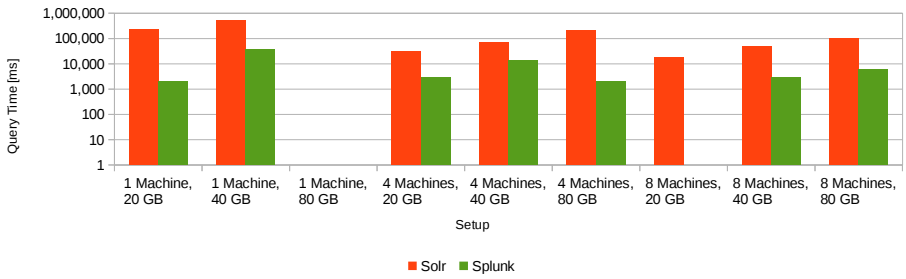

Figure 4.11: Query 1

Figure 4.11 shows the results for the first query over all different setups. Because of the large variance in the results, the search time is shown at logarithmic scale. It is clearly visible that Splunk outperforms Solr in each setup. Splunk returns the results faster than Solr by a factor of 10 to 100. The reason lies in the tokenization mechanism of the two systems. Solr's default tokenizer does not handle most of the log messages correctly and therefore query 1 had to

be executed as a wildcard search to retrieve any results and wildcard searches are usually much more costly. Splunk in contrast seems to handle the log messages very well and does a good job on tokenization, such that no wildcard search is needed.
In the four machines, 80 GB dataset setup, the result for Splunk is probably a measurement error, as it is lower than the four machines, 40 GB setup and also lower as the eight machines, 40 GB setup and thus should be discarded.

The results of query 2 are shown in figure 4.12. Again, the results are shown at logarithmic scale. This time the difference between Solr and Splunk is not as big, but Solr returns the results up to 10 times faster than Splunk in every setup. The conclusion that can be drawn from this experiment is that Solr seems to perform very well on indexed fields, where no wildcard search is needed. On the other hands it is impossible to reason about Splunk, as the internals are not known.

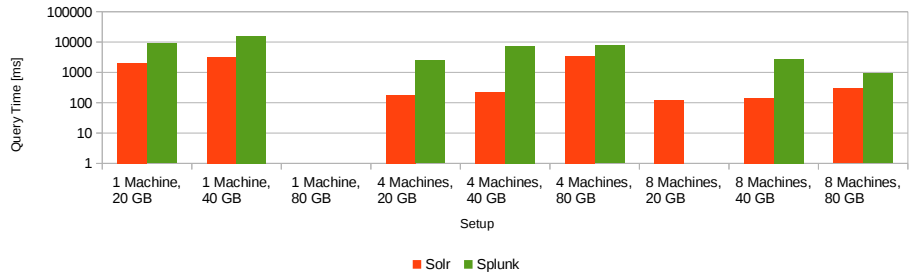

Figure 4.12: Query 2

The results for query 3 and query 4 are shown in figure 4.13 and 4.14 respectively at logarithmic scale. Note that the measured times are in seconds. As mentioned, the retrieval capabilities for Solr are limited using the built-in web interface. Therefore the results shown for Solr are the times needed to retrieve the first 1000 results.

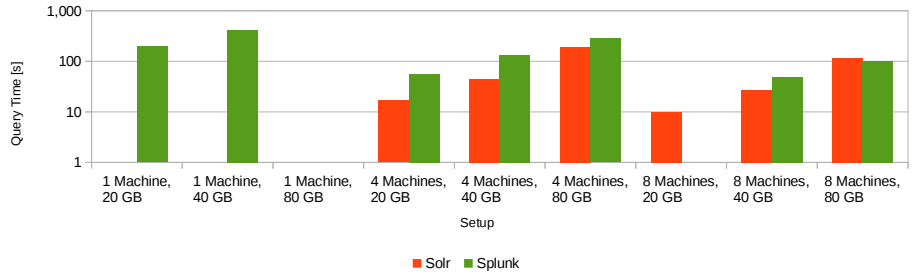

Figure 4.13: Query 3

From the shown results it is not possible to draw a conclusion about the differences between the two systems. What we can see is that both systems seem to behave as expected when increasing the data size or the number of

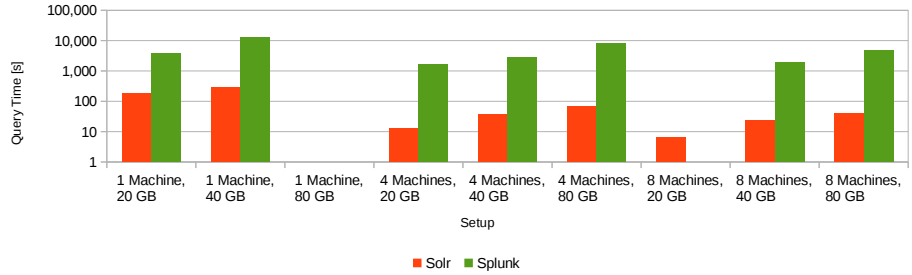

Figure 4.14: Query 4

machines. Again, in the case of Splunk it seems that there is a non proportional penalty on the query time when indexing more than 10 GB of data per machine.

4.6.3 Conclusion from the Comparison

The analysis consisted of two parts: indexing and retrieval. For the indexing part we could show that HBase and Solr, in combination or as a single setup, are comparable to Splunk. All of the systems are able to index a data volume or around half a terabyte per day, given the mentioned hardware. Looking at the search performance it is not so easy to give a final conclusion, at some queries Splunk outperformed Solr, at other queries vice versa.
Looking at the Amadeus use case, where multiple terabytes of log data per day are collected, it is not possible to directly derive a conclusion which of the platforms should be preferred from our experiments, because the maximum tested data sizes were only around half a terabyte. Also regarding the retrieval capabilities we had to come up with our own queries, which might not reflect the use cases at Amadeus. But focusing on the indexing of these massive amounts of data, HBase could be elected over Splunk as it has been proved to scale well over large amounts of data, for example at Facebook's messaging platform [8]. Moreover, looking at the qualitative comparison, we see some advantages of HBase and Solr over Splunk, for example data replication, which is crucial in a productive environment. Another decision criteria could be the licensing model of Splunk, whereas HBase and Solr are both open-source.
Therefore, for a more meaningful comparison the use case must be defined more precise and the experiments should be scaled out to larger amounts of data.

4.7 Disk Drives

The general hardware recommendations for HBase propose to have one disk per core at the region servers [4]. For our experiments we were limited to servers which had only one disk and therefore could not fully confirm that. Nevertheless it is clear that for high write loads, the region servers need to have fast disks. To show that in our setup the disks were a limiting factor we captured the I/O wait, e.g. the time the CPU is waiting for the disk, during the time of a write intensive experiment. In the following test run, we setup an HBase cluster with five nodes, one master node and four region server nodes. There were four separate client nodes, each of them inserting a 20 GB dataset, thus

in total 80 GB were inserted.
Note that all of the following three figures that are presented for this experiment show the same time range, more precise a 24 hour interval is shown.

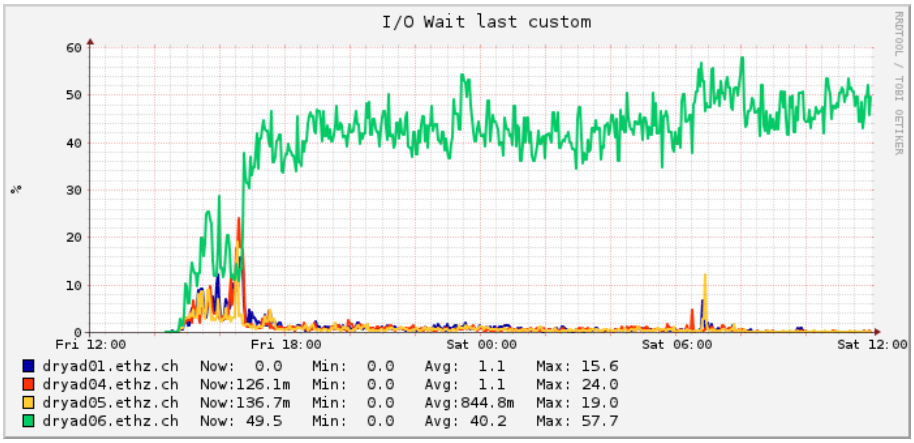

Figure 4.15: I/O Wait - Region Server

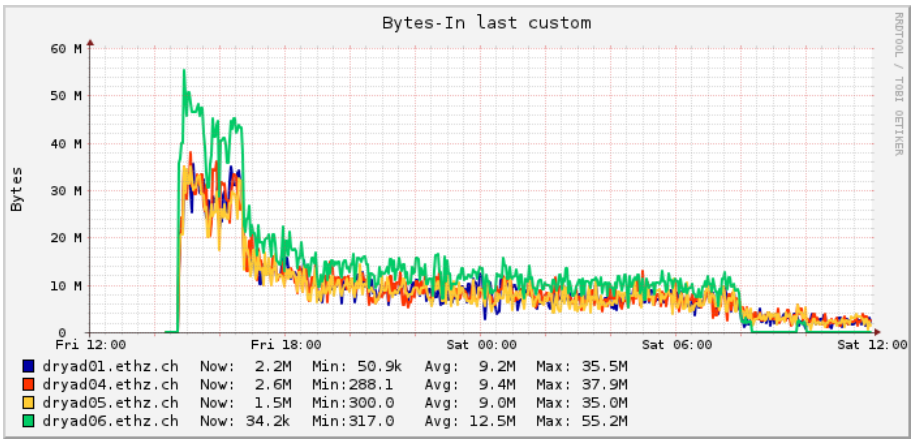

Figure 4.16: Bytes-In - Region Server

Figure 4.15 shows the percentaged CPU time spent on waiting for disk I/O. As the experiment continues, one of the nodes suddenly has an increased percentaged time spend on waiting. In figure 4.16 the incoming network traffic is measured for each node over the same period of time. We can see that the more time the CPU spends waiting on the disk, the lower the throughput. On client side we often observed timeouts during our experiments. Moreover, when we tried to saturate the system as good as possible by increasing the number of client threads, we had to be really careful to not overload the system. Otherwise the region server started performing really bad and the clients timed out even more. This situation should be avoided in a system that has a constant stream of incoming data.

To have a closer look what is happening on the region server that gets overloaded, we captured the size of the compaction queue during the insertion process (see figure 4.17). When new compactions are triggered, in this case minor compactions, and there is still another compaction running, they are put into a queue. The figure illustrates that the queue grows as long as the insertion is running and therefore clearly shows that the region server cannot keep up with the compactions, which merge files together and perform many I/O operations on the disk. As the disk is heavily utilized by the compactions, the flushing process from the memory store to the disk gets also slowed down and therefore influences the insertion rate directly. After a certain threshold is reached, where the memory is full and data is waiting to be flushed to disk, the region server blocks the clients until the data could be flushed to disk. If the blocking takes too long, the clients receive a timeout.

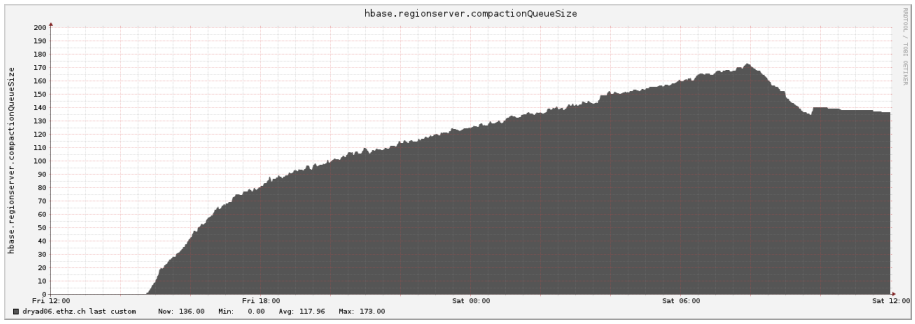

Figure 4.17: Compaction Queue - Region Server

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis we propose and implement a system to store large amounts of log data and have presented various benchmarks showing advantages and also limitations. The design section shows multiple ideas and concepts how the system consisting of HBase and Solr could be designed and implemented. Additionally a comparison with the commercial log analysis platform Splunk has been conducted which also shows interesting insights.
Overall we can say that the goal to have a database logging system able to handle high loads has been accomplished. Nevertheless the solution explored using HBase and Solr does not have to be fixed. Especially the connection between the two can be seen as the most unstable part and has not been fully solved. Besides that, the thesis can also be seen as a source for HBase applications as multiple findings have been presented which could be used in a different context as well.

Regarding the Amadeus use case the proposed system has been tested using data directly from their current systems. It could be shown that the system scales over increasing data size and the number of machines. The system was able to process around 5000 log entries per second per node. This seems to be far away from the desired hundreds of thousands of logs per second but considering the given hardware and the findings from the experiment section, the system proofed to be a candidate for a solution at Amadeus. If Solr is part of that solution has to be discussed but HBase seems to be a good choice as the core of the storage platform. In terms of the retrieval capabilities we do not draw any conclusion as the use case was not fully specified and multiple assumptions have been made for the benchmarks.

The comparison with Splunk showed that a combination of HBase and Solr could compete with commercial tools and even offers clear advantages. Of course those advantages come with more effort that is needed for certain tasks. We could also see the current state of a log analysis platform and where it excels for example at automatic parsing and indexing and at visualization.

All in all the thesis is a starting point for a database logging system based on HBase and in the future work section multiple ideas are listed for other design variants and components that could be considered for a future implementation.

5.2 Future Work

5.2.1 HBase

For the thesis HBase has been elected as the main storage system and has been proved to work well. However it could be worth to evaluate other systems like Cassandra or Hypertable. Hypertable for example claims to have really high performance and also works on top of HDFS.

5.2.2 Solr

As mentioned the connection to Solr is still unfinished work. Besides the variants explained in the thesis another option could be to have a queue which acts as a connection between HBase and Solr. As queue any existing high performance queuing system could be used or one could take HBase to build a queue on top of it. Whether such a solution is practical has to be found out but there already exist queue implementations on top of HBase[1].
During the beginning of the thesis we decided to use Solr as the component to build a large additional index for the unstructured part of the data. But choosing Solr also introduced some limitations. For example Solr's architecture is mainly built around the web server container and therefore all interaction with Solr is done through HTTP. For many applications this works well, for our case it did not help too much. Choosing Lucene instead of Solr we could build the full text index and do not need to care about how we can connect to the external Solr system. Working with Lucene the following ideas could be investigated:

Fully integrated: The Lucene library could directly be integrated in to the existing region servers. On client side we could then specify on which columns or even column families to have a full text index. The index itself could for a simple implementation be stored on the local disk drive. In a more advanced implementation the Lucene index files could be stored on HDFS such that the Lucene index is distributed, replicated and that the index is also where the HBase data resides. However it could be quite challenging to have the Lucene index on HDFS and still achieve good performance because the access patterns of HDFS are different and the file formats of Lucene might need to be adapted to that first.

Coprocessor: The Coprocessor framework is a great way to add customization to the region servers without changing the existing code base. Therefore adding the additional full text index through a Coprocessor is an elegant solution. In this thesis the Coprocessors implemented could not fully convince us, nevertheless this could be more investigated. Also one could have a look at

[1] `http://www.lilyproject.org/lily/about/playground/hbaserowlog.html`

an alternative Coprocessor implementation, where the Coprocessor is hosted on a separate process, in other words how it has been proposed by Google for Bigtable[2]. This could give more control over the resource consumption and a heavier Coprocessor implementation like a full text indexing Lucene Coprocessor could benefit from that. And of course, the points how to store the Lucene index also hold for this variant.

[2] `https://issues.apache.org/jira/browse/HBASE-4047`

Appendix A

Example Logs

```
2012/05/15 04:32:42.215725 sitmt301 muxT2-332108 Trace name:
all0302
Message sent [con=266996 (SBOOBEinMux2), cxn=1631331361
(172.17.39.33:58806), addr=0x1220a830, len=156,
CorrID=0001001UZCXKV7, MsgID=GTH95Q4G99YF2JE6XMU2O8]
UNB+IATB1+1ASRPAP+1A0SBR+120515:0432+00BEAT0001+001UZCXKV70001++T'
UNH+1+PCCHRR:10:1:1A+0001001UZCXKV7'
RCI+:YUDN8L'
PTD++14'
UNT+4+1'
UNZ+1+00BE0H3ZAT0001'
```

Listing A.1: Example Log 1

```
2012/05/15 04:32:42.274686 sitst201 srvT2M-838059 Trace name:
all0302
Message sent [con=3428618 (FE_EXT_TCIL-I9735_PPPK-005_PPPK-REQ),
cxn=1498681843 (172.31.51.5:21100), addr=0x1db583a8, len=1336,
CorrID=58616D59, MsgID=E#HM7L5ON38KG0FI9Q81O8]
+----- ADDR -----+--------------- HEX ---------------+-----
ASCII ----+---- EBCDIC ----+
 000000001db583a8 554e421d 49415442 1f311d31 4153491d
UNB.IATB.1.1ASI. .+.............
 000000001db583b8 4e494e43 50524943 494e474b 1d313230
NINCPRICINGK.120 +.+.&....+......
 000000001db583c8 3531351f 30343332 1d303050 43594742
515.0432.00PCYGB ...........&....
 000000001db583d8 57384830 3030311d 1d1d4f1c 554e481d
W8H0001...O.UNH. ..........|..+..
 000000001db583e8 311d5452 4d524551 1f30351f 311f3141
1.TRMREQ.05.1.1A ....(...........

[...]
```

Listing A.2: Example Log 2

```
2012/05/23 07:08:09.056720 siwsp102 srvWS1M-115669 Trace name:
MSG4k
Message received [con=3171248 (inSrvWS1_XMLv2), cxn=1396874017
(172.24.36.97:58184), addr=0x1c7b47830, len=944,
CorrID=0001P3b5eC1, MsgID=R$ERANT3OW54WYMNAPAI8O]
<?xml version="1.0" encoding="UTF-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header
xmlns="http://www.amadeus.net/1axml-msg/schema/msg-header-1_0.xsd">

[...]
```

Listing A.3: Example Log 3

Appendix B

Secondary Index Comparison

The following experiment compares secondary index creation using a Coprocessor against direct index creation through the client. The test was performed on a five node HBase setup: one master node and four region server nodes. As input data, log data from Amadeus was used. Two datasets were used, one with 10 GB and one with 20 GB in size. Figure B.1 shows the insertion time for both datasets.

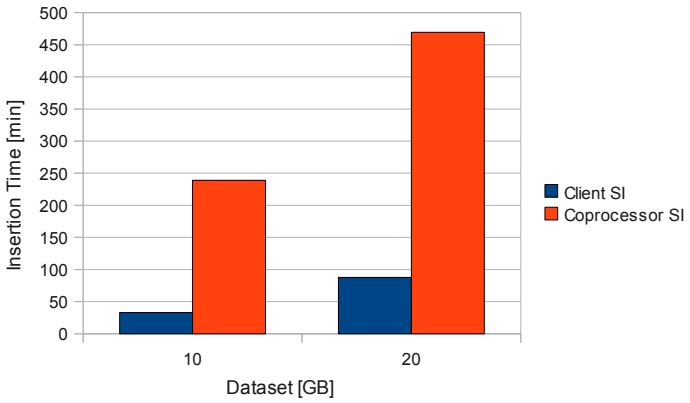

Figure B.1: Client vs. Coprocessor

It is clearly visible that secondary index creation through the Coprocessor achieves much lower throughput. The reason for such a big difference could not be fully clarified. But on one hand, the Coprocessor is executed for each insert operation that is performed by the client. Because not all log entries come with all attributes, for some insertions a secondary index can not be written. Nevertheless, the Coprocessor is executed.
Performing secondary indexing on client side, the client will not carry out the operation, if the secondary index is empty. On the other hand, depending where the index table regions reside it could happen that having the index insertion distributed over the coprocessors results in a less effective usage of operation batching. Executing the index inserts on client side we make use of region batching. This collects multiple insertions that affect the same region

together and executes them in a single remote procedure call.
Finally, Coprocessors are a slightly new feature of HBase and we could also observe a similar performance decrease when indexing data to Solr using the Coprocessors. However it remains to investigate this issue for example by profiling the region server.

Appendix C

HBase/Solr - Splunk - Qualitative Comparison

Criteria	**HBase/Solr**	**Splunk**
Indexing		
Effort for loading data into index	High. Manual parsing, custom client for HBase, multiple ways to index data in Solr. All schemas (HBase tables, Solr document fields) have to be defined.	Very low, convenient web interface.
Effort for field extraction and field indexing	Medium to high. Solr has a web interface for querying. HBase only a command line shell. Need for custom client implementation.	Very low, convenient web interface.
Data export interfaces	Many interfaces exist. HBase can be accessed by almost any language using Thrift. MapReduce can directly read the data. Solr is mainly accessible by HTTP. HBase and Solr are both open-source.	Manual export into CSV.
Redundancy of index	HBase operates on top of HDFS, which is replicated by default. Solr offers master/slave architecture, so that the index can be replicated.	No native way of index replication.
Continued on next page		

Criteria	**HBase/Solr**	**Splunk**
Searching		
Ease/power of query language	Interactions with HBase have to implemented based on the supported CRUD operations. Solr offers query functionality and advanced query language. Highly extensible.	Supports filtering, aggregations.
Result presentation	No visualization	Powerful visualization and plotting tools
Drill down capabilities	Some functionality supported by Solr (faceting)	By simple click, very intuitive.
Complex computations		
Distributed computations	Can be done using MapReduce or Coprocessors.	Aggregation functions are distributed as MapReduce. KMeans does not scale linearly.
Complexity and extendability	Almost anything is possible. Highly extendable.	Not too complex and not extendable.

Table C.1: Qualitative Comparison HBase/Solr - Splunk

Bibliography

[1] Matteo Bertozzi. HBase I/O - HFile. `http://www.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/`, September 2012.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[3] Jeff Dean. Google - Designs, Lessons and Advice from Building Large Distributed Systems. Presented at Ladis 2009, 2009.

[4] L. George. *HBase: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2011.

[5] Google. Snappy. `http://code.google.com/p/snappy/`, September 2012.

[6] Apache HBase. Apache HBase. `http://hbase.apache.org/`, September 2012.

[7] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[8] Kannan Muthukkaruppan. The Underlying Technology of Messages. `http://www.facebook.com/note.php?note_id=454991608919#`, September 2012.

[9] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[10] Apache Solr. Apache Solr. `http://lucene.apache.org/solr/`, September 2012.

[11] Apache Thrift. Apache Thrift. `http://thrift.apache.org/`, September 2012.

[12] Apache ZooKeeper. Apache ZooKeeper. `http://zookeeper.apache.org/`, September 2012.

List of Figures

List of Tables