Cloud bursting for cloudy

Author(s):
Unternährer, Thomas

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005816848

Rights / License:
In Copyright - Non-Commercial Use Permitted
CLOUD BURSTING FOR CLOUDY

Master Thesis

Systems Group
November, 2008 – April, 2009

Thomas Unternaehrer
ETH Zurich
unthomas@student.ethz.ch

Supervised by:
Prof. Dr. Donald Kossmann
Tim Kraska
There has been a great deal of hype about cloud computing. Cloud computing promises infinite scalability and high availability at low cost. So far cloud storage deployments were subject to big companies but an increasing amount of available open-source systems allow also smaller private cloud installations. Cloudy is such an open-source system developed at the ETH Zurich. Although Cloudy has some unique features, such as advanced transactions and consistency guarantees, so far, it lacks an appropriate load balancing scheme making real-world deployments almost impossible. Furthermore, Cloudy scales only manually by interactions of an administrator who has to watch the system and register new machines if needed.

The aim of this thesis is to extend Cloudy by a sophisticated load balancing scheme and by cloud bursting. The latter describes the method to automatically integrate or remove nodes from the system depending on the load. Hence, this thesis first compares different load-balancing schemes, followed by a description of the newly implemented load balancing mechanism for Cloudy. The chosen mechanism not only allows to perform range queries but also makes it easier to integrate and remove nodes by minimizing data movements. Furthermore, this mechanism forms the foundation for the implementation of cloud bursting which is described afterwards. Finally the thesis presents first performance results for the load balancing and cloud bursting.
# Contents

1 Introduction 7

2 Cloud Computing 9
   2.1 Computing in the Cloud 9
   2.2 Cloud Bursting 10
   2.3 Available Systems 10
      2.3.1 Amazon Elastic Compute Cloud (EC2) 10
      2.3.2 Google App Engine 11
      2.3.3 Azure 11
      2.3.4 GoGrid 11

3 Related Work 13
   3.1 Fundamental Architectures 13
      3.1.1 Single Load Balancer 13
      3.1.2 Token Ring and Hash Function 15
   3.2 Available Protocols and Systems 17
      3.2.1 BigTable 18
      3.2.2 Cassandra 18
      3.2.3 Chord 18
      3.2.4 Dynamo 18
      3.2.5 Yahoo PNUTS 18

4 Cloudy 19
   4.1 Introduction 19
   4.2 Storage System 20
      4.2.1 Data Model 20
      4.2.2 Read/Write Path 21
      4.2.3 Memtable 22
      4.2.4 SSTable 23
   4.3 Membership and Failure Detection 24
   4.4 Messaging 26
   4.5 Partitioning and Replication 26
   4.6 Hinted Handoff 27
   4.7 Consistency 28
   4.8 Load Balancer 29
1 Introduction

Web applications and 3-tier applications are divided into several pieces. A presentation interface, a business logic and the storage and programming part to manage it. Most of this parts differ for different applications. The one part that can often remain the same for all applications is the underlying storage system. As a consequence of this, a whole bunch of different storage systems are already designed and implemented, commercial ones an open-source solutions.

These storage systems in general have to fulfill several requirements. One of it definitively is scalability, while others may be durability and availability. Having scalability in mind, these systems could be roughly categorized into two groups, the scale-up and the scale-out systems.

Traditional Databases
The traditional approach for a storage system are databases. They are well tested and proved to run very fast and secure for small web applications and services. Having the scalability in mind, databases are scale-up (horizontal scalability) solutions. Databases usually have one large shared memory space and many dependent threads. Scalability usually is achieved by adding more memory and CPUs. Furthermore, databases guarantee strong consistency.

Cloud Storage
A relatively new approach of storage system is cloud computing. The main difference compared to the database approach is that computing in the cloud includes several commodity hardware servers and not just a single machine. Cloud computing has vertical scalability (scales out) and has many small non-shared memory spaces, many independent threads and is usually a loosely-coupled system. Cloud storage systems usually do not guarantee strong consistency and have weak or eventual consistency to guarantee a good performance of the system.

Many vendors offer the service of cloud computing. The biggest player at the moment probably is Amazon with its web services. But the open-source world is still lacking a good cloud computing software that provides eventual consistency and ACID-like guarantees if needed. This is where Cloudy comes into place. Cloudy is a open-source distributed storage system\(^1\) developed at ETH that can be used to run as a cloud com-

\(^1\)Cloudy could be seen as a peer-to-peer system
puting service. Cloudy runs as a one node system as well as a large multi node system.

The crucial part of distributed systems in general is the load balancer. Without a load balancer the system behaves like a single server and has the same limitations. Although Cloudy is based on Cassandra, an open-source distributed storage system, it lacks so far a load balancer. Thus, the first part of this thesis was to expand Cloudy with a sophisticated load balancing algorithm that can be configured to use different load informations such as key distribution or CPU load.

Cloud Bursting

Another problem of web applications or applications in general is that the load usually isn’t uniformly distributed over the whole year. During Christmas time for instance, the request rate on Amazon’s Book store is much higher than usual. This is where dynamic scalability or cloud bursting is needed. The system should adapt itself to the load needed and integrate new nodes if the system is overloaded and remove nodes if the system is underloaded.

The design and implementation of a cloud bursting feature to Cloudy is the second part of this thesis. Cloudy should be able to automatically adapt its size depending on the overall system load.

The thesis has the following structure. Chapter 2 gives an overview of cloud computing and explains cloud bursting and its advantages. Chapter 3 presents several approaches for distributed systems and their load balancing and partitioning algorithms. In chapter 4 the storage and messaging service of Cloudy will be presented. Chapter 5 illustrates the load balancer in detail. Chapter 6 contains performance tests for the whole Cloudy system and in the last chapter conclusions are drawn and discussed.
2 Cloud Computing

As Cloudy is a cloud computing service, this chapter gives an overview of the two paradigms cloud computing and cloud bursting. Section 2.1 provides an introduction to cloud computing while section 2.2 describes the concept of cloud bursting. Section 2.3 lists and introduces several available cloud computing services.

2.1 Computing in the Cloud

Cloud computing is a new and promising paradigm delivering IT services as computing utilities [5]. It overlaps with some of the concepts of distributed, grid and utility computing, however it does have its own meaning. Cloud computing is the notion accessing resources and services needed to perform functions with dynamically changing needs. An application or service developer requests access from the cloud rather than a specific endpoint or named resource. The cloud is a virtualization of resources that maintains and manages itself [6].

The big advantage of cloud computing is that scalability and almost 24/7 availability is guaranteed by the service vendor.

In general it is a combination of several concepts like software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS) [6].

SaaS "Software as a service" is a business model that provides software on demand. A vendor offers a piece of software, which can be used with a web browser and usually is not installed on the client side. An example for SaaS is Google Docs [7] or Gmail.

PaaS Providers of "platform as a service" allow you to build your own applications on their infrastructure. Customers are usually restricted to use the vendor’s development software and platforms. Examples are Google App Engine [8], Facebook [9] or Microsoft’s Azure [10].

IaaS "Infrastructure as a service" seems to be the next big step on the IT market. Compared to PaaS there is usually no restriction for a customer. The idea is that everybody can rent any amount and size of services and virtual servers they need and has fully access to the infrastructure. This has the big advantage that a
customer can easily absorb even the highest load peaks on their application and
data with only low additional cost (no need to buy additional hardware). Ex-
amples are Amazon’s Web Services [1] including EC2, S3 and SQS or GoGrid
[11]. IaaS describes the combination of the two above. Instead of purchasing
dedicated hardware and software everything runs on a cloud computing system.

Cloud computing as well as utility computing describe the pricing of the service as a
customer pays-per-use. Furthermore, cloud computing is a business models where the
hardware or even the software is outsourced and the customers can focus on the develop-
ment of their applications. In a sense this represents the opposite of the mainframe
technology as a scalable application is built on many commodity servers which are not
even located at the customer’s company (they are somewhere in the cloud).

2.2 Cloud Bursting

Cloud bursting is a new paradigm that applies an old concept to cloud computing.
It describes the procedure in which new servers are rented if needed and returned if
not needed anymore. Cloud bursting represents the dynamic part of computing in the
cloud.

Having Cloudy in mind, cloud bursting means that if the system is overloaded for any
reason, the load balancer automatically integrates new nodes from the cloud. The big
advantage of this is that without any human interaction a highly scalable system is
available. The system grows and shrinks depending on the load and because of that
automatically minimizes the overall cost.

2.3 Available Systems

In the following sections some of the commercial vendors and market leaders of cloud
computing services will be presented.

2.3.1 Amazon Elastic Compute Cloud (EC2)

Amazon’s EC2 [1] is a commercial web service that allows customers to rent server
hardware and run their software on it. A customer can create an image which is loaded
at startup. A web interface allows easy configuration of rented servers. Almost unlim-
ited scalability is guaranteed by Amazon. A Java API [12] allows the user to start and
shut down applications using client software. Customers have to pay on demand.
Because of the very feature-rich Java API and the nice configuration possibilities,
Cloudy runs and uses Amazon’s EC2. Amazon’s EC2 provides "infrastructure as a
service".
2.3.2 Google App Engine

Google’s App Engine [8] is designed to run scalable web applications. The scalability is guaranteed by Google itself as it starts new servers if needed. The data is stored in a BigTable database and only a limited number of applications are supported. Google’s App Engine has a Java and Python API. Google’s App Engine provides "platform as a service".

2.3.3 Azure

Microsoft’s Azure [10] is a cloud computing service by Microsoft where the specialized operating system Azure runs on each machine. Scalability and reliability are controlled by the Windows Azure Fabric Controller. Microsoft provides customers with access to various services such as .Net, SQL or Live Service. Microsoft’s Azure provides "platform as a service".

2.3.4 GoGrid

GoGrid [11] is similar to Amazon’s EC2. GoGrid offers several different operating systems on which to deploy a customer’s web software. In addition, GoGrid has an integrated load balancer which distributes requests to different server nodes. GoGrid does not yet support auto-scalability as other vendors, like Google’s App Engine, do. GoGrid provides "infrastructure as a service".
3 Related Work

The first task of this thesis was to design and implement a new and sophisticated load balancing algorithm. Thus, related algorithms and protocols got studied first. This chapter gives a general discussion about load balancers and approaches for mapping data items to server nodes.

In section 3.1 3 different approaches will be presented on how to map data items to servers and which part in each system is responsible for load balancing. In section 3.2 some important protocols and systems will be presented.

3.1 Fundamental Architectures

One of the aim of a distributed storage compared to a single server system is to improve the overall storage space and availability. To improve the storage space a data item cannot be stored on every node in the system and a sophisticated algorithm has to be used to decide which data item has to be stored on which server node. In this section different partitioning algorithm as well as load balancers will be presented.

3.1.1 Single Load Balancer

A single load balancer or load dispatcher is probably the simplest approach in terms of implementation. Every request gets handled by the load balancer and forwarded to the appropriate node. The load balancer forwards requests to nodes based on the load on every single node. Figure 3.1 shows a simple setup of a single load balancer system with several clients and storage nodes.

As shown in figure 3.1, the interaction between the clients and the load balancer can be minimized by only sending metadata and letting the clients and server nodes interact directly for write and read requests. At least the first read request needs to be handled by the master node as the client doesn’t know where the data is stored. All subsequent request for the same data item can be cached on the client side. In the case of many clients, this approach leads to the problem that the master or load balancer is the bottleneck of the system as well as a single point of failure.

Different approaches can be used to bypass these problems. The approach taken by the Google File System[13] is that only meta data is stored on the master node and only node locations are transfered from the master to a client. All the data transfer is
Figure 3.1: Illustration of a single load balancer with several clients and storage servers. The overall idea is well described in [13]. Clients send a location request for a given data item to the load balancer. The load balancer is the only node in the system which has full partitioning information (mappings of keys to server nodes). Usually the load balancer periodically balances data items between server nodes.
handled by the clients and the storage servers directly. The number of possible clients and storage servers can be pretty large in such systems. The problem of having a single point of failure is solved by backing up the metadata on the master on several other nodes, which can quickly replace the master node in case it fails.

The look-up times in such systems depend on the load on the master node, but in general are constant $O(1)$. That means the look-up times does not depend on the number of storage nodes, but can depend on the number of clients and the number of requests per time interval. Scalability depends on the hardware of the master and the number of participating nodes and clients.

The main problem with having a single load balancer is the performance of the system. A well designed peer-to-peer-like system is able to handle many more requests per time interval. An other problem is that the load balancer should run on a separate node as the effect of the bottleneck should be minimized.

### 3.1.2 Token Ring and Hash Function

The idea of a token ring and a hash function is that the position of a given data item with a given key $k$ is calculated with a hash function $h(k)$. This ensures that the position is known a priori without any interaction with a master node. Two different approaches using hash functions will be discussed in the following subsections, consistent hashing [14] using a random hash function and the order preserving hash function.

Both of these approaches map the keys $k$ using the hash function $t = h(k)$ with a resulting token $t$ on a token ring $T$. Figure 3.2 illustrates these mappings.

The $m$-bit tokens are ordered on a token ring modulo $2^m$ where the larger the value of $m$, the more unique tokens exist. In a productive system $m = 64$ seems to be enough\(^1\) as this means that $\approx 2 \cdot 10^{19}$ different tokens are available.

The key to node assignment works as follows. Key $k$ is assigned to the first node whose token is equal to or follows $k$ on the token ring. This node is called the successor or the primary node of key $k$. In case of $n$ times replication, every key gets assigned to its $n$ successor nodes.

**Consistent Hashing with a Random Hash Function**

The idea behind consistent hashing (see [14] for further details) is that the assignment of keys to nodes is completely random and as a consequence the keys are uniformly distributed over all existing nodes. This can be achieved by a cryptographic hash function like SHA-1, [15] where given a large number of $m$, collision is believed to be hard

\(^1\)Cloudy uses $m = 128$
Figure 3.2: Token ring with three nodes and several keys. The range between node \( n_1 \) and node \( n_2 \) is the range of node \( n_2 \), that means that every key in this range will be stored on node \( n_2 \).

Load balancing seems to be automatically given by a random hash function, and as a consequence no data has to be moved during a balancing process, nor does metadata about storage places have to be sent from clients to the master and vice versa. However, this approach has three major drawbacks

- Only the keys get balanced over the system. This means that whenever a data item is bigger in size than another, the balancing is non-optimal. Furthermore, if the request distribution is skewed, some nodes have a much bigger load than others.
- This approach does not support range queries. Therefore, every single data item needs to be looked up as similar keys are generally not on the same node.
- The nodes are only balanced if the number of keys is much larger than the number of nodes.

**Order Preserving Hash Function**

The main difference of order preserving hash functions compared to consistent hashing is that range queries are possible. This is ensured as keys get mapped on the token
ring without randomization. As an example, if keys can only be integer values, key 2 is guaranteed to be a direct neighbor of the keys 3 and 1. This ensures that range queries are possible.

Order preserving hash functions also have disadvantages. Assuming the nodes in figure 3.2 are equally distributed, the overall load is only balanced if the load is also uniformly distributed. As this is rarely the case, the system is generally in an unbalanced state.

To overcome this problem, a load balancing process needs to be run from time to time (see also section 3.2). The idea behind this load balancing process is pretty simple. Whenever a node is overloaded, it tries to balance itself with other nodes in the system, preferably with its direct neighbors. This can be achieved by giving the heavily loaded node an other token and moving the appropriate keys to the neighbor node (figure 3.3 illustrates this process).

Figure 3.3: A token ring with 3 nodes. Node 1 is heavy and gets moved to a new position. After moving to a new token it sends the appropriate keys (range is represented by the thick line) to node 2.

3.2 Available Protocols and Systems

This section presents some of the more important available systems and protocols regarding partitioning and load balancing of distributed systems. Further protocols using
consistent or order preserving hashing and general discussions about partitioning, not presented here, are described in [16] and [17].

3.2.1 BigTable

Google’s BigTable [2] provides high scalability and performance. It is built on top of the Chubby Lock Service [18] and the Google File System [13]. BigTable is not an open source solution but can be accessed by the Google App Engine. The Google File System is a master-slave-like design, where the master node is responsible to handle the load balancing of the system.

3.2.2 Cassandra

Cassandra [4] is a structured storage system design for use in Facebook [9]. It uses an order preserving hash function to map data items to server nodes. As Cloudy is based on Cassandra, a detailed description can be found in chapter 4. In time of this thesis Cassandra does not have a load balancer, but it is planed to used the load balancer algorithm described in [19].

3.2.3 Chord

Chord [20] is a structured peer-to-peer system where nodes do not have a full view of the system. This means that a single node just has partial information on other nodes and routed queries are performed to get the needed information. The partitioning information is stored in so called finger tables. The main advantage of this approach is that it guarantees unlimited scalability. On the other hand, it has the disadvantage that the look-up time is in $O(\log n)$. Chord uses an random hash function.

3.2.4 Dynamo

Amazon’s Dynamo [21] is a key-value storage system which uses a random hash function to uniformly distribute data items over the available nodes in the system. It has extremely fast read and write response times but is limited in size due to the gossiper and other message services.

3.2.5 Yahoo PNUTS

PNUTS [3] provides ordered tables, which are better suited for sequential access. PNUTS has a messaging service with publish/subscribe pattern and does not use a gossiper and quorum replication.
4 Cloudy

As the aim of this thesis was to implement and test a new load balancing algorithm, the first step was to choose an underlying system. The development of Cloudy on top of Cassandra [4] was already started in a previous master thesis [22]. The aim there was to build a storage system with adaptive transaction guarantees. Therefore, in the following we use and expand Cloudy.

This chapter describes Cloudy. Section 4.2 describes the structured storage system. The next sections describe the gossiping and messaging services of Cloudy. Section 4.5 gives an illustration how partitioning and replication is solved. Section 4.6 describes the solution to ensure that Cloudy is always writable and section 4.7 describes the consistency model of Cloudy. The last section gives a very brief description of the not yet implemented load balancer of Cassandra which is replaced in Cloudy and described in detail in the next two chapters.

4.1 Introduction

The Facebook data team initially released the Cassandra project on Google Code [4]. The problem Cassandra tries to solve is one that all services that have lots of relationships between users and their data have to deal with. Data in such systems often needs to be denormalized to prevent a lot of join operations. This means that because of the denormalisation, the system needs to deal with increased write traffic. A lot of the design assumptions of relational databases are violated, and because of this using a relational database is inappropriate. Facebook tackled this problem by coming up with Cassandra. Cassandra is a cross between Google’s BigTable [2] and Amazon’s Dynamo [21]. The whole storage system is inspired by BigTable (described in the next section) whereas the idea of the eventual consistency model and the "always writable" state of Cassandra is taken from Dynamo.

This chapter describes Cloudy but the majority of what is mentioned here is also true for Cassandra.
4.2 Storage System

4.2.1 Data Model

The Cloudy data model is more or less derived from the data model of BigTable, but includes a few extras.

The entire system is one big table with lots of rows. Every row is a string of arbitrary length and each row has a unique key. Read as well as write operations on keys are atomic.

Every row exposes an abstraction called a "column family" (an overview of a row is illustrated in figure 4.1).

Figure 4.1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN’s home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t3, t5, and t6.

Source of this graphic: BigTable [2]

**Column Family**

A column family\(^1\), which can be seen as a schema for a row, can have one of two structures, simple columns or super columns. The difference is that a super column can contain many simple columns. The column family can be seen as a schema for a row.

*Type simple:* A column family of type simple conceptually provides a two-dimensional sparse map-like data model:

\[
\text{Table:CF:(key name, column name)} \rightarrow [\text{value, timestamp}] 
\]

\(^1\)parts of this section are taken from the Google Code site of Cassandra[4]
The Cloudy model also allows wild-card style look up on the column name dimension. Table:CF:(key name, *) will yield back the matching set as a bunch of <column name, value, timestamp> tuples.

**Type Super:** Column families of type super allow a three-dimensional sparse map-like data model:

Table:CF:(key name, super column name, column name) → [value, timestamp]

As in the case of two-dimensional column families, wild cards are allowed for any parameter of a query. A column family of type super allows an infinite number of simple columns.

As it is assumed that column families change very rarely, column families can not be dynamically modified. A restart of the system is required.

**Reason for Column Families**
A row-oriented database stores rows in a row major fashion (i.e. all the columns in the row are kept together). A column-oriented database on the other hand stores data on a per-column basis. Column Families allow a hybrid approach. It allows you to break your row (the data corresponding to a key) into a static number of groups a.k.a. column families.

**Timestamp**
Each value is assigned a timestamp (32-bit integer). In Cloudy the timestamp must be set by the client and is used for resolving conflicts. Different clients could set the same timestamp what could lead the system into an undefined state (but this is very unlikely). In BigTable timestamps are used for versioning. In Cloudy on the other hand, timestamps are just used to get the most recent version of a value (other values will be overwritten).

### 4.2.2 Read/Write Path
Figure 4.2 gives an overview of the different storage parts and a write path in Cloudy. The memtable (memory representation of a column family) as well as the SSTable (disk representation) will be presented in the subsequent sections.

A write request has to contain the table name, the row key, the column family name or a super column family and all its column names, a timestamp and the cell data. All this information has to be sent in a so called rowMutation object. Using this rowMutation object a commit log entry is written and the changes for each column family are written to the corresponding memtable (located in main memory). If one of several
conditions is satisfied, the memtable gets flushed to a so called SSTable on the disk.

A read request is handled using the same order of access as a write request. First the corresponding memtables for the specified column families are checked. If the query includes the exact names of the columns to be read, the query is fully answered. Otherwise the client may have send a wild-card type of a query and the system has to return all columns. If the key cannot be found in one of the memtables, the SSTables have to be scanned using the in-memory bloom filter (see section SSTable for a more detailed description).

![Illustration of the write path of Cloudy](image)

Figure 4.2: Illustration of the write path of Cloudy. A rowMutation object gets sent to a node, which applies all mutations first to the commit log and then either to the corresponding Memtables or to the SSTables.

### 4.2.3 Memtable

In memory each column family is represented using a hashtable (Memtable), which includes all row mutations. If it exceeds one of several conditions, the memtable is written to a so called SSTable on the disk. In Cloudy it is also possible to force this flush from memory to disk by sending a `flush-key`.

Conditions for flushing memtables to disk (can be configured)

- maximum memory size
- maximum number of keys
- lifetime of an object
4.2.4 SSTable

SSTables\(^2\) are the disk representation of column families. Each column family is written to a separate SSTable file. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Each SSTable contains a sequence of blocks (in Cloudy each block has a length of 128 KB) with a block index at the end of the file (index on the keys of each row). After opening a SSTable the index is loaded into memory and allows fast seeks. At the end of the SSTable a counting bloom filter is added for fast checks if a given key is in the current SSTable. As the SSTable is immutable, every change creates a new file.

As mentioned above under some circumstances Memtables are flushed to SSTables on disk. As they are immutable, every flush creates a new file for every column family. This process is called *minor compaction* in Cloudy. Because of this, the system will generally have to merge several SSTables during a read operation, which can cost a lot of time. To overcome this problem Cloudy periodically merges SSTables to speed up read operations during so called *major compactions*.

**Bloom Filter**

As mentioned above, a bloom filter \(^{[23]}\) is used to decide if a SSTable contains a given key. In general a bloom filter is a space efficient probabilistic data structure used to test whether an element is a member of a set. False positives are possible, but false negatives are not. As elements can be added but not removed this is a counting filter. In short it holds that the more elements are added, the higher the probability of false negatives.

Counting bloom filters are used in Cloudy to minimize the disk reads of SSTables.

*Algorithm:* A bloom filter is a bit array of length \(m\) (initially all values are set to 0). A set of \(k\) hash functions with a uniform random distribution hash a key \(l\) to at most \(k\) different bits on the array and set those to 1. To query if an element with key \(l\) is part of the set only the \(k\) hash values (positions on the array) are computed and checked if all the bits are set to 1. Removing values from the bloom filter is impossible\(^3\).

*Space/Time:* Bloom filters have a big space advantage over other data structures. With 1% error and an optimal value of \(k\), the filter requires only about 9.6 bits per element, regardless of the size of the elements. Every additional 4.8 bits give a decrease of false positives by a factor of 10. The time to add an element or check if an element is part of the set is constant in \(O(k)\).

---

\(^2\)see Google’s BigTable\(^{[2]}\)

\(^3\)One approach is to simulate this by using a second bloom filter containing the removed keys. Note: This approach leads to false positive values in the original bloom filter.
**Probability of false positives:** Assuming that the hash functions are normally distributed, the probability that a single bit is set to 1 is:

\[ 1 - \left(1 - \frac{1}{m}\right)^{kn}, \text{ for } m \text{ bits, } k \text{ hash functions and } n \text{ inserted elements} \]

Now the probability that for a not inserted element all the bits are set to one is:

\[ \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \]

which is minimized by \( k = m/n \ln 2 \) and becomes

\[ \left(\frac{1}{2}\right)^k \approx 0.6185^{m/n} \]

### 4.3 Membership and Failure Detection

**Membership and Gossiping**

Cloudy uses a gossip-style protocol to propagate membership changes, very similar to the one used in Dynamo [21]. In Cloudy at least one node acts as the seed where all the other nodes have to register themselves. After registration each node has a list of state triples [address, heartbeat, version], where the address is the hostname of a node, the heartbeat an increasing number for life detection and the version the last time the heartbeat was increased.

**Gossip Algorithm:** Every node \( a \) periodically\(^4\) increases its own heartbeat and randomly selects an other node \( b \) in the system. Node \( a \) sends \( b \) a so called SYN message including a digest of its own state. Node \( b \) then checks if something differs from its stored information about \( a \) and updates it if needed. Node \( b \) responds with an ACK message, which includes a request if something was missing in \( a \)’s message. Node \( a \) responds and terminates the interaction with an ACK2 message.

**Scalability:** Gossip protocols are highly scalable protocols, as a single node only sends a fixed number of messages, independent of the number of nodes in the system.

**Properties:** Gossip protocols have some nice properties. A node does not wait for an acknowledgement and as all information usually gets received by a number of nodes, fault-tolerance can be achieved. Furthermore, all nodes perform the same tasks. Thus, whenever a node crashes the system will still work.

---

\(^4\)In Cloudy every second
**Dissemination Time:** For a defined number of nodes $n$ and the number of nodes $k$ that already have a specific piece of information, the probability that an uninfected member gets infected\(^5\) in a gossip round is as follows:

$$P(n, k) = 1 - \left(1 - \frac{1}{n}\right)^k$$

Thus, the expectation value for newly infected nodes is:

$$E(n, k) = (n - k)P(n, k)$$

Figure 4.3 shows the infection rate for a 100-node system. It can be seen that after 10 gossip rounds more than 99% of all nodes are infected. This result is very important as the load balancer presented later uses the gossiper to spread load information.

![Figure 4.3: Infection rate for a 100-node system](image)

It can be summarized that the infection rate is in $O(\log n)$.

**Failure Detection**

Every distributed systems has to deal with failure detection as at any time a node could crash or has to be replaced. Thus, a node that is down has to be detected in a reasonable amount of time.

Cloudy uses a $\varphi$ accrual failure detector[24].

**Algorithm:** The value $\varphi$ describes a suspicion level that a node has crashed and is dynamically adjusted to reflect the current network situation. The value of $\varphi$ is

\(^5\) This model of infection spreading is referred to as proportionate mixing. A machine that has received the update is described as infected and one that has not received it yet as susceptible
calculated as follows.

\[ \phi(t_{\text{now}}) = -\log_{10}(P_{\text{later}}(t_{\text{now}} - t_{\text{last}})) \]

Where \( t_{\text{now}} \) is the current time, \( t_{\text{last}} \) is the time when the most recent heartbeat message was received and \( P_{\text{later}} \) is the probability that a heartbeat message will arrive more than \( t \) time units after the previous one. \( P_{\text{later}}(t) = 1 - F(t) \) is a cumulative normal distribution function with mean \( \mu \) and variance \( \sigma^2 \).

As the difference \( t_{\text{now}} - t_{\text{last}} \) goes to infinity, the probability \( P_{\text{later}} \) goes to 0 and the suspicion level \( \phi \) goes to infinity. In Cloudy two levels of node crashes can be configured by adjusting a number of thresholds \( \theta_i \), a node can either be temporarily or permanently down. If \( \phi > \theta \) a node is assumed to be dead. Given the definition of \( \phi \), a threshold of \( \theta = 1 \) leads to an error of 10% which increases by a factor of 10 if \( \theta \) gets incremented by 1.

As Cloudy is always writable, node crashes are handled by the hinted handoff approach (see section 4.6).

### 4.4 Messaging

Cloudy uses the staged event-driven architecture (SEDA [25]) as message service. The idea behind this is that on the receiver’s side several thread pools handle requests and each of these thread pools processes only a part of the whole task. On receiving a message, the message service decides based on a verb entry in the message header which thread pool is responsible for a given message and calls its verbhandler. In the original paper these thread pools are dynamic and can adjust their size. This feature is not implemented in Cloudy and the thread pool size needs to be adjusted manually\(^6\).

### 4.5 Partitioning and Replication

Partitioning is based on an order preserving hash function as described on page 16.

**Algorithm:** A client sends data items including a key to one of the system’s entry points (every node could be an entry point). The entry point hashes the key to get a 160 bit token value, and maps this value on the token ring. The first node on this ring with a greater or equal token is called the successor or primary node for this key and has to handle the write or read request. The token of each node gets periodically gossiped and because of that every node knows the whole setup of the system at any time.

\(^6\)The size of each thread pool could be critical in a system with a large number of clients
given point in time\footnote{Because of the log \( n \) delay of the gossip protocol, it is possible that the system is inconsistent for a small time interval}. Figure 4.4 illustrates this mechanism.

A description of several write mechanisms in Cloudy can be found in section 4.7.

Figure 4.4: A token ring with three nodes. A write request from a client to the entry point (node 2 in this setup) gets forwarded to the appropriate node for this key (key 1 gets forwarded to node 3).

Usually a data item is not just stored on the primary node but also on all its replicas. For a \( n \)-replicated system, every node stores \( n \) different key ranges (See figure 4.5).

### 4.6 Hinted Handoff

Cloudy ensures an always writable state. This means that as long as at least one node is up and there is enough disk space available, a data item can be written to the system. This is even true for keys that are in a range for a node that is currently down. To ensure this Cloudy uses an approach called hinted handoff \cite{21}.

**Algorithm:** Hinted handoff is pretty simple in theory. Given is a set of nodes \( \{A, B, C, D, E\} \) where every node has a primary range (D has the range \([C, D]\)). If a node \( C \) goes down, then the range is taken over by the next node.
Figure 4.5: token ring with three nodes and a replication factor of two. Key $k_1$ gets forwarded to its primary node $n_2$, which forwards the write request to all its replications (node $n_3$ in this case).

on the token ring ($D$ in this example). Node $D$ is now responsible for the range $[B, D]$. If node $C$ is up again, $D$ will send all the keys and data items in the range $[B, C]$ back to $C$.

4.7 Consistency

Strong consistency in distributed systems is very time intensive and in general slows down the whole system a lot. Thus, Cloudy has different levels of consistency and lets the client decide if a certain read or write request needs stronger or relaxed consistency. Cloudy uses the following protocols for this purpose.

**Blocking Write Protocol** The blocking write protocol works as follows. The entry point $n_e$ sends the write request to the primary node $n_p$ for a particular key $k$ and data item. The primary node then sends key $k$ to all its replicas and waits for their acknowledgments. If the majority response with a successful ACK, the write was successful and the primary node sends a ACK back to the client.

**Non-Blocking Write Protocol** Non-blocking write works similar to blocking write except that the primary node immediately returns if its local write was successful. It does not wait for the replicas of the given key.
**Strong Read Protocol**  If a read request gets handled by the primary node it reads the requested columns and forwards the request to the next $n - 1$ replicas. All replicas respond with a hash value of the requested data. The primary node compares all hash values and returns the data item if the majority of the values are the same. For nodes that do not agree, a read-repair mechanism gets started, which takes the value with the most recent timestamp and updates all nodes.

**Weak Read Protocol**  The weak read protocol works pretty much the same way as the strong protocol does with the exception that the requested data immediately gets returned without waiting for all other replicas. The read-repair mechanism runs in the background.

### 4.8 Load Balancer

Cassandra has no load balancer implemented yet, but an implementation of [19] is planned. A detailed description of all the load balancing processes can be found in the next chapter as Cloudy uses a similar approach.

---

8 Assuming the data items are stored on $n$ nodes.
9 This protocol is not yet available in Cloudy’s RPC interface.
5 Load Balancer

In this chapter the newly added features of Cloudy are illustrated. Section 5.1 gives a mostly informal overview of the load balancer while in section 5.2 the implementation is presented.

5.1 Abstract Description

This section gives an overview of the load balancer in Cloudy. In section 5.1.1 the requirements of the load balancer are listed and discussed. Section 5.1.2 is a discussion of the load unit used in the load balancer. In section 5.1.3 the design decisions are presented. Section 5.1.4 and section 5.1.5 are descriptions of the load balancer algorithm.

5.1.1 Requirements

Listed below are the design requirements the load balancer and the whole system have to fulfill.

**Range Queries** Range queries are one of the major requirements of Cloudy. As an example, a query for \( n \) consecutive keys should include only one request from the client and all the keys should be either on the same node or distributed on a few consecutive nodes on the token ring. This means that the node holding the first key should be able to just forward the query to its successor if needed.

**Consequence:** As described on page 15, a random hash function does not fulfill this requirement as similar keys get randomly distributed over the available nodes. Thus, Cloudy uses an order preserving hash function\(^1\).

**Minimized Data Movement** The big advantage of a random hash function is that on average the load is equally distributed over all nodes. As this is not the case using an order preserving hash function, a load balancer has to move the nodes and as a consequence a part of the data. In general this is a very time consuming task and can slow down the system dramatically. Thus, data movement has to be minimized.

\(^1\)Cassandra is already designed to use an order preserving hash function
**Manageable Complexity** The load balancer task on each node should not slow down the whole system too much. Also, the algorithm should be implementable in the available time.

**Intelligent Load Information** The load information of systems using random hash functions are often based on the key distribution. In Cloudy it should be possible to configure the system to use other criteria and more sophisticated load information.

**Cloud Computing** Cloudy must run on a cloud computing environment such as Amazon’s EC2.

**Cloud Bursting** The size of a running Cloudy cluster should be automatically adjusted by the load balancing process. This means that whenever the system load exceeds a certain threshold, new nodes should be integrated to the system as fully functional nodes.

### 5.1.2 Load Information

A load balancer is essentially based on at least one number, which enables it to decide if a node is abnormal (over- or underloaded) in relation to other nodes in the system. This number is the load of a node. In general, if the load on one node is much higher than on another node, an algorithm on at least one node has to decide how to react on this state. The load balancer has to balance the nodes, so that all nodes have similar nodes at any given time.

Cloudy can be configured to use one of three different load ratings: the key distribution, the CPU load and the number of requests. The following part shows the advantages and disadvantages of each of these.

**Key Distribution**

As often used in systems using a random hash function for partitioning data items, the key distribution probably is the simplest kind of load information. Key distribution in this case is the same as data item distribution. The aim is to have data items equally distributed among all nodes in the system.

**Advantages:**
It seems to be the easiest way for load balancing. Every node can just count its stored keys and disseminate this information.

**Disadvantages:**
If the distribution of the keys is the only load information, the system is only balanced if all data items are of equal size and if the requests for all the keys are the same.
**CPU Load**

It seems to be appropriate that in systems where the CPU is the bottleneck, the CPU load is the information to use for load balancing. The experiments presented later should show if this information is appropriate for Cloudy.

*Advantages:*
In computing intensive systems this number seems to be appropriate.

*Disadvantages:*
In systems where the network traffic or the disk is the bottleneck, the CPU load is not meaningful. Furthermore, in case of more than one virtual node per server node, the CPU load is only measurable for the whole node.

**Number of Requests**

Read and write requests can be used as an approximation of CPU load and network traffic.

*Advantages:*
This number covers much more than the two metrics mentioned above. The network traffic as well as the CPU load is combined.

*Disadvantages:*
Differences in data item size are not factored in.

**Combination of the above**

A combination of e.g. CPU load and network traffic isn’t yet implemented in Cloudy.

*Advantages:*
Covers more different aspects of the system like CPU load, network traffic, request rates and key distribution.

*Disadvantages:*
A meaningful weighting of all considered factor is needed.

**5.1.3 Design Decisions**

As listed in section 5.1.1, the load balancer in Cloudy has to fulfill several requirements. In this section the design decisions will be presented.

**Minimized Data Movement**
To minimize the overhead of moving many, possibly large data items while load balancing, the load balancer in Cloudy uses **virtual**
nodes (this is illustrated in figure 5.3). The notion of virtual nodes is taken from Dynamo [21] and means that a physical server node could be represented by more than one node on the token ring. The exact effect of using virtual nodes will be illustrated in section 5.1.5. The major drawback of using virtual nodes is the higher complexity of the whole load balancing algorithm but the minimized data movement makes it worthwhile to make this trade-off.

Cloud Computing Cloudy runs either on a local cluster or on Amazon’s EC2. As Amazon’s service has a nice Java API and is already in use at our department, we decided to use EC2 as the testing system.

Cloud Bursting As mentioned above, Amazon’s EC2 has a Java API where nodes can be started and integrated with preconfigured images as well as removed. This feature is not available for local clusters, but Cloudy could be relatively easily enhanced to run dynamically even on local clusters.

Leader In Cloudy one node gets elected to be the leader node. This node handles the dynamics of the system, which means that only the leader can remove or integrate new nodes. This has the advantage that only one node at a time can be removed or integrated, respectively. This prevents the system to grow or shrink faster than needed.

The load balancing algorithm is splitted into a node driven and a leader driven part, which will be discussed in the following sections.

5.1.4 Node Driven Balancing

Every node \( n_i \) periodically checks its own state. If a node declares itself overloaded the following algorithm gets started.

1. if \( l_i > \mu L \), with \( l_i \) the load of node \( n_i \), \( L \) the average system load and \( \mu \) a configurable factor\(^2\), node \( n_i \) is overloaded. Then one of the following steps will be executed.
   a) if \((l_{i+1} + l_i)/2 \leq \mu L \) or \( l_{i+1} < L \), with \( l_{i+1} \) being the load of the successor node of \( n_i \), \( n_i \) moves itself to a lower token on the token ring
   b) if \((l_{i-1} + l_i)/2 \leq \mu L \) or \( l_{i-1} < L \), with \( l_{i-1} \) being the load of the predecessor node of \( n_i \), \( n_i \) sends a moveNode request to its predecessor
   c) if neither the successor nor the predecessor are light, node \( n_i \) selects a random light node \( n_r \) in the system and sends a createVirtualNode request to node \( n_r \)
   d) if no light enough node can be found, node \( n_i \) just skips the load balancing step

\(^2\)Default is 1.5 in Cloudy
2. If \( l_i < \mu L \) node \( n_i \) just skips the load balancing step.

The condition \( l_j < L \) for \( j = i + 1, i - 1 \) ensures that the load balancer works even if a lot of data is on one node and all the other nodes are completely free.

The following figures illustrate each step. Figure 5.1 shows step a) where the successor is light, figure 5.2 shows b) where the predecessor is light and figure 5.3 shows the creation of a virtual node.

![Diagram of load balancing steps](image)

Figure 5.1: Successor is light. Node 1 is overloaded and moves itself to a lower token. After that it sends the appropriate keys to the light successor node.

### 5.1.5 Leader Driven Balancing

One node in the system plays a more important part than the others. One node gets elected to be the leader node and has some special purposes.

The leader is responsible for handling the dynamics of Cloudy. It is able to add and remove nodes as described in the following sections.

---

3Leader election is not implemented yet. The node with the lowest host id currently plays the leader part. This makes it possible for two nodes to be leader nodes at the same time for a short period of time.
Figure 5.2: Predecessor is light. Node 1 is overloaded and node 2 is not light enough. Thus, node 1 sends a move node request to the light predecessor node 3 and moves the appropriate keys.

**Remove Nodes**

Whenever the whole system is light, the leader node checks if there are any light nodes available. If there is any light node, the leader sends a remove node request. If a node receives a remove node request, it sends its data to the next replica and gossips a removed node message. All nodes in the system instantly remove this entry and update their token maps.

**Cloud Bursting**

The second task of the leader is to handle the state of an overloaded system. If the average load is above a given threshold, the leader integrates a new node from the cloud.

*Algorithm:* The procedure is very similar to creating new virtual nodes. The leader computes the target token based on the current load distribution and starts a new EC2 node with the computed target token. Figure 5.4 illustrates this procedure.

---

4This is a parameter that can be set in the configuration file
Figure 5.3: No light neighbor. Node 1 is overloaded and node 2 and node 3 is not light enough. Thus, node 1 sends a create virtual node request to the light node 4 and moves the appropriate keys. In case of without replication only the range from node 4_2 to node 3 needs to be moved. If no virtual nodes would be used, also the range from node 4_1 to node 2 needs to be moved to node 3.

Figure 5.4: Integration of a new node from the cloud.
5.1.6 Replication

Usually distributed systems allow replication of data. The replication factor has an influence on where to move the keys in a given range on the token ring. Figure 5.5 illustrates a system with a replication factor of 2. Node 1 is overloaded and sends the keys in the range to move to node 3 instead of node 2. This is because node 2 already has these keys and node 3 is now responsible for them, too.

Figure 5.5: System with a replication factor of 2. Node 1 sends the data items to node 3 instead of node 2 (which it would do if there were no replication)
5.2 Implementation

In this section detailed descriptions and code samples of the load balancer will be presented. Section 5.2.1 illustrates how the load information affects the balancer. Section 5.2.2 presents the algorithm each node can perform and section 5.2.3 presents the algorithm of the leader node.

5.2.1 Load Information

In Cloudy the load information, as described in section 5.1.2, is implemented as follows.

The system can be configured to use one of the described primary loads (key distribution, CPU load, number of requests). This load as well as the key distribution (secondary load) gets collected on every node and periodically gossiped around. As described in section 4.3, the gossiper disseminates the load information to all other nodes in $O(\log n)$, $n$ being the number of nodes. This means that with a rather small delay every node in the system knows the load on every other node.

The CPU load as well as the number of requests are mean values over a preconfigured time interval. This prevents the load balancing algorithm to be based on temporary peaks that are unusual for a given key range and time period.

5.2.2 Node Driven Balancing

As described in section 5.1.4, each node periodically performs its own load balancing task. Given the primary and secondary load on all other nodes, every node is able to decide if its own load is above a given threshold and if its neighbors are light enough to perform a load balancing step.

Algorithm 1 illustrates the interaction with the neighbors. A generalized situation of a token ring is illustrated in figure 5.6.
Algorithm 1 Handle local load.

1: procedure HANDLELOCALLOAD
2:   localLoad ← lLoad(loadMap) \Comment{global field, updated by tryNeighbor}
3:   for all vn : virtualNodes do
4:      tryNeighbor(vn, successor)
5:      tryNeighbor(vn, predecessor)
6:   end for
7:   if !isHandled AND isHeavy then
8:      createRemoteVirtualNode()
9:   end if
10: end procedure

Description of handleLocalLoad: The algorithm steps through the list of available virtual nodes (N1_1 and N1_2 in figure 5.6). For every virtual node it tries to balance load with its immediate neighbors. In case of n-time replication this will be the n-th neighbor. If some amount of load can be balanced it updates the variable localLoad and tries the next virtual node until the current node is no longer overloaded. If no direct neighbor is light, it sends a createRemoteVirtualNode request to a remote light node including the target token.

Algorithm 2 shows the most important computation steps when trying to balance the
load with a given neighbor.

**Algorithm 2** Try to balance with the successor

1: procedure TRYNEIGHBOR
2:   percToMove ← (localLoad − loadLight)/(2 · localLoad)
3:   loadToMove ← min{percToMove · keysOnHost, keysOnVirtualNode}
   ▷ keysOnHost: number of key on the server node
4:   targetToken ← lookUpToken(loadToMove)
5:   moveNode(targetToken)
6:   moveData(currentToken, targetToken)
7: end procedure

The variable loadToMove is the number of keys that can be moved from a given virtual node to its neighbor. Assuming a node moves 3 keys to its successor it has to find the key with the third highest token $k_3$ and move itself right next to this key with token $h(k_3) − 1$.

The procedure for the predecessor and a random new virtual node is very similar to algorithm 2 except that the current node sends a moveNode or a createVirtualNode message to the target node and does not move itself or any data in this step. The part of the algorithm that sends the data is shown in algorithm 3.

**Algorithm 3** Try to balance with the successor

1: procedure TRYNEIGHBOR
2:   if nodeInRange then
3:     sendKeys(currentToken, newToken)
4:   end if
5: end procedure

When a target node receives a moveNode message it updates its local token and gossips the new token to all other nodes in the system. Whenever a node detects that a token was moved into its own primary key range it runs algorithm 3 and sends the appropriate keys.

The main advantage of this procedure is that the moveNodeMessage can be lost without any effects. If its lost, the load balancer just sends another message to the target node. Also sending more than one message is not a problem as the target node can just ignore the second message with the same target token.

Using the gossiper to disseminate token updates and to send data later has the drawback of a small latency. This latency is in general in $O(\log n)$ but on average much smaller.

To overcome this problem of a possible inconsistent state, Cloudy has the read-repair consistency model, see section 4.7.
5.2.3 Leader Driven Balancing

This section describes the tasks of the leader node. The first subsection describes how nodes can be removed from the system and the second subsection how cloud bursting works in detail in Cloudy.

remove nodes

If the system’s average load is below a given threshold, the system is in an underloaded state. The leader then looks for any server node with more than one virtual node and tries to remove the lightest virtual node from the system. If no virtual node can be remove, the leader looks for a node where the whole load is below a given threshold and tries to remove it (see algorithm 4 for the leader part and algorithm 5 for the receiver part of the remove node message).

Algorithm 4 Remove a light node from the system

1: procedure REMOVELIGHTNODE
2:   lightNode ← findLightNode(loadMap)  ▷ findLightNode excludes the seed
3:   sendRemoveMessage(lightNode)
4: end procedure

Algorithm 5 Received remove node message

1: procedure RECEIVEMOVENODEMESSAGE
2:   gossip(removeMe)
3:   sendData(primaryRange, nthSuccessor)
4: end procedure

To summarize, removing a node $n_r$ from the system includes gossiping the new state of the system and the movement of the primary range of $n_r$ to its n-th successor.

cloud bursting

In the configuration file of Cloudy an upper limit for the average load can be specified. If this threshold is exceeded, the leader node finds the most overloaded node and computes the target token for the new node. If the target token is known, the leader sends a newInstanceRequest to the EC2 web service an integrates the new node at the computed token. The detection of the new node and the data movement is the same procedure as above.
6 Experiments

In this chapter several performance tests will be presented. Section 6.1 contains basic tests as response times for read/write requests for different data sizes as well as different requests rates. Section 6.2 contains performance tests for the load balancer over time and section 6.3 a discussion of the cloud bursting feature and the still available bugs.

As the distribution of response times is in general very skew, they are shown using boxplots (see figure A.1 for a description of a boxplot).

6.1 Basics

In this section read and write requests got measured depending on the data size and also the scalability for a fixed system will be presented.

The general system setup is that 3 nodes are running with a replication factor of 2 (read-repair is enabled and every request gets handled by 2 nodes). Furthermore, all 3 nodes are entry points and are known to each client. Requests are uniformly distributed over the 3 nodes.

Request times for different data sizes

Figure 6.1 shows the write requests for data sizes from 1 byte to 10 megabyte including 200 runs per data size.

It seems that the response times for 1 KB are smaller than for 1 byte. But as all these numbers are in the range of milliseconds, this could be an artefact as the larger data sizes were written after the smaller ones.

Handled Requests per time for different request rates

On every node the handled requests per time interval were measured. The test setup was as follows. On 4 client servers simultaneously a client software were started. This benchmarker starts on every client server every 20 seconds an additional 50 threads. This means, that every 20 seconds 200 additional client threads got started which continuously send requests to one of the 3 storage nodes.
Figure 6.1: Write requests for a three node system. Local requests are requests that could read from the local hard drive whereas remote requests are requests that got forwarded to other nodes. Note that the time axis is in logarithmic scale.

Figure 6.2 show the number of write requests dependent on the number of client threads. Figure 6.3 is the analog picture for read requests.

The figures show the maximum request rate that could be handled by the system. The smoothed curve shows a maximum value just above 600 requests per seconds. As it is a three node system with a replication factor of 2, the request rate for a one node system is above 400 requests per seconds.

Both figures show that the maximal throughput of the system is achieved when running about 250 client threads per server node.

The variability seems to be proportional to the number of clients. This can be explained by several factors like disk flushes of memtables and an increasing number of messages between the nodes.

**CPU Load**

Figure 6.4 shows the CPU load for an increasing number of clients. At the end of the graphic the whole system has maximal throughput. The CPU load is only around 40%. This graphic shows, that the CPU is not the bottleneck of the system, thus the CPU load is most likely not a useful load information for Cloudy.
Figure 6.2: Write requests per clients.

Figure 6.3: Read requests per clients.
6.2 Load Balancing

In this section the influence of the load balancer on the request times get presented.

The system setup is as follows. 3 nodes are running and 750 client threads send requests to the system. To test the system under a realistic load situation, a 80-20 distribution rule was used (see page 55 for details).

In the following we use the mentioned fixed load distribution and varying the load information.

Key Distribution

Figure 6.5 shows the request rate for the 3 nodes. The 3 nodes have the same amount of keys, but the request rate on the nodes differs. At time step 10 the first load balancing step is performed. As the black curve decreases after this step, some highly requested keys are moved to one of the nodes (blue curve increases). The skewness of the number of requests per key leaves the system in an unbalanced state regarding the requests over time.

Number of Requests

Figure 6.6 shows the request rate for the 3 nodes. The load over time is approximately uniformly distributed over the 3 nodes. The first load balancing step at time step 10 has the highest impact on the number of handled requests as the nodes after this first
Figure 6.5: Load information: number of keys. Request rates of a 3 node system, with
the number of keys as the load information. The 3 curves represent the 3
nodes in the system.

step have more or less the final positions the handle the 80-20 request distribution.
Figure 6.7 shows the response times for the above experiment. The response times
are almost constant over the whole experiment except for the time of the first load
balancing step.

6.3 Known Issues

Initially in this section the cloud bursting feature should be benchmarked. As there
are two bugs that couldn’t be fixed in time, Cloudy is very unstable if the number of
requests per node exceed the capacity of the node.

Bugs

- **JVM Crash:** Whenever a node is heavily overloaded over a longer period, the
  node completely crashes. This can even include a crash of the JVM.

- **Broken Message System:** At a "random" point in time, the message system of a
  node seems to be broken and this particular node isn’t able any longer to receive
  TCP messages. This means that the gossiper still works, but the node cannot
  receive any write requests or interact with other nodes during a load balancing
  step.

The mentioned bugs have big influence on the stability of Cloudy as shown in figure
6.8. The figure shows a simulation run starting with 3 nodes and a single client thread.
Every 60 seconds 60 new client threads are added and whenever the average number
Figure 6.6: Load information: number of requests. Request rates of a 3 node system with the number of request as the load information. The nodes have more or less the same load. The 3 curves represent the 3 nodes in the system.

Figure 6.7: Response times for a 3 node system. Note: the y-axis is in logarithmic scale.
of keys per system exceeds a given threshold, the current leader in the system add a
new node (represented by the vertical grey lines). The point in time when a node gets
corrupted is represented by the red dots.

The cloud bursting feature is working, but because of the mentioned bugs the system
does not scale.

Another conspicuousness is the variability that seems to be proportional to the number
of available clients. This has several reasons:

- As the number of requests increases, the number of forwarded messages in the
  system increases. This has the effect that incoming write requests have to wait
  longer to get handled.
- Cloudy periodically flushes memtables to disk.
- A load balancing step has an impact of the number of handled requests per time
  interval.
- Corrupted nodes have a big impact of the number of handled requests as a lot of
  timeouts occur and clients need to reconnect to other nodes in the system.

Figure 6.8: Cloud bursting simulation starting with 3 nodes and an increased number
of client threads per time interval. A data point in the graphic represents
the number of handled requests per second in the whole system (aggregated
over all available nodes). The numbers at the top of the graphic represent
the number of nodes in the system and the vertical lines the point in time
where a new node was fully integrated into the system. The red points are
points in time where a node gets corrupted (isn’t writable).
7 Conclusions and Future Work

7.1 Conclusion

Cloud computing and cloud bursting are new and promising paradigm in the IT world. The ability of using hardware and services on demand and especially pay-on-demand opens great new potentialities. Cloud computing allows even small companies to perform tasks that need a huge network and cloud bursting overcomes the problem of handling short but very large peaks in the system load.

At the time of this thesis only commercial products and solutions exist and the open-source world was still lacking a cloud computing service that is also offering cloud bursting features. Cloudy is the first step into this direction. It is a peer-to-peer like system with a sophisticated load balancing algorithm that is also able to integrate new nodes on demand and release nodes if not needed anymore.

Cloudy uses an order-preserving hash function to map data items to nodes and every node is able to balance its own load. Nevertheless, one node acts as the leader node of the system and is responsible for the dynamic of Cloudy. The leader node is capable of integrating new nodes if the system is overloaded and to remove nodes if they are not longer needed. As an additional feature, the load balancer in Cloudy can be configured to use different load information, e.g. the number of requests or the key distribution. The request rate leads to a better load distribution, but depending on the data to be stored in Cloudy, we are able to choose the appropriate parameter.

The experiments show, that the load balancer implemented leads pretty fast to an uniformly distributed system based on either the key distribution or the number of requests. The cloud bursting feature is also working pretty good as new nodes get fully integrated and the system gets rebalanced.

7.2 Future Work

Experiences with Cassandra

Cloudy is based on Cassandra and uses its storage as well as its messaging system. The development process of Cassandra was in alpha state when Cloudy was started. As the documentation was extremely sparse a lot of the initial work was to read and
adjust parts of the code. Especially the storage system including logging and data movement was very buggy and a significant part of the thesis was to fix several bugs of this multithreaded and distributed system. Cassandra was shortly moved from Google Code to Apache and a lot of changes and bug fixes were added. A next step is to integrate the added features of Cloudy to the new code base of Cassandra to get an overall much more stable system.

**Stability**

Some problems with the TCP thread pool of Cloudy limit the systems scalability. Nodes crash if the load exceeds their capacity and the message system blocks which leads to corrupted nodes. It is also possible that the whole JVM crashes.

**Cloud Bursting**

At the moment Cloudy runs either local without the cloud bursting feature or on EC2 where this feature could be enabled. To also use it in a local environment, the API needs to be adjusted. This means, that it has to be defined how to start a local node given a token.
A Graphics

Boxplot
Boxplots are often used in this thesis to represent response times as the lowest value as well as the median and some kind of a confidence interval is shown for every distribution. The lowest value can be viewed as a approximation for the theoretical value and the median as an approximation for the expectation.

Figure A.1: Simple representation of two normal distributed variables. Group1 has a $\mathcal{N}(1, 2^2)$ and group 2 a $\mathcal{N}(0, 0.7^2)$ distribution. Shown are the median and the 1st and 3rd quartile as well as the upper and lower whisker. The upper whisker is the highest data value smaller than $1.5\cdot\text{IQR}+3\text{rd quartile}$, with IQR the interquartile range.
B Miscellaneous

80-20 rule

The 80-20 rule used in this thesis is a definition of load distribution, which is as follows.
20% produce 80% of the system’s load, where 20% of the 20% produce 80% of the 80% of the system’s load and so on. It is a recursive definition of load distribution. Figure B.1 shows the distribution function of the 80-20 rule for 10000 different keys. Note: the 80-20 rule confirms a pareto distribution.

Figure B.1: 80-20 distribution function. The y-axis is in logarithmic scale.
Bibliography


