Vrije Universiteit Amsterdam

MASTER PROJECT

- Parallel and Distributed Computer Systems -

Resource Provisioning for NoSQL Datastores

Scientific Adviser
Dr. Guillaume Pierre

Author
Eng. Mihai-Dorin Istin
Student No. 2128276

- August 1st, 2011-

Abstract

Popular web applications like Facebook, Google Search Engine or Amazon have to face major challenges. First of all, the volume of data that has to be managed by these applications goes far beyond the limits of traditional data bases. As a reaction, a new generation of very scalable storing solutions, NoSQL data stores, has been design by the major Internet companies. Another important challenge that popular applications have to deal with is represented by flash crowds. In this context efficient resource provisioning is mandatory in order to avoid unpleasant situations like Amazon crashes from 2008 and 2011. This thesis addresses the problem of resource provisioning in the context of NoSQL data stores. First of all, we analyze the impact of different design decisions from the resource provisioning and we present the necessary steps toward building an accurate model for NoSQL solutions. As a proof of concept, our research presents the performance model of Cassandra, one of the most popular NoSQL implementations.

Keywords: dynamic resource provisioning, performance modelling, NoSQL data stores

Contents

List of Figures

Chapter 1

Introduction

Web applications have gained increasing popularity and now they play an important role in our lives. During the last years, the web applications have evolved not only in term of complexity, but also in term of diversity. The static web pages used in the past have been replaced by very dynamic applications enhanced with new features like customization, advanced search or high interactivity enabled by AJAX and Javascript. Thus, today, we use the Internet to chat with our friends using instant messaging applications (GTalk), to share our pictures from the latest trips (Facebook or Picassa) or to buy books or other products from online stores (Amazon).

The expansion of web applications has also led to increasing requirements and harder challenges that have to be addressed. One of the most important issues is represented by the large amount of data that has to be stored and processed. For instance, Facebook has more than 500 million active users that share more than 30 billion pieces of content including web links, news, blog posts, pictures or videos [25]. Moreover, as the application becomes more and more popular it should be able to scale up and to deal with important variations in the load. Maintaining a low response time represents an important aspect, as users have a limited time to spend over the Internet. As there are a lot of alternatives, when they encounter a slow application, they will just switch to another similar solution. This is highlighted in a Forrester Research study published in 2009 shows that 47% of the user expected that a shopping web page loads in no more than two seconds [22].

Designing scalable web applications represents a very important problem especially when we want to have a design that is capable to dynamically adapt its processing capacity according to the variations in load. One of the main challenges is to develop algorithms capable of deciding when is the best time to modify the resources configuration by adding or releasing new resources. The quality of these algorithms is very important not only in terms of resource requirement, and costs (as most of today's solutions are deployed on Cloud platforms), but also in

terms of user experience as provisioning the required resources too late usually leads to a significant increase in the response time. Moreover, as current web applications involve a large number of components/services that have complex interaction models, the resource provisioning problem becomes even more challenging as we also have to determine which component would benefit the most from the provisioning process.

Storing and managing an application's data becomes very challenging as the quantity of information increase very fast. Applications like the Google Search Engine or Facebook can deal with tens of terabytes of information every day. In these conditions the constraints of traditional databases are too strong. Relational databases such as MySQL enforce the ACID properties: all transactions are atomic (*Atomicity*), any transaction will take the the database from one consistent state to another (*Consistency*), there is no concurrent access to the same data (*Isolation*) and the system is able to recover all committed data after a failure (*Durability*). Unfortunately, as the size of the database increases, maintaining these properties comes with important time penalties as traditional databases based on the relational model have not been designed to maintain the quantity of information that are encountered today on applications like Facebook or Ebay.

One of the most important features of traditional databases is the posibility to perform different types of joins based on the relations within tables and between tables. This type of operations reduces the quantity of data to be stored by reducing the need for data redundancy. On the other hand, the cost of join operations is very important as data for multiple tables should be aggregated. As the quantity of data to be stored increases, the size of the tables within the data base also increases and thus the cost of the join operation. For applications that deal with very large quantity of information, join operations are too costly and thus are no longer a viable solution especially due to the very long responce times associate with this operation. Thus most of the current large scale storing solutions trade the convenience of arbitrary complex queries (such as joins) for high performance and scalability.

Given the stringent performance and availability requirements, the designers of large scale web applications like Google have come to the conclusion that respecting the ACID properties and thus having serializable transaction is impractical for general transactions as most of the web applications tend to manipulate only one record at a time [5]. Based on these assumptions, important companies like Yahoo! or Amazon have developed their own highly scalable data store solution (PNUTS [6] and Dynamo [8]), tailored for specific data patterns and requirements. This type of solution known as NoSQL data stores trades the expressiveness of classical databases for scalability, high availability and performance.

1.1 Project Motivation

All work being done on resource provisioning currently focuses on Web applications hosted with application servers and relational (SQL) database servers. Moreover, SQL solutions have not been designed as highly scalable solutions, they no longer represent a good solution for complex web applications that have to manage very large quantities of data. The solution comes from NoSQL data stores that are particularly well-suited for resource provisioning as they are designed to be incrementally scalable when adding computing resources to them.

The goal of this project is to explore resource provisioning in the context of NoSQL data stores. The main questions that this thesis addresses are how to model the performance of a NoSQL data store (keeping in mind that different NoSQL systems will have different performance characteristics), how to calibrate the models, and how to use these models for effective and practical resource provisioning.

Building a high quality model for NoSQL data stores is a challenging objective as this type of solutions is very complex. Most of the existing implementations are proprietary, being developed by the companies like Google, Amazon, Yahoo! or Facebook. The available documentation is often not sufficient to understand all the mechanisms that contribute to the performance and scalability of each particular implementation. In this thesis we have chosen Cassandra, one of the most popular NoSQL solutions that has also become an open source project [19].

Resource provisioning in the context of NoSQL data stores is a very complex process as it also involves a large quantity of data to be transfered from existing servers to the new servers that are provisioned. Existing provisioning algorithms like [17] start the provisioning process when the response time, of the application violates a Service-Level Agreement (SLA). When the response time of the NoSQL data store increases very much, this usually means that the NoSQL servers are overloaded and a few extra clients could crash the system. If we decide reprovisioning at this moment, the transition period is very long, according to [7] can last several hours, and during this period the performance of the data store degrades even more.

1.2 Outline

We start our research by analyzing in Chapter 2 what are the existing approaches for resource provisioning. Then we analyze reasons for moving from SQL database to NoSQL data stores. Moreover, we present the most important trade offs that have to be considered by NoSQL solutions and how these design decisions affect the resource provisioning process. Chapter 3 presents the methodology used in order to build the profile (Chapter 4) and the performance models (Chapter 5) for Cassandra. Next, Chapter 6 presenta the limitations of existing resource

provisioning algorithm and we show how they can be improved in order to be able to deal with NoSQL data stores. Chapter 7 concludes and presents future research directions in the domain of resource provisioning for NoSQL data stores.

Chapter 2

Related Work

More and more web applications have to manage amounts of data that seemed utopian when relational data bases models have been designed. Thus, today's storing solutions have to face new challenges that have not been addressed by the SQL solutions.

As the volume of data to be stored and managed increases, a single data base server is no longer enough. The obvious solution is to split the data into multiple chucks that are spread across multiple servers or even across different data centers. In these circumstances, it is very important to consider the possibility of having network or machines failures and to use replication in order to prevent data loss.

SQL solutions are designed to provide strong consistency of replicated data. This means that all replicas have the most recent update of each data immediate after the commit. Although strong consistency is a very convenient feature for a data store, systems implementing it are limited in term of scalability and accessibility [21]. Moreover, solutions based on strong consistency are not capable of handling network partitions because if the connection to at least one replica is lost, the commit is impossible (CAP Theorem).

For popular web applications like Google Search Engine or Facebook, dealing with network partitioning is something usual. Such systems receive a very high number of requests from hundreds of millions users at peak times and run on infrastructures having several tens of thousands servers spread all over the world. It is extremely important for this type of applications to be able to provide high availability even when they are dealing with network partitions. In this case the consistency provided by traditional databases represents a major inconvenience for dealing with high request rates and very large data volumes.

2.1 Resource Provisioning

2.1.1 Motivation

Major web applications like Amazon have to deal not only with large amounts of data, but also with important variations in terms of load. For instance, if the release of a new version of a popular game or gadget it is announced, Amazon will expect to have a very high number of requests for that product in the minutes after the release. This usually leads to a sudden increase in terms of request rate, phenomenon known as *flash crowds* [11]. In these cases the application must be able to dynamically adjust its configuration to prevent poor user performance or even worse, a site crash. Reconfiguration is done by adding new resources or reallocating existing ones in order to optimize the application performance. On the other hand, when the request rate decreases under a certain threshold a cost-efficient application should be able to release a part of its resources while maintaining a good performance.

The process of dynamically adjusting the computational capacity of a web application is known as dynamic resource provisioning. It aims at maintaining the end-to-end response time of a web application within a fixed Service Level Agreement (SLA). This value usually represents a lower bound for the performance of the application. An SLA can be assigned for each component of the application and the violation of any of the associated SLA triggers the reprovisioning process. A better approach is presented in [17]. Here a single SLA is assigned to the front-end service of the web application and the provisioning process starts once the SLA is violated.

2.1.2 Web Application Models

The performance of resource provisioning algorithms is highly influenced by the accuracy of the assumed architectural model of the web applications. Most of the exiting work on resource provisioning considers only single-tier [10, 1] or multi-tier applications [18, 27, 28]. This assumption holds for an important fraction of the existing web applications, but is not fit for more complex designs. Large web applications have very complex architectures and a single user request usually lead to tens of subsequent services invocations. For instance in Amazon's decentralized service oriented infrastructure, a page request to one of the e-commerce sites typically requires the rendering engine to construct its response by sending requests to over 150 services [20]. These services often have multiple dependencies, which frequently are other services, and as such it is not uncommon for the call graph of an application to have more than one level [8]. The structure of such complex architectures can be represented as a Directed Acyclic Graph (DAG). The resource provisioning solution presented in [17] is proper for web applications having arbitrary complex architectures that can be represented as a DAG where

each node corresponds to a service.

2.1.3 Performance Models

One of the most challenging aspect of dynamic resource provisioning is to decide which service should be de-/re-provisioned such that the whole application will respect the SLA. To address this challenge, for each service is associated a performance model in [17]. The model is capable of predicting the future performance of the service if more machines are provided or if a fraction of the exiting machines are removed. Moreover, a good performance model should be able to also predict the future performance of the application if the request rate increases or decreases. This thesis follows a similar approach and builds a performance model for the service represented by a NoSQL data store.

Resource provisioning for NoSQL data stores is very challenging due to the high complexity of these storing solutions. Building an accurate performance model for NoSQL data stores supposes a deep understanding of the design and trade-offs of these solutions. In the next section we present different NoSQL designs and we analyze their impact from the resource provisioning perspective. To our best knowledge, no performance model exists to capure the profile of NoSQL data stores.

2.2 NoSQL Data Stores

The requirements of the most popular web applications go far beyond the features provided by traditional data bases. SQL solutions are still very good for most of the existing web applications, but are no longer a solution to consider when dealing with large data volumes. As there was not a good storing solution capable of meeting the requirements of successful web applications, most of the important companies like Google [5] or Amazon [8] started to develop they own storage solutions, tailored for the particular properties of their products. This has led to a new trend in data storing solutions, known as NoSQL data stores which often trade the convenience of serializable transactions, strong consistency, relational models and arbitrary complex queries for high availability and fault tolerance in the presence of network partitions.

2.2.1 The CAP Theorem and Consequences

All existing NoSQL solutions, like Google's BigTable [5] or Facebook's Cassandra [19], face the tradeoff from Brewer's *CAP Theorem* [3]. This is one of the most important theoretical result regarding the properties of distributed systems that share data. It states that for such systems it is impossible to guarantee, at the same time, data consistency (**C**), availability (**A**) and tolerance to network

partitions (**P**). Regarding this result, traditional databases trade the tolerance to network partitions for availability and consistency. On the other hand, NoSQL solutions usually trade consistency for high availability and tolerance to network partitions.

2.2.2 NoSQL Architectures and Implications

The architecture of a NoSQL data store and the routing mechanism used for queries that cannot be resolved by the node that receives the request is another important aspect that influences the performance model, as in most cases a large fraction of the incoming requests must be routed to a node that is able to respond to the query. Systems like Bigtable [5] use a master server to control the load balancing, churn and the distribution of the data to the tablet servers that can be dynamically added or removed during the reprovisioning process. The metadata regarding the tablet distribution is stored in the memory of each tablet server and sent to the client. The clients cache the metadata tree and then submit the queries directly to the corresponding server. Thus, the overhead implied by the routing process is eliminated.

Cassandra [19] and Dynamo [8] use a P2P overlay to organize all servers in a ring according to an order preserving consistent hashing. In both cases, each node maintains availability and key range information for all the nodes in the ring. This enables a one step routing, independent of the size of the cluster. The key range and availability information are spread throughout the cluster using a gossiping protocol. The order preserving hashing implies that two keys that are close in topological order, have also close key hashes. This feature makes Cassandra an efficient solution for range queries.

Solutions like Scalaris [23, 24] use a distinct approach for query routing. Here, each nodes no longer maintains the state and ranges for all other nodes in the cluster, but only for a reduced fraction of them. The space complexity for routing information is in this case proportional to $O(\log_2 N)$. On the other hand, the constant time routing is no longer possible and the worst case scenario supposes a number of messages proportional to $O(\log_2 N)$.

From the resource provisioning perspective, it is very important to be able to determine the inter-cluster communication as it directly influences both the response time and maximum throughput of the system. The performance model that we have built for Cassandra highlights the routing overhead as we are going to see in the next chapters.

2.2.3 Tradeoffs in NoSQL Data Stores

Unfortunately, there is no general recipe for building a high-performance NoSQL solution. The performance requirements implied by the large amount of data to manage, lead to many trade offs that must be addressed and thus, important

decisions in the design of each solution. Each NoSQL data store is specifically designed for a certain type of applications based on their access patterns and unique characteristics. Some web applications like financial applications need coherent data at all replicas to avoid possible losses caused by duplicate transaction. On the other hand, on Facebook it may be acceptable that not all friends of a user are able to see his new status immediately after an update.

In this section we present the most important trade offs that NoSQL data stores address. An overview of the different design decisions taken by some of the most popular solutions is presented in Figure 2.1.

Strong Consistency versus Eventual Consistency

Most of the existing NoSQL solutions use a weak consistency model known as eventual consistency. This allows for updates to be propagated to all replicas asynchronously. Thus, an application that requires an *update()/put()* query from the data store can receive the response before the new value has been propagated to all replicas. Although this allows fast updates, it comes with an important drawback in the fact that a subsequent *get()* request can return a value that is not the latest update for the given key. Eventual consistency guarantees that the latest update will eventually be propagated to all replicas. However, under certain failure scenarios like network partitions, the convergence time of an update might be longer that the usual bound [8].

Eventual consistency is also the approach chosen by the designers of Cassandra [19], the NoSQL data store that we will analyze in this thesis as a proof of concept regarding the resource provisioning in the context of NoSQL data stores. Another NoSQL solution based on this model is Dynamo [8]. On the other hand, solutions like Bigtable[5] and Scalaris [24, 23] implement strong consistency. Bigtable is built on top of the Google File System (GFS) [13] that ensures consistent updates, while Scalaris uses the Paxos Commit protocol.

Latency versus Durability

Deciding when an update should be written to disk represents an important design decision for a NoSQL data store. As the latency of writing an update to disk is very high, many implementations trade the safety and convenience of immediate disk commits for a faster solution that keeps the updated data in memory and eventually writes multiple updates in a row. The main problem of these approaches is that, when a node crashed, all updates that have not been written on the disk are lost.

Many NoSQL data stores like PNUTS [6] and Big Table [5] trade durability for latency because they do not afford to lose valuable data. On the other hand, HBase [16] prefers low latency. A very interesting approach of this trade off is

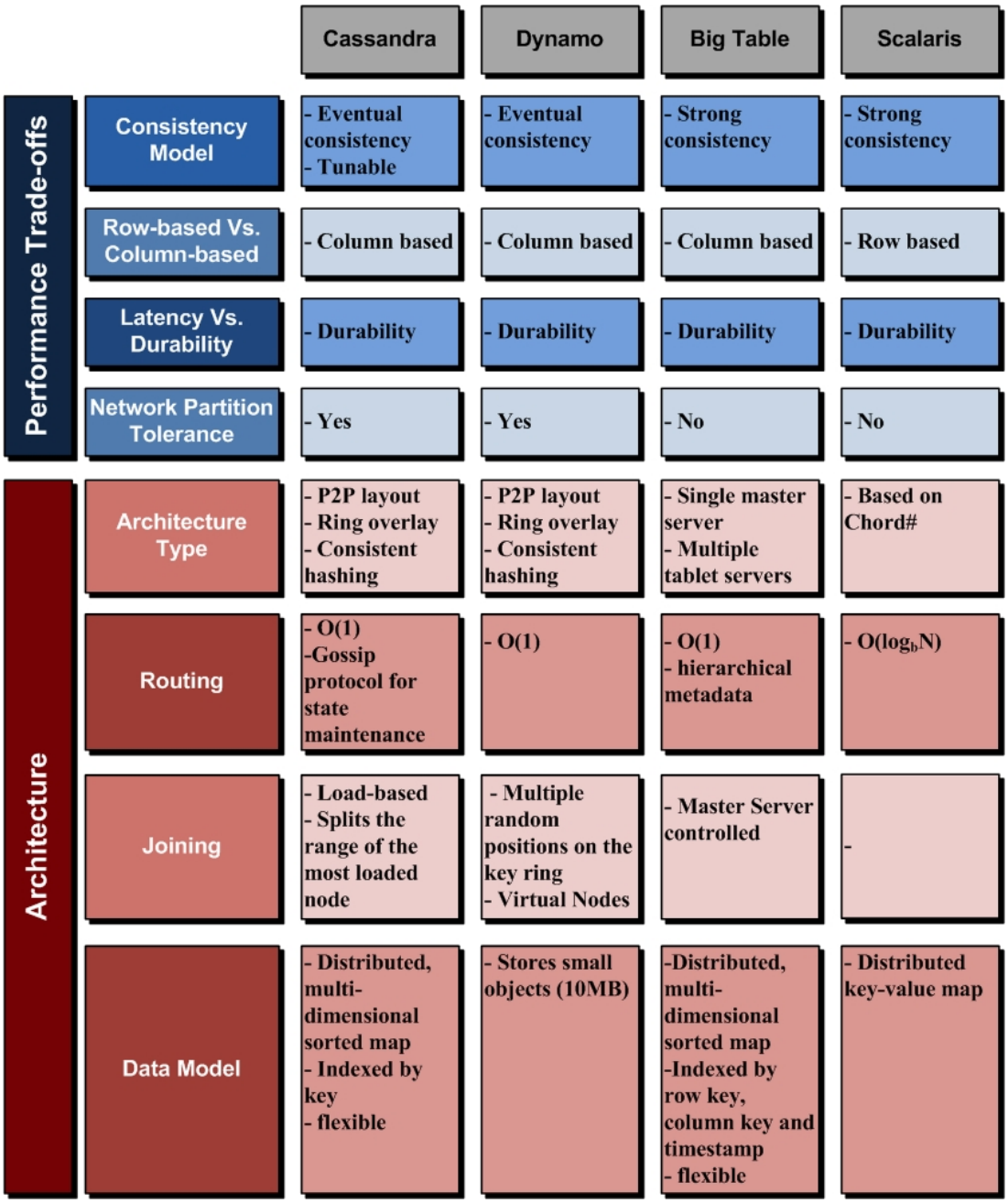

Figure 2.1: Comparison of Cassandra, Dynamo, Big Table and Scalaris

encountered in Cassandra, where the user can tune the data store such that it best fits the requirements of the application.

Column versus Row Based Storage

Current NoSQL systems use two main types of data organization: row based or column based. Row based systems like PNUTS offer access methods (get/put) only for an entire row[6].This type of solutions stores all columns associated with the same key in a contiguous zone on the same disk. On the other hand, solutions like Cassandra [19] or Big Table [5] follow a column based strategy which enables access no only to the entire row, but also to single columns or different groups of columns. These solutions can group the columns that are generally requested together (similar to supercolumns in Cassandra). Moreover, these groups are stored in the same place to optimize the access latency.

2.3 Impact of Dynamic Provisioning

In the case of NoSQL data stores, triggering a resource provisioning process based only on the violation of a front-end SLA might lead to poor performance if the data store is heavily loaded. This happens because adding a new node to the data store implies, at first, transferring a large amount of data from the most loaded node to the new comer, as it happens for Cassandra [19] and PNUTS [6]. Other approaches like HBase [16] do not transfer data to the new node until a compaction occurs. This leads to short transition times, but the new node is not used at maximum capacity from the beginning.

In solutions like Cassandra [19] or PNUTS[6], the source of the data transfer is a node that is already overloaded. The additional computation caused by the transfer usually leads to very long transition times that can be in the order of hours for Cassandra [7]. Here, adding a new node to a Cassandra cluster containing 5 nodes, the systems still was not stabilized after 350 minutes. The clients had to be stopped to allow the system to complete its repartitioning. During the transition time, the overall performance of the system is worse than the performance before provisioning. During a similar experiment PNUTS stabilizes after about two hours [6]. On the other hand, HBase stabilizes after only a few minutes [16]. The goal of our performance model is to determine *when* to reprovision the system in order to minimize the transition time and impact over the performance of the application.

Another important aspect is that for solutions like Cassandra, the position of the new node in the ring is randomly selected at the cluster creation time. This usually leads to highly un-balanced load distribution. As the size of the cluster dynamically grows, the position of new nodes is computed such that the range corresponding to the most loaded node to be split. Thus, the system is capable

of self balancing after a certain amount of time.

As we will see in the next chapters, different design decisions, like the ones presented here, have a direct impact over the performance model of the system. The performance model that we have built for Cassandra follows closely the system behaviour and highlights design decisions like system architecture, routing cost or replication strategy. The proposed performance model, is capable of predicting the behaviour of the data stores with an error no higher than 5% for up to 30 data store servers.

Chapter 3

Methodology

This chapter presents the methodology used to analyze the resource provisioning for NoSQL data stores and to build a performance model for a specific NoSQL implementation. As a proof of concept we have chosen one for the most popular open source NoSQL implementations - Apache's Cassandra [4]. We start by presenting the main reasons for choosing Cassandra. Then we highlight the most important design details of Cassandra and how they impact the dynamic resource provisioning process.

In the second section of this chapter we present the NoSQL benchmark used to build the runtime profile of Cassandra. Next, we introduce the framework we have developed to integrate and synchronize Cassandra and the benchmarking tool. The next part of chapter presents the experimental setup used to profile Cassandra, to build and to test the performance model that is introduced in the next chapters.

3.1 NoSQL Data Store

3.1.1 Motivation

During the last years NoSQL solutions have been developed by some of the most important internet companies, but only a part of the implementation details have been made publically available. As a response, the open source community has developed new solutions that try to mimic the behavior of proprietary implementations. One of the most important results, Apache's Cassandra is based on the Facebook solution presented in [19]. Other similar approaches are encountered in the case of HBase [16] that follows the design decisions of Google's Bigtable [5]. Thus, the first step in deciding which NoSQL implementation to use as a proof of concept, was to analyze the open source implementations as they offer, on the one hand, a complete implementation of the NoSQL data store and, on the other hand, more detailed implementation insights.

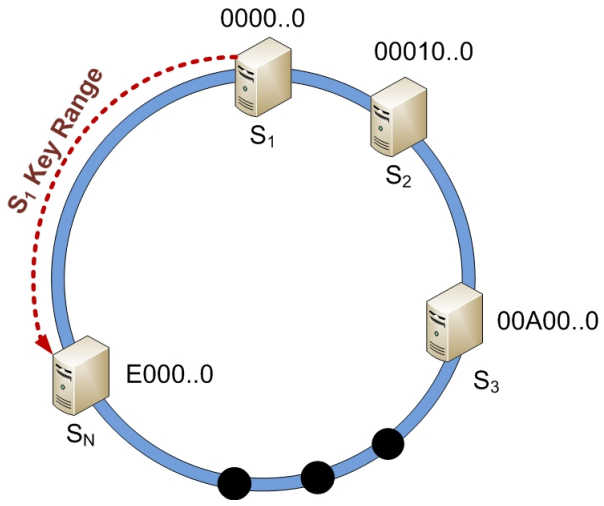

Figure 3.1: Cassandra Key Space Organization

Another important aspect that we have taken into account in deciding which NoSQL data store to use has been the popularity of different open source solutions. From this perspective, Apache's Cassandra [4] is one of the best choices as it is currently used not only by Facebook, but also by many other web applications like Digg [26] or Cisco [4]. Moreover, in 2010, Twitter has decided to replace their shared MySQL and memcache storing solution with Cassandra [2].

The hybrid design used by Facebook to design Cassandra [19] has also been an important argument to support our decision. As Cassandra inherits many aspects from both Dynamo [8] (the P2P architecture, eventual consistency model, gossip based discovery, constant time routing) and Bigtable [5] (data model), an accurate performance model for Cassandra, can be extended to other similar solutions. Moreover, the convergence problems of Cassandra, when dealing with dynamic resource provisioning for overloaded nodes make the designing of a good performance model an even more challenging problem [7].

3.1.2 Cassandra Overview

Cassandra uses a P2P architecture, similar to the one used by Dynamo. The nodes are organized in a ring overlay topology similar to the one presented in Figure 3.1, where the position of each key is computed using a hashing function. The function used for hashing ensures an order preserving consistent hashing. This allows Cassandra to obtain very good performance for range queries.

Each node in the systems is assigned a random number within the key space representing its position in the ring. Each data item is uniquely identified by a key and it is assigned to a node by hashing its key to determine its position on the ring, and then walking the ring clockwise to find the first node with a position

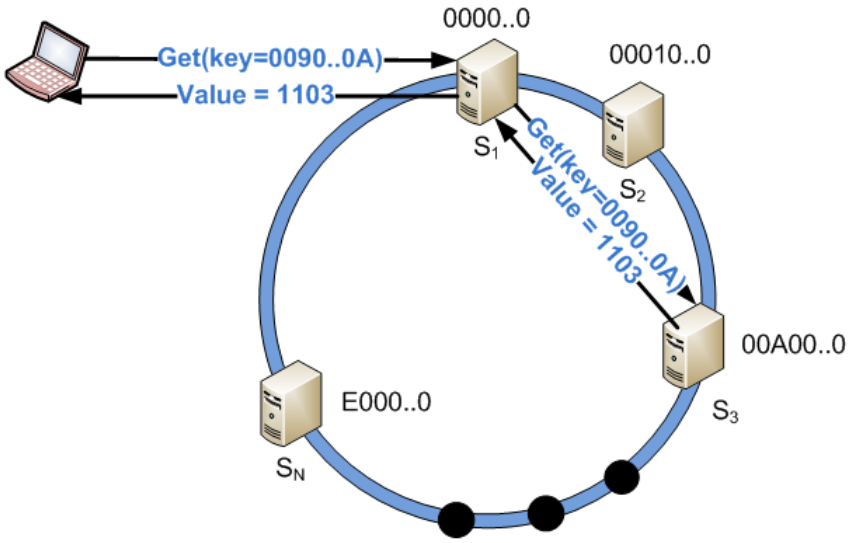

Figure 3.2: Cassandra constant time routing mechanism

larger than the item's position [19]. Thus each node is responsible for the region in the ring between it and its predecessor node. For instance, in Figure 3.1 node S_2 with ID 00010...0 is responsible for all keys having the corresponding hash in the range [0000...0, 00010...0].

By using consistent hashing, Cassandra has the advantage that a node failure/departure only affects its successor and not other nodes from the ring. For instance, in Figure 3.1 if node S_2 fails this only affects node S_3 which will also be responsible for all keys in the range corresponding to node S_2.

Cassandra also inherits from Dynamo the gossiping protocol used to spread the state information of all nodes. This allows each node to have a complete overview of, not only the state of all other nodes, but also of what the key range corresponding to each node is. Thus, a request that arrives to a node that is not responsible for the range containing that key, is immediately forwarded to a node that can resolve the request (constant time routing). For instance, in Figure 3.2 the request sent by the client to server S_1 contains a key that is not in this server's range. Thus, server S_1 determines, base on the key ranges table, which node(s) are responsible for the key requested by the client and forwards the request to that server, S_3 in this case. When S_1 receives the response from S_3, it forwards the message to the client.

The performance model of Cassandra that we propose in the next chapters relies on the routing mechanism of cassandra to compute the performance penalty caused by the requests that have to be routed to the corresponding nodes.

The mechanism used by Cassandra for adding new nodes to the existing cluster is extremely important from the dynamic resource provisioning perspective.

Cassandra monitors the performance of each server and, when a new node is added, it is assigned a position in the ring such that the range of the most loaded node is split. This approach guarantees that the most heavily loaded node will eventually hand off half of the keys it is responsible for. The newcomer becomes responsible for these keys and thus the performance of the system is improved. On the other hand, the data transfer from a very loaded node to the new comer also increases the load of the source node. This leads to very long transition times until the system stabilizes. During the transition phase, the performance of the whole system degrades. Knowing the impact of adding new nodes is crucial from the dynamic resource provisioning perspective and it directly influences the performance model as the provisioning must happen before the system becomes too loaded.

3.2 Benchmarking Application

The first step in building a performance model of a NoSQL data store is to analyze the behavior of the chosen solution (in our case Cassandra) by building its performance profile. To achieve this objective, we need a benchmarking tool which is able to simulate the behavior of users that interact with the data storing component of a web application. A good NoSQL benchmarking solution has to be able to generate tests that can highlight different characteristic of the application that we want to profile.

3.2.1 Requirements

A good performance profile of a NoSQL implementation should be able to determine the behavior of a specific application in different environments. Most of the current NoSQL solutions are optimized either for read operations or for write operations. Thus, it is necessary to analyze the performance of the data store for multiple workload mixes having different read/write ratios. Moreover, a web application does not always have to face the same request rate. Thus, the benchmarking tool should be able to generate different request rates.

The access patterns are different from one application to another. Many web applications have a reduced set of records that are very often requested (*zipfian* request distribution). This type of behavior is usually encountered in online stores where the products that have special prices or discounts are requested and bought by more clients. Other applications might have a large fraction of the requests for the most recent records. For instance, in the case of a news application, the most recent articles are typically accessed by most of the visitors. Last, but not least, some applications may follow an uniform request distribution. Taking all these into account, an important characteristic that a NoSQL benchmarking tool should have is the support for different request distributions.

Another important requirement that a benchmarking tool for NoSQL data stores should meet is providing a fair comparison between different implementations. As we have seen before, the requirements of web application can be very different and, typically, the performance of each NoSQL solution is presented only for the type of workloads that it has been tailored for. Thus, it is very difficult for a user to decide what is the best option when he has to choose a NoSQL solution for his own application.

3.2.2 Yahoo! Cloud Serving Benchmark

Yahoo! Cloud Serving Benchmark (YCSB) [7] represents the first benchmarking solution, for NoSQL data stores, that is capable of producing detailed comparison between some of the most popular NoSQL solutions including Cassandra [19], PNUTS [6] and HBase [16].

One of the reasons for choosing YCSB for our work is that it has already integrated Cassandra in the framework. It is important to mention that the results presented in the YCSB paper [7] for Cassandra have been conducted on the 0.6.1 version while this thesis analyzes the performance of Cassandra 0.7.0 which was the latest version at the beginning of our work.

Another advantage of YCSB is that it provides a very flexible framework that allows one to customize the parameters of the tests when analyzing the performance of a specific NoSQL implementation. YCSB provides two benchmarking tiers. The first tier focuses on the performance of NoSQL data stores while the second one analyzes their ability to scale.

The performance tier of YCSB aims to characterize the trade off between requests latency and throughput for a give hardware configuration. Basically, the benchmarking framework measures the latency as the throughput is increased, until the point at which the data store is saturated and the throughput stops increasing [7]. This type of profiling is extremely important as the dependency between the latency and the throughput represents one of the key aspects of the model we aim to build.

On the other hand, the scaling tier examines the ability of NoSQL data stores to scale elastically by analyzing the performance impact of adding new machines. Two distinct scenarios can be derived here: how does the database performs as the number of machines increases (*scaleup*) and more interesting, how does the database performs as the number of machines increases while the systems is running (*elastic speedup*).

Moreover, YCSB meets all our requirements in terms of request distributions and workload mixes. The user can select the requests distribution from multiple options including zipfian distribution and uniform distribution. The same happens in the case of workload types as YSCB allows its user to tune the read/write ratio and it also supports range queries.

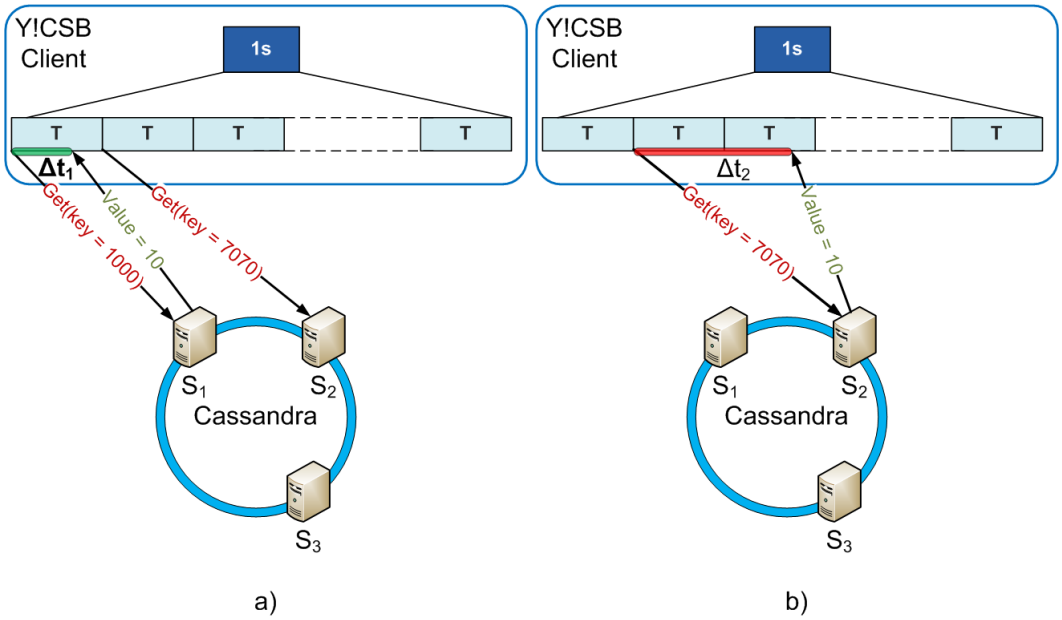

Figure 3.3: YCSB Client request generation for target throughput below (a) and above (b) the maximum throughput of the Cassandra cluster

Taking all these into account, YCSB fits our requirements for a good benchmark tool for NoSQL data stores. We have used it throughout the whole profiling and evaluation process.

3.2.3 Request Generation

One important aspects of the Yahoo! Cloud Serving Benchmark that we have to understand is the mechanism used to generate the requests to Cassandra during each test. YCSB allows users to tune, through a configuration files, each workload. The parameters that can be used include the number of records to be injected in the data store, the number of requests to be sent during the test and, last but not least, a target request rate. The last parameter specifies what the desired request rate is. Moreover, it sometimes happens that this target value cannot be achieved.

For a specific request rate specified in requests per second, a YCSB client splits each second of its run time into equal time slots and tries to send a request at the beginning of each time slot as we can see in Figure 3.3. The number of time slots is equal to the target number of requests per second. Thus, the size of each time slot can be expressed as:

$$T = \frac{1}{TargetThroughput}.$$

If the response for the most recent request arrives within its time slot ($\Delta t_1 < T$) as it happens in Figure 3.3a, the client sleeps until the end of current time slot and sends a new request at the beginning of the next time slot. Moreover, if the response does not arrive within its time slot ($\Delta t_2 > T$ in Figure 3.3b), the YCSB client waits until the response is received and, only then, it sends the new request. In this case, if most of the responses arrive after their slot has elapsed, it is highly probable that the client cannot achieve the target request rate.

As we are going to see later, each Cassandra cluster configuration has a maximum throughput that can be obtained. If the request rate is lower than this value, then most of the messages arrive in its corresponding time slot and the YCSB clients are able to generate the target request rate. On the other hand, for request rates higher than the maximum throughput, the time slots can no longer be respected and the resulted request rate is lower than the target request rate. Typically, in this case, the YCSB client stabilizes at a request rate equal to the maximum throughput of the Cassandra cluster.

3.3 Cassandra and Yahoo! Cloud Serving Benchmark Integration

Building a good profile of Cassandra in our case supposes a large number of tests to analyze different runtime aspects of the data store. To efficiently run the necessary tests, we have designed and implemented a synchronization mechanism between the Cassandra cluster and the YCSB clients as shown in Figure 3.4.

A *Test Coordinator* is responsible for running a set of tests according to some parameters specified in the configuration file. Each test from the set is started by initializing a *Cassandra Master* which is further responsible for synchronizing all Cassandra servers and YCSB clients. Once the test is completed, the *Cassandra Master* sends a message to the *Test Coordinator* and then it shuts down. After receiving the completion message the *Test Coordinator* starts a new test from the set, until all tests are completed.

Synchronization between Cassandra instances is very important. A Cassandra instance should wait a certain period of time after a previous instance has started. The recommended time span between the initialization of two different Cassandra server is 120 seconds. These are necessary for each new comer to connect to the seed node, choose its random position from the ring, and initiate the Gossip protocol to receive the ranges and availability of all other nodes in the ring. During the the same time, the new node spreads its local information (IP, position in the ring, etc) to the other nodes.

The synchronization between Cassandra servers is done by the node hosting the first Cassandra instance. It uses a special thread, *the Cassandra Master*, that is responsible for starting the other servers after the necessary initialization time has past. After each test that we perform, the *Cassandra Master* announces the

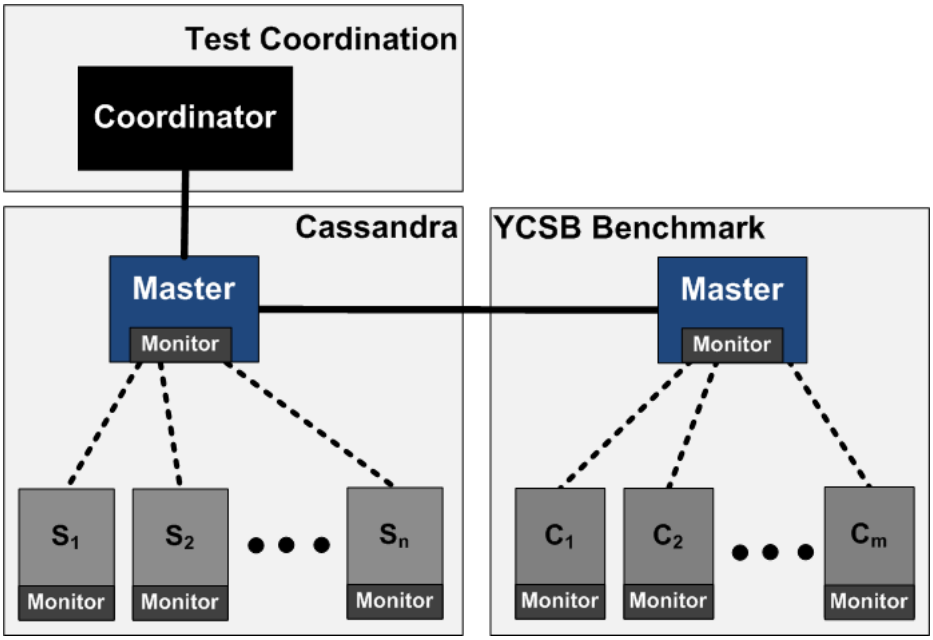

Figure 3.4: Testing Framework

tests completion to all Cassandra servers such that they can shut down. It is important to mention that after each test we start another Cassandra cluster. This happens because each workload contains not only read requests, but also updates and the initial data base is altered. Thus, we want each test to be performed on the same set of data in order to have coherent results.

During each test, the *Cassandra Master* starts a *YCSB Master* which is responsible for the test preparation, test running and finally for announcing the *Cassandra Master* about the test completion. Once *YCSB Master* starts, it load the data set corresponding to the current test into Cassandra data store and then, it starts all YCSB clients that are necessary to perform the current test. Each client sends a message to the coordinator once the test is completed and then shuts down. Once the *YCSB Master* receives the completion message from all YCSB clients it sends a completion message to the *the Cassandra Master* and shuts down.

3.4 Experimental Setup

The experiments conducted to profile Cassandra and to evaluate the proposed performance models have been conducted in a homogeneous environment from Vrije Universiteit Amsterdam (VU), part of the DAS4 supercomputer (Distributed Ascii Supercomputer 4 [9]). DAS4 consists of six clusters spread all over the Netherlands. The cluster from VU has 74 dual quad-core processors having

2.4GHz and 24GB of memory. The communication is built on top of two networks: an Ethernet network for inter-node communication and an InfiniBand used by distributed file system.

The methodology presented in this chapter is used to build the performance profile that we present in the next chapter.

Chapter 4

Performance Evaluation of Cassandra

Building a performance model for a NoSQL data store supposes first of all a deep understanding of the design decisions and runtime profile of the chosen system. Taking these into account, we have used the profiling framework presented in the previous chapter in order to build a performance profile for Cassandra.

We start by analyzing how the number of messages exchanged within the Cassandra cluster varies as the number of servers increases. Further, we investigate how the throughput of Cassandra varies as we increase the request rate and as we increase the number of servers. Last but not least, we build the read and update latency profile for Cassandra.

4.1 Network Traffic Profile

One of the first steps towards the understanding of the behavior of Cassandra is to analyze the quantity of traffic exchanged during each test. Understanding the traffic patterns represent the step towards understanding Cassandra's mechanisms and then building an accurate performance model. As we are going to see in the next chapter, the model that we have built for the traffic directly influences the maximum throughput model. This happens because the additional messages generated when the cluster size increases represent a performance penalty that must be considered in our maximum throughput model.

Measuring the traffic generated within a Cassandra cluster for a specific workload is done by a monitoring component that we have attached to each Cassandra server (including the *Cassandra Master*) and to each YCSB client (including the *YCSB Master*). The monitor component inspects the traffic received/sent on both the Ethernet and InfiniBand networks of each DAS-4 node and uses log files to store the results. Monitoring the YCSB machines is necessary in order to compute the communication within the Cassandra cluster. Basically, all messages

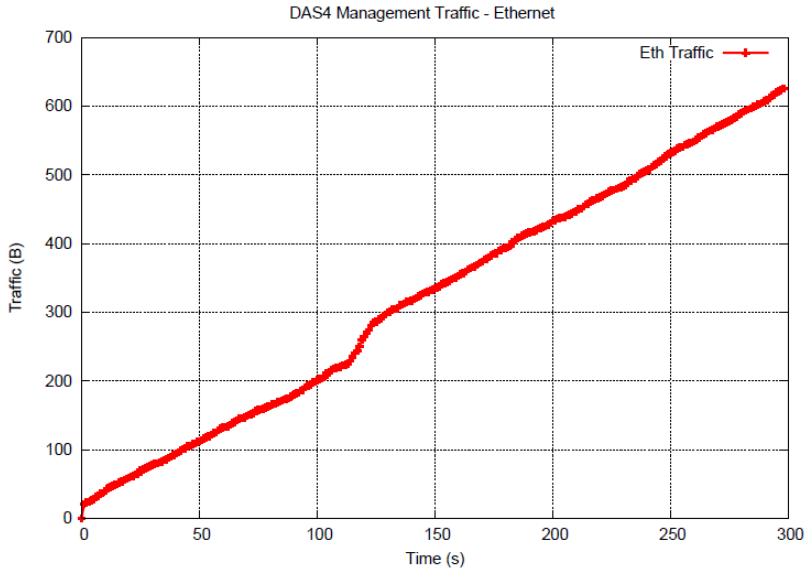

Figure 4.1: Management Traffic Profile for DAS-4 on Ethernet

that are sent by YCSB clients are received by Cassandra servers and all messages received by YCSB clients have been sent by Cassandra servers. Thus, we can subtract the traffic measured at YCSB machines from the traffic generated by Cassandra servers and we obtain the inter-cluster traffic.

Another source of traffic that we have to take into account is represented by DAS-4 management traffic. In order to estimate the quantity of management traffic we have run only the monitoring components on several DAS-4 nodes. The results presented in Figure 4.1 show that this type of traffic is almost insignificant as on the Ethernet network are typically sent, on average, a bit more than 2 Bps. On InfiniBand the amount of data sent/received is even lower - 0.5 Bps (see Figure 4.2).

In Figure 4.3 we present the total traffic received by a Cassandra cluster during a workload containing 1000 requests, as we increase the size of the cluster. We have to notice that management traffic presented in Figure 4.1 is very low compared to the traffic generated by the test. This means that the management traffic can be ignored in our model.

In the case of one server, the measured traffic represents just the requests coming from the YCSB client. This happens because the server is responsible for all requests and no request routing is necessary. Moreover, the gossip protocol presented in the previous chapter is not used in this case as, the server does not have to be aware of any other node.

As the size of the cluster increases we start to have more and more intra-cluster communication. A fraction of the incoming requests, $1 - \frac{1}{n}$, at each node cannot

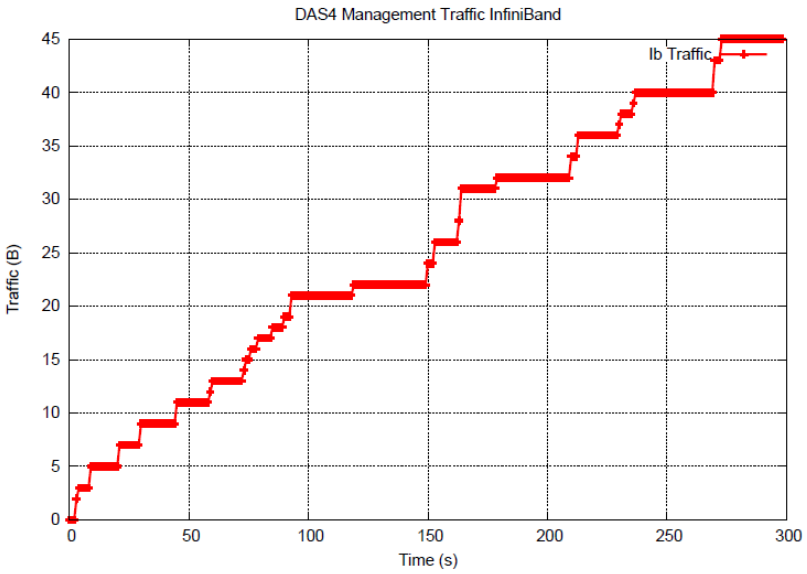

Figure 4.2: Management Traffic Profile for DAS-4 on InfiniBand

be resolved locally and they have to be routed to another node. On the other hand, the gossiping protocol starts to spread the node availability and key range information. This explains the 100% rise in terms of traffic when the Cassandra cluster evolves from one server to two servers. As we can observe in Figure 4.3, the traffic increases with each additional node. For 18 servers, the volume of traffic is 4.5 times larger than in the case of a single server.

Based on these information the next chapter introduces, a performance model for the generated traffic and we also analyze the impact of the traffic over the maximum throughput of a Cassandra cluster.

4.2 Throughput Profile

A throughput profile is very important from the resource provisioning perspective as it shows what is request rate value that overloads the server. In fact, each server is capable of serving a certain number of requests per second. After this point, the server produces approximately the same throughput. At the same time, the response time increases as the request can no longer be processed on the flight, but have first to wait until the server processes all requests that have come before.

The maximum throughput is not a fixed value but it is influenced by the type of workload and the size of the data base. For instance, a workload containing range queries overloads the servers faster than one containing single key read and write operations.

The setup we have used to build the throughput profile is based on a database

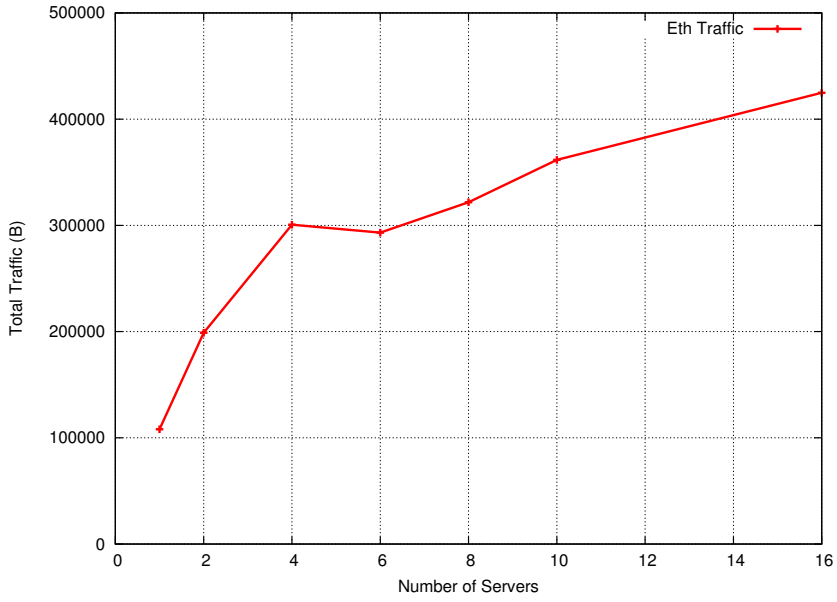

Figure 4.3: Traffic send/received by a Cassandra Server

containing 500,000 records. We used a workload containing 50% read operations and 50% write operation. To build the profile we have varied the target request rate from 10,000 to 512,000 requests per second.

For the profile presented in Figure 4.4 we have used a Cassandra cluster containing three servers. We have to specify that a similar profile has been created for multiple cluster configuration varying for 1 server to 20 servers and the profile of each configuration is similar to one profile presented in the figure.

It is important to notice in Figure 4.4 that for request rates lower than 73,000 requests per second, the throughput of the data store is approximately equal to the request rate. This means that the server is not overloaded and it is able to serve the requests as they come. After this point, the requests start to be delayed.

For the considered setup, we have set a threshold at 73,000 (the green horizontal line in Figure 4.4) and this represents the maximum throughput that can be achieved with three Cassandra servers. We have decided to set the maximum throughput to the lowest value that we have encountered after the data store is overloaded. For some request rates we can obtain higher values, as it happens in the case of 240,000 requests per second where the throughput is 90,000. The reason for these results is that Cassandra servers do not have equal ranges of keys from one test to another and this typically leads to unbalanced load between servers. This happens because each test (having a new request rate) is run on a new cluster, the position of each node changes as it is randomly chosen by each server at initialization.

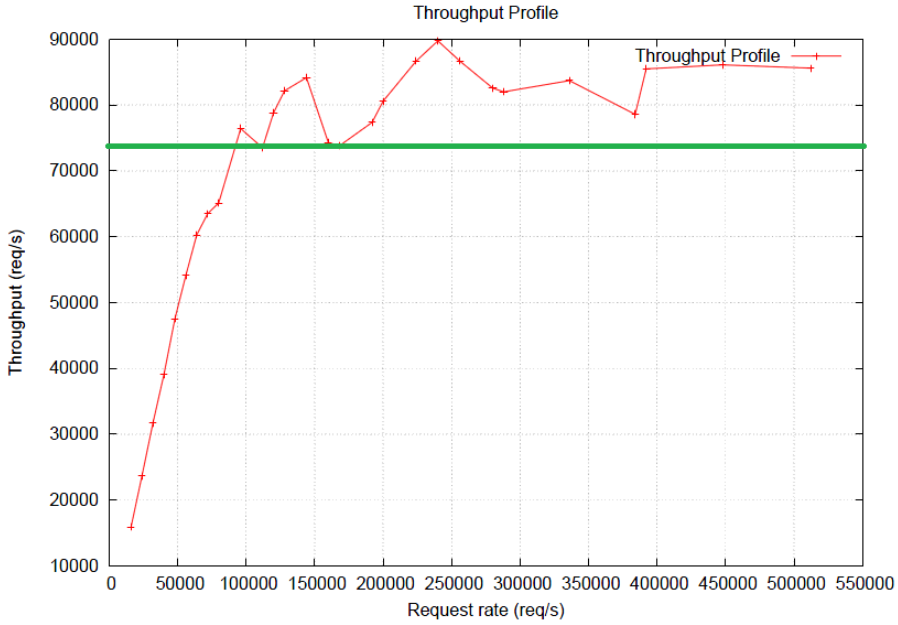

Figure 4.4: Throughput Profile of a Cassandra cluster with 4 servers

4.3 Maximum Throughput Profile

The next step in understanding the behavior of Cassandra is to build the maximum throughput profile. This profile is very important for the resource provisioning perspective, as it shows what is the maximum throughput we can achieve with a specific Cassandra configuration (or cluster size). Moreover, we can use the maximum throughput profile to determine what will be the throughput of the system if we add (or remove) one or multiple servers.

Building the maximum throughput profile supposed the analysis of the throughput profile for different Cassandra cluster sizes in order to determine the maximum throughput that can be achieved for each configuration. Based on these values, we have identified the maximum throughput profile that is presented in Figure 4.5.

It is very important to understand what happens when we move from a cluster having a single server to a cluster with two or more servers. As we can see in the Figure 4.5, the maximum throughput for a single server is approximately 95000 requests per second. When we add the second server we observe a 40% drop to about 55000 requests per second. This happens due to the overhead implied by request routing and information spreading gossiping protocol.

When we have a single server, all requests are resolved by the same server, as it is responsible for all keys from the data stores. Moreover, the gossiping protocol used to distribute the state and key range associated with each node is

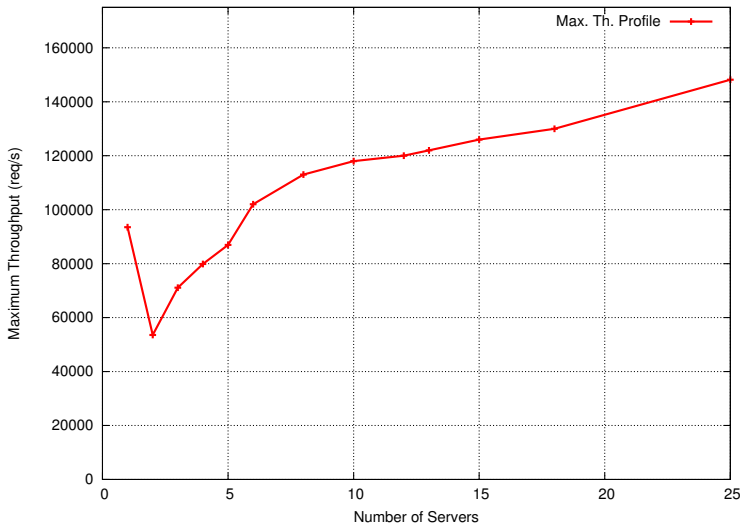

Figure 4.5: Maximum Throughput supported by a Cassandra Cluster depending on the cluster size

not necessary in this case. On the other hand, when the cluster size increases to two servers, each server is responsible for half of the keys. Thus, it can no longer respond to all requests and 50% of them are redirected to the other server. Moreover, for a cluster size of two or more servers, the gossiping protocol starts to distribute the state and key range information.

As the Cassandra cluster size increases, the overhead implied by the request routing and information spreading protocol increases. According to the network traffic model that we have presented the previous section, the network traffic for the same number of requests, increases more than four times when we increase the cluster size from 1 to 20 servers. This behavior is also reflected in Figure 4.5. We can observe that the gain obtained, in terms of throughput, when adding an extra machine is higher for small clusters then for clusters containing many servers. For instance, when we add a machine to a 2 server cluster, we gain more than 15000 requests per second. In contrast, we have to add 5 more machines to a Cassandra cluster having 18 servers to gain the same amount of extra requests per second.

4.4 Read/Write Latency Profile

The read or write latency profile analyzes how the response time of a Cassandra cluster varies when we increase the request rates. To build this profile we have chosen to use a Cassamdra cluster having 3 servers. The total target throughput that we have used for our YCSB clients varied from 12,800 to 128,000 requests

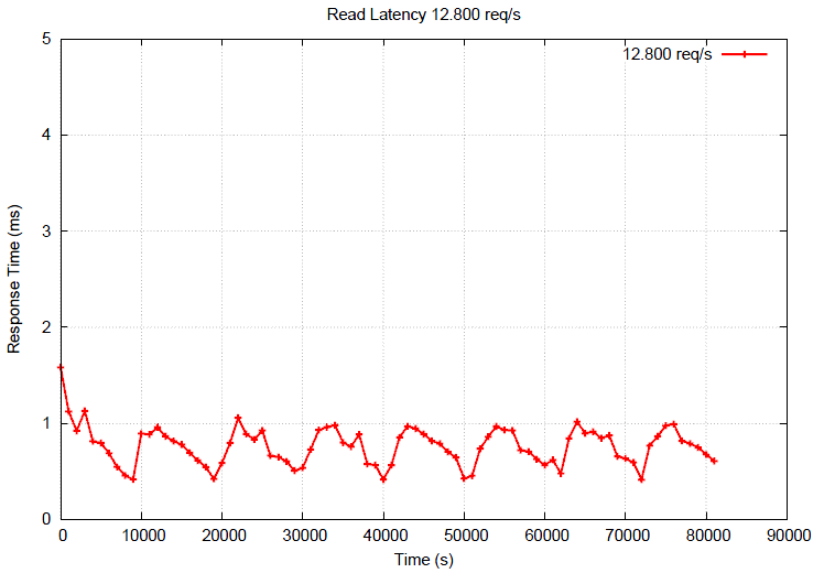

Figure 4.6: Read latency for request rate below the maximum cluster throughput

per second. We have chosen this range of target request rate in order to include the maximum throughput point found at around 73,000 requests per second. Moreover, we have used a data store containing 500,000 records.

The first step in building the read latency profile is to analyze how the response time for read requests varies over time. For this, we are going to present the most significant two profiles corresponding to the minimum (12,800) and the maximum target request rate (128,000).

The first profile setup uses 3 YCSB clients to generate a request rate lower than the maximum throughput of the Cassandra cluster. In this case we generate 12 800requests per second. Figure 4.6 shows that the response time for read operations is very low, being less than 1 millisecond.

The second scenario show more interesting results. For this scenario, the target throughput of the YCSB clients is set to 128,000 requests per second, which is much more than the maximum throughput of the servers. In this case, we observe in Figure 4.7 that at first Cassandra is completely overwhelmed and the response time increases very fast to more than 100 ms which is more than 100 times higher then in the previous case. After a few seconds the response time starts to decrease and at 6 second it is about 50ms. Then, after another 5 seconds, it stabilizes to a narrower range [2.5 ms, 3.5ms].

This behavior could easily be explained if we look at the average maximum throughput generated by the YCSB clients. In fact, even if we set the target request rate to 128,000 requests per second, YCSB clients cannot generate more than 75,000 requests per second. Thus, at first, the clients start to send the request at 128r000 requests per second rate, but Cassandra cannot handle so

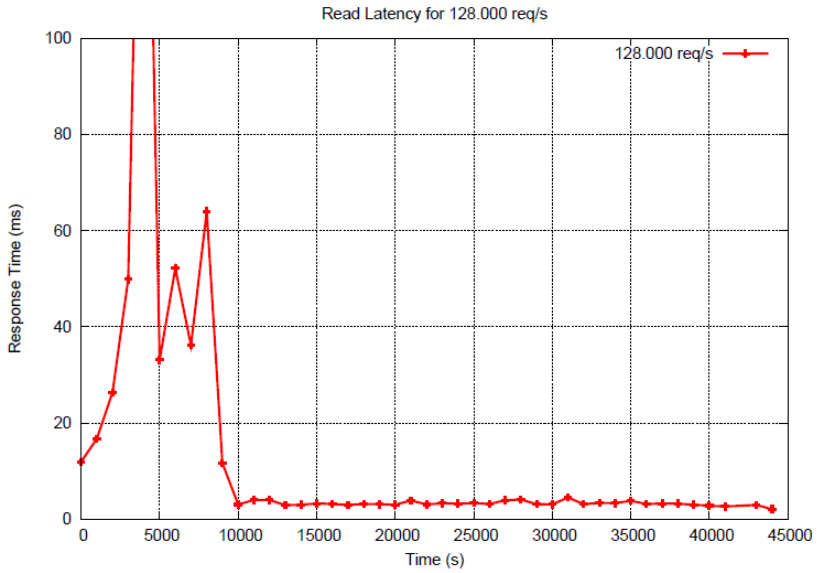

Figure 4.7: Read latency for request rate below above the maximum cluster throughput

many requests, so the responses are delayed. As YCSB clients do not send the next request before they have received the response for the previous request, the generated request rate decreases until it reaches the maximum throughput of the configuration. If the YCSB clients had been able to generate the target request rate, Cassandra would have crashed.

This limitation of YCSB is very important as we can neither find the exact point where Cassandra is completely overwhelmed, nor can we analyze the behavior of Cassandra in the vicinity of this point.

We also analyzed the update latency for the same scenario results. The only important difference is that the stable update latency for updates is a bit lower than the read latency. The results are presented in Figure 4.8 for target request rate lower than the maximum throughput of Cassandra. For target request rates higher than the maximum throughput the results are presented in Figure 4.9.

Last, but not least, we are interested in analyzing how the read/update latency varies as we increase the request rate. In order to achieve this objective so we have built the profile for different request rates placed in the [12 800, 128 000] requests per second range. The results for read operations are presented in Figure 4.10. The profile obtained for update operations is similar.

As discussed before, we observe that the response time is very stable for request rates below the maximum throughput, but it increases very quickly around the maximum throughput point. This behavior is very important from the resource provisioning perspective as it highly influences the resource provisioning decisions. A good resource provisioning algorithm must supply new machines to Cassandra

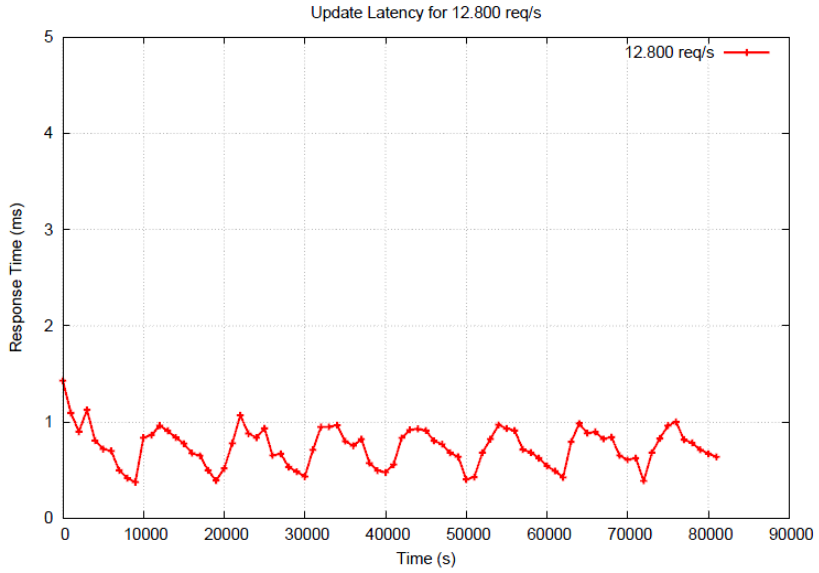

Figure 4.8: Update latency for request rate below the maximum cluster throughput

before the cluster is saturated. In the next chapters we will analyze how this behavior influences the resource provisioning algorithms and how we can adapt them to handle NoSQL data stores similar to Cassandra.

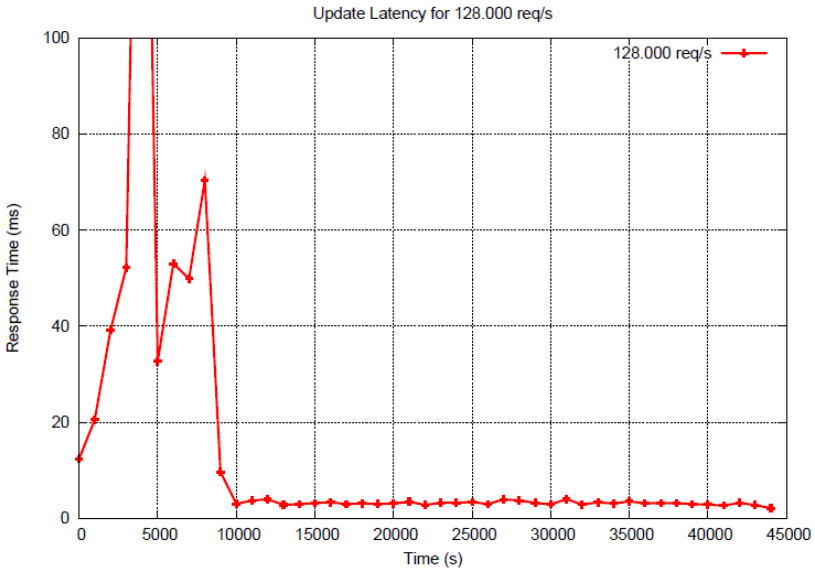

Figure 4.9: Update latency for request rate below above the maximum cluster throughput

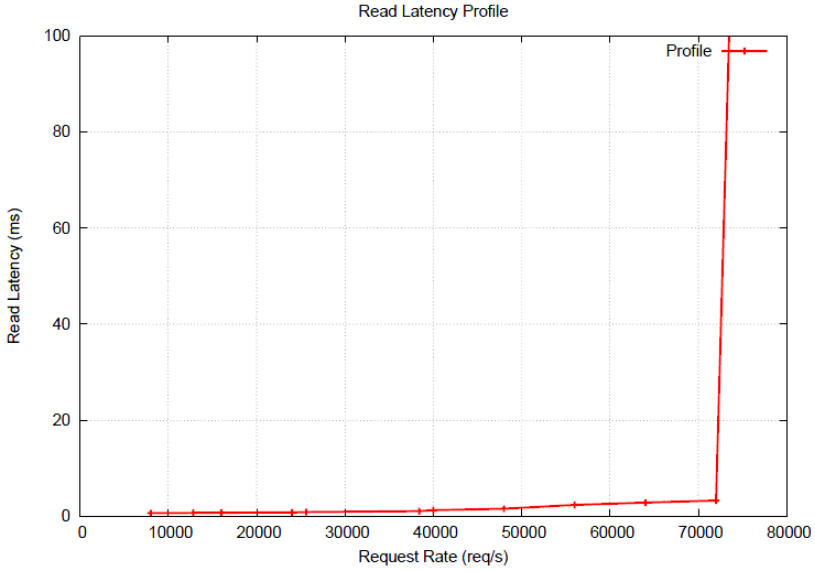

Figure 4.10: Read latency model for Cassandra

Chapter 5

Performance Model of Cassandra

A performance model represents a mathematical tool that is able to predict the future performance of a specific service (in our case Cassandra). Basically, we are interested in predicting how the performance of the service (read/write latency and maximum throughput) is affected by adding (or removing) one or multiple machines. Moreover, the performance model should also predict the behavior of the service if the request rate increases (or decreases).

Building a performance model for Cassandra represents a first step in understanding resource provisioning for a specific NoSQL data store. Using the profile built in the previous section, we present what are the implications of using NoSQL solutions for resource provisioning algorithms and how we can build a performance model for NoSQL data store.

We start by building a model for the traffic generated by a Cassandra cluster. This model is very important as it allows us to estimate what is the penalty, in terms of messages sent/received, of adding (or removing) resources to a Cassandra data store. Next, we build the maximum throughput model when the cluster size increases or decreases. We take into account the computing capacity improvement as well as the penalty of extra routing and gossiping. Last, but not least, we build the read/update latency model which is based on queuing theory.

All performance models that we propose in this chapter are highly influenced by the design of Cassandra, but the principles behind the design process of every performance models, can be used to build model for other NoSQL solutions. For each parameter that we aim to model, we start by building a theoretical model according to the way a specific NoSQL data stores (in our case Cassandra) works. Then, we use a set of values that we have obtained during the profiling phase in order to calibrate the model. Last, but not least, we evaluate the performance of the model by analyzing the error between the measured values and the predictions of our models, for other configurations then the calibration set.

5.1 Network Traffic Model

In this section we propose a prediction model for the traffic generated by Cassandra. Having a good approximation of the quantity of traffic that is generated for a certain request rate and cluster size is the first step in building the performance model for Cassandra. Based on the results proposed in this section, we can estimate the overhead implied by the inter-cluster communication (message generation time and network delays).

5.1.1 Theoretical Model

Provisioning resources to a Cassandra cluster increases not only the processing capacity, but also the traffic exchanged within the cluster. The additional traffic is generated by two distinct sources: the gossiping protocol and request routing. As we have seen in the previous chapter, a cluster containing a single node does not need either request routing, or gossiping, as all requests can be resolved by the node that is responsible for all keys. In this case, the messages sent and received by Cassandra represent the communication with the clients.

Taking all these into accountm we can express the traffic generated by a Cassandra cluster having n servers as:

$$T(n) = T_1 + T_{gossiping}(n) + T_{routing}(n), \tag{5.1}$$

where T_1 represents the traffic generated by a Cassandra cluster having a single node, $T_{gossiping}(n)$ represents the traffic generated by the gossiping protocol, while $T_{routing}(n)$ stands for the traffic implied by the requests that have to be routed to other nodes.

Assuming that we have a fixed request rate R, the traffic received by a Cassandra instance having a single service can be expressed by

$$T_1 = l \cdot R, \tag{5.2}$$

where l represents the average message length.

Gossiping Traffic Model

The gossiping protocol is used by Cassandra to exchange information regarding the state and key range corresponding to each node from the ring. As the cluster size increases, the traffic generated by the gossiping protocol also increases as more information has to be spread. Assuming that the quantity of data exchanged by the gossip protocol increases linearly with the cluster size, we can express the gossiping traffic as:

$$T_{gossiping}(n) = b \cdot n, \tag{5.3}$$

where b represents a constant that expresses the length of each message and the average number of gossiping messages that are exchanged within a certain period of time. More precisely, we could express

$$b = b(t), \tag{5.4}$$

but this would increase the complexity of our model as it is extremely hard to predict the life time of a certain cluster configuration. For simplicity, we analyze the traffic for a specific period of time and assume that b is constant during this period.

Routing Traffic Model

On the other hand, as the cluster size increases, the number of keys assigned to each node decreases. Thus, a larger percentage of the incoming requests have to be routed to a node that is responsible for the required key. Basically, for a cluster of size n we have for each node a fraction of the requests,

$$f_1(n) = \frac{1}{n}, \tag{5.5}$$

that are served locally while the other requests,

$$f_2(n) = 1 - \frac{1}{n}, \tag{5.6}$$

have to be routed to the nodes that are responsible for the required keys.

Assuming that the load is evenly distributed between nodes (on average, all nodes are responsible for the same number of keys and receive the same number of requests per second), we can consider that not only the requests arriving to a specific nodes are split according to f_1 and f_2, but all the requests to Cassandra follow the same distribution. Thus, we can express the routing traffic $T_{routing}(n)$ as:

$$T_{routing}(n) = f_2(n) \cdot R \cdot h = \left(1 - \frac{1}{n}\right) \cdot R \cdot h, \tag{5.7}$$

where R is the request rate while h is the overhead implied by message routing.

Taking all these into account we can express Formula 5.1 as

$$T(n) = l \cdot R + b \cdot n + \left(1 - \frac{1}{n}\right) \cdot R \cdot h. \tag{5.8}$$

For simplicity we consider $d = l \cdot R$ and $c = R \cdot h$. Thus, we can express the traffic as:

$$T(n) = d + b \cdot n + c\left(1 - \frac{1}{n}\right). \tag{5.9}$$

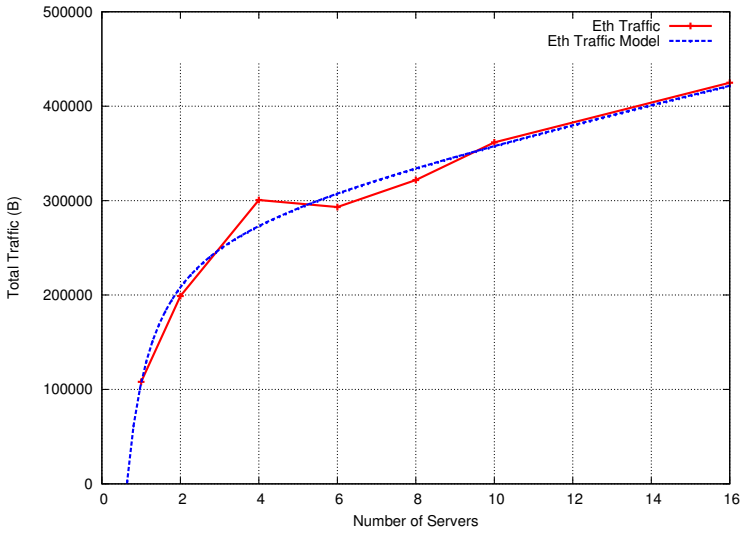

Figure 5.1: Traffic Model Calibration for Cassandra

Going one step forward,

$$T(n) = d + b \cdot n + c - c\frac{1}{n} = d + c + b \cdot n - c \cdot \frac{1}{n}. \tag{5.10}$$

Finally, considering that $a = c+d$ we obtain the model for the traffic generated by a Cassandra cluster of size n for a certain request rate:

$$T(n) = a + b \cdot n - c \cdot \frac{1}{n}. \tag{5.11}$$

5.1.2 Model Calibration

The traffic model that we have proposed varies over time and from one Cassandra deployment to another. This happens because parameters of the model a, b and c are not constant over all configurations. For instance parameter

$$a = d + c = R \cdot l + R \cdot h \tag{5.12}$$

is influenced by the request rate R, average message length l and the overhead implied by extra message routing. Typically, we can express $h = 2l$ as Cassandra uses one step routing and thus, a request that cannot be resolved locally is forwarded to a node that is responsible for the required key. The node that forwards the message waits for the reply and then forwards the response to the client. This means that instead of having one request and the corresponding response, we have two requests and two responses.

The average message length depends on the type of data stored in the database. Although Cassandra uses a key-value data model, the data section is not necessarily a single value (for instance an integer). More often, Cassandra stores, for each key, a collection of values, each one corresponding to a specific column. Thus, a read operation for a specific value can lead to a response containing multiple values, similar to an entire row from a traditional database.Taking all these into account, the value of the parameters should be computed according to the characteristics of each Cassandra deployment, during a calibration process.

In order to prove the correctness of our theoretical model, we measured the traffic generated by different Cassandra deployments having from one server to 18 servers. We have loaded 500000 records into the data store and we have used a workload containing a small number of requests (1000) in order to avoid high network load. Out of the nine configurations that we have used in the profile, we have used seven to calibrate the model using the method of the least squares. The other two configurations are used, in the next section, to evaluate the performance of the model.

Next, we used the parameters obtained during the calibration process to compare the model profile with the real data and the comparison is presented in Figure 5.1. As we can see in the figure, the shape of our model matches closely the profile of the measured traffic. Moreover, the model is very accurate for the considered configuration. The maximum error, approximately 9%, is encountered for a Cassandra cluster having 4 servers. For all other configurations, the error is less than 5%. It is very important to notice that, for the 4-server configuration, the traffic profile presents an anomaly as the traffic generated by 4 servers seems to be larger than in the case of 5 and 6 servers. The reason for this anomaly is probably extra DAS-4 management traffic during this experiment.

5.1.3 Experimental Results

Evaluating the performance of our model is done using two extra Cassandra cluster sizes that have been configured using the same parameters. The evaluation is done by analyzing the errors between the real values and the value predicted by our traffic model. The parameters of the model are the same as the ones obtained after the calibration process presented in the previous section. The results are presented in Figure 5.2. As we can observe in the figure, our model offers very accurate prediction for both configurations (13 and 18 servers). In both cases, the error is under 5%. These results prove the correctness of theoretical model that we have build for the traffic generated by Cassandra.

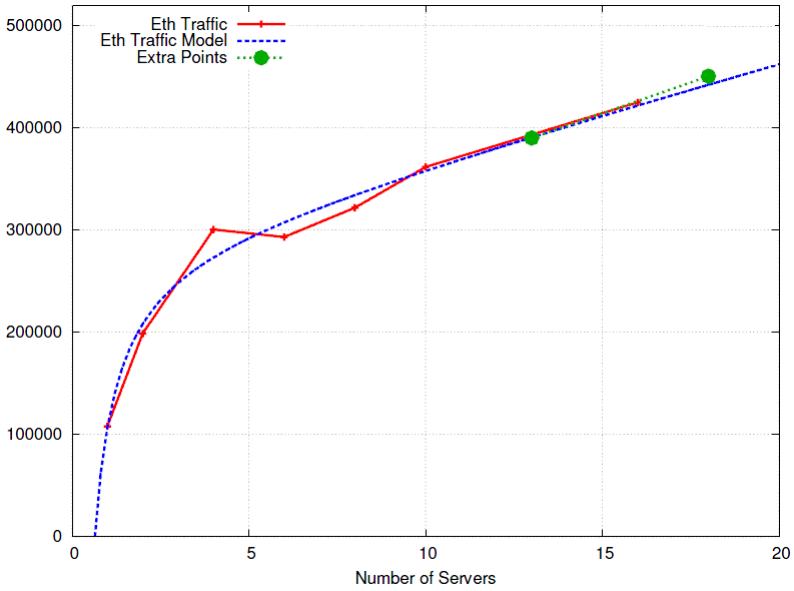

Figure 5.2: Traffic Model Evaluation for Cassandra

5.2 Maximum Throughput Performance Model

As we have seen in the previous chapter, Cassandra can support a maximum request rate for a certain configuration. Moreover, if we could maintain a constant request rate, larger than the maximum throughput of the cluster, Cassandra would crash. This is extremely important from the resource provisioning perspective as it shows that new resources must be added before the request rate reaches the maximum throughput of the Cassandra cluster. Thus, a performance model capable of predicting the maximum throughput of a specific Cassandra configuration is mandatory.

5.2.1 Theoretical Model

The first step in building the maximum throughput model is to understand what happens when the cluster size increases. First of all, by adding new resources, we increase the processing capacity of the Cassandra cluster. On the other hand, as the number of servers increases, the traffic generated within the cluster also increases as we have seen in the previous section. Taking all these into account, we can express the maximum throughput of a Cassandra cluster having n servers can be expresses as the difference between an ideal throughput ,$Th_{ideal}(n)$, considering that there is no intra-cluster communication, and the overhead implied by the communication, $O_{comm}(n)$:

$$Th_{max}(n) = Th_{ideal}(n) - O_{comm}(n) \tag{5.13}$$

Ideally, if we had a homogeneous set of servers that do not interact, each server would be capable of handling a maximum number of requests per second, independent of the cluster size. Thus, we can express the ideal throughput of a Cassandra cluster that contains n nodes as:

$$Th_{ideal}(n) = n \cdot Th_{max}(1) \tag{5.14}$$

Furthermore, the overhead implied by the inter-cluster communication can be expressed as the total traffic generated times the average overhead implied by the handling of each message. Thus, the communication overhead can be written as

$$O_{comm}(n) = t_{proc} \cdot T(n). \tag{5.15}$$

Replacing the $T(n)$ in the previous equation, according to Formula 5.11, we obtain:

$$O_{comm}(n) = t_{proc} \left(a + b \cdot n - c \cdot \frac{1}{n} \right) \tag{5.16}$$

Next, we replace $T_{ideal}(n)$ and $O_{comm}(n)$ in Formula 5.13 according to Formula 5.14 and Formula 5.16 and we can express the maximum throughput as:

$$Th_{max}(n) = n \cdot Th_{max}(1) - t_{proc} \left(a + b \cdot n - c \cdot \frac{1}{n} \right) \tag{5.17}$$

Going one step forward, the maximum throughput can also be expressed as:

$$Th_{max}(n) = n \cdot Th_{max}(1) - t_{proc} \cdot a - t_{proc} \cdot b \cdot n + t_{proc} \cdot c \cdot \frac{1}{n} \tag{5.18}$$

or

$$Th_{max}(n) = t_{proc} \cdot a + n \left(Th_{max}(1) - t_{proc} \cdot b \right) + t_{proc} \cdot c \cdot \frac{1}{n} \tag{5.19}$$

Last, but not least, if we replace $t_{proc} \cdot a$ with a_{th}, $Th_{max}(1) - t_{proc} \cdot b$ with b_{th} and $t_{proc} \cdot c$ with c_{th} we obtain the maximum throughput model for Cassandra:

$$Th_{max}(n) = a_{th} + b_{th} \cdot n + c_{th} \cdot \frac{1}{n} \tag{5.20}$$

5.2.2 Model Calibration

It is important to notice that the maximum throughput model uses the same type of expression as the network traffic model from the previous section (see Formula 5.11). However, the parameters of the model are different and highlight different aspects of a specific Cassandra configuration. Thus, we need a calibration process, similar to the one presented in the previous section, in order to adapt the model to each specific Cassandra deployment.

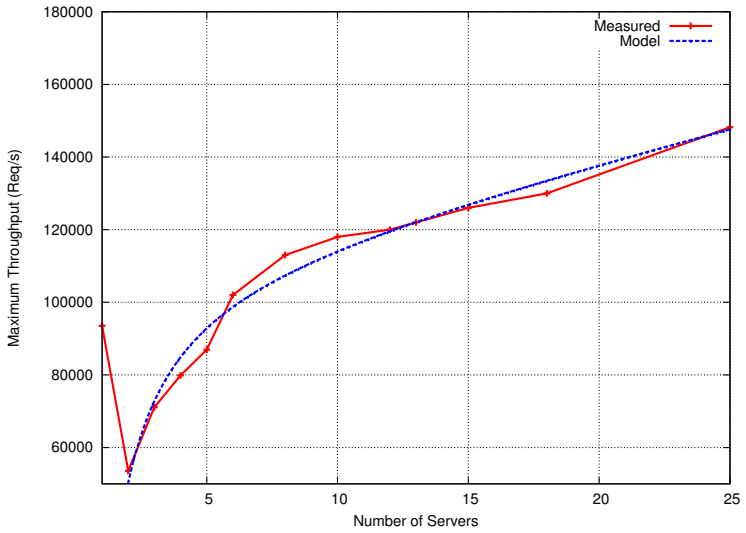

Figure 5.3: Maximum Throughput Model for Cassandra

To calibrate the model, we have used the profile presented in the previous chapter (see Figure 4.5). Thus, the parameters of the maximum throughput model have been computed using the method of the least squares in order to minimize the error of the model for the calibration points. The calibration set contains the maximum throughput measured for 12 different Cassandra cluster sizes, containing from 2 to 25 servers. During each test, 500,000 records were inserted in the data store, an then, multiple clients have stressed the cluster by submitting up to 1,000,000 requests.

Based on the parameters obtained through the calibration process we can compare the maximum throughput model to the profile obtained in the previous chapter. Both models are represented in Figure 5.3. As we can see in the figure, our model closely follows the maximum throughput profile. Moreover, the model produces very accurate approximations for all configurations. The maximum error, approximately 5%, is encountered for a Cassandra cluster having 5 servers. For all other configurations, the error is less than 5%.

5.2.3 Experimental Results

The evaluation of the proposed model takes into account three extra Cassandra configurations having 18, 28 and 30 servers. This values have been chosen to evaluate the accuracy of our model, not only for configurations having more servers then the maximum cluster size used within the calibration set, but also a configuration (18 servers) having less servers. The results are presented in Figure 5.4 where we show Cassandra's maximum throughput profile, model and

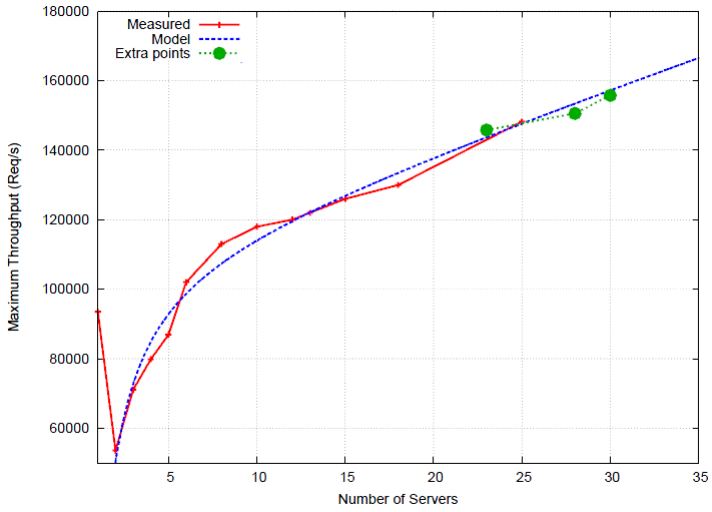

Figure 5.4: Maximum Throughput Model for Cassandra - Results

evaluation points (with green).

As we can see in Figure 5.4, for all evaluation points, the maximum throughput predicted by our model is very accurate, the maximum error being less than 3%. These results highlight not only the correctness of our model, but also its ability to offer very accurate predictions. This is extremely important as the maximum throughput of the same Cassandra configuration varies from one run to another. This happens because the each node that joins the cluster at initialization randomly chooses its position in the ring. Thus, the load among all servers is not always evenly distributed. This typically leads to variations of all parameter that we are interested in (e.g. maximum throughput, read latency or write latency).

It is important to mention that all the models that we have proposed assume that the load is evenly distributed between all Cassandra servers. Moreover, we assume that all resources are homogeneous. The first assumption is realistic because even if, at the first, thte load is not uniformly distributed, Cassandra uses compaction mechanisms in order to balance the load over time.

5.3 Read/Write Latency Performance Model

The read and the update models of Cassandra are very similar for the test scenarios that we have used. Thus, we focus only on the update latency profile considering that the same principles also apply for the read latency profile. What is extremely important about these models is the behavior of the system around the maximum throughput value. Unfortunately, the design of the YCSB clients

presented in the previous chapters limits the analysis of the request rates very close to the maximum throughput value.

Another important aspect is that, as we have already mentioned in the previous section, the maximum throughput point typically varies from one run to another. This leads us to the impossibility to determine the exact point where the system crashes. Using a lower bound for this value is the best choice we have, as we make sure that setting a lower maximum throughput than the actual value does not lead to system crash.

5.3.1 Theoretical Model

Modeling the update latency of Cassandra can be easily done if we consider that Cassandra acts like a M/M/n queue [15]. Thus, using the results of the queuing theory, we can express the latency of a request as

$$L_{QT}(\lambda, n) = \frac{1}{\mu - \frac{\lambda}{n}}, \tag{5.21}$$

where λ represents the request rate, μ stands for the maximum throughput, while n is the number of Cassandra servers.

In reality, Cassandra is a bit more complicated than a M/M/n queue. Basically, when a request arrives at a specific server, it first joins the waiting queue of the server until all requests that have arrived before it are resolved. When the request becomes active, based on the required key, the server can either resolve it or it can route the request to a server that is capable of producing the result. In this case, the request is forwarded to another server and has to wait in another queue. Taking all these into account, we can assume that all requests containing keys placed in the range of the destination server wait just one time, while all other requests have also to wait a second time. According to this principle, we can compute an average latency as follows:

$$L_{avg}(\lambda, n) = \left[\frac{1}{n} + 2\left(1 - \frac{1}{n}\right)\right] L_{QT}(\lambda, n) \tag{5.22}$$

It is important to understand that the previous expression computes the average latency at the server. On the other hand, the profile that we have built for Cassandra, measures the latency encountered at the client (YCSB client in our case). Thus, a more realistic model has also to take into account the network latency. Thus, we can express the update latency as

$$L(\lambda, n) = L_{avg}(\lambda, n) + N_{Lat}. \tag{5.23}$$

Last but not least, the latency model from the queuing theory is a bit rigid as it assumes that, for most of the values lower than the maximum throughput, the update latency is mostly the same. In reality, we have observed a linear growth

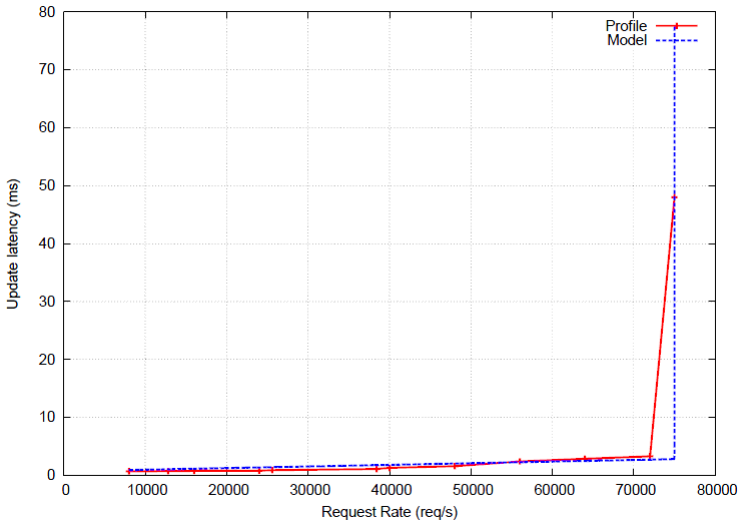

Figure 5.5: Update latency model for Cassandra

on the read latency. According to this, we have adjusted our model with a linear term as follows:

$$L(\lambda, n) = L_{avg}(\lambda, n) + N_{Lat} + a\lambda \tag{5.24}$$

Taking all these into account, we can express the update latency as

$$L(\lambda, n) = \left[\frac{1}{n} + 2\left(1 - \frac{1}{n}\right)\right]\left(\frac{1}{\mu - \frac{\lambda}{n}}\right) + N_{Lat} + a\lambda \tag{5.25}$$

5.3.2 Model Calibration and Evaluation

To calibrate the model, we have used the profile presented in the previous chapter (see Figure 4.10). The profiling has been performed on a Cassandra cluster containing 3 servers. During each test, 500,000 records were inserted in the data store, and then, multiple clients have stressed the cluster by submitting up to 1,000,000 requests. Moreover, he have increased the total target request rate from 8000 requests per second to up to 256,000 requests per second. As expected (considering the maximum throughput profile), the maximum throughput of the server is at most 74 000 requests per second.

Based on the value obtained (using he least squares method), we have been able to compare our performance model to the profile. The results are presented in Figure **??**. As we can see from the figure, the proposed model mimics the behavior of Cassandra very wel. The model offers a very good estimation of the update latency for values less than the maximum throughput. Moreover, our

model is also capable of predicting the system crash for request rates higher than the maximum throughput.

Unfortunately, as already explained at the beginning of this section, is it very hard to analyze the behavior of Cassandra for values very close to the maximum throughput value. This happens because the YCSB clients adjust the request rates to the maximum throughput of the data store. This mechanism is used to avoid system crash, but it is not realistic. Moreover, for different benchmark runs, the key distributions are different and thus, the maximum throughput point may vary according to the load distribution.

To illustrate the behavior of the Yahoo! Cloud Serving Benchmark clients, we chose the same Cassandra configuration, but we started 7 YCSB clients that started one by one with a 2 minutes delay in between. Each client has been configured to generate a target of 1,6 000 requests per second. The results are presented in Figure 5.6. In the upper chart, we present how the read latency experienced by the first clients varies over time, as more clients start. On the lower chart, we show how the request rate generated by the same client evolves as more clients start.

At first, the request rate is constant for up to four clients (at most 64,000 request per second). On the other hand, the average read latency increases a bit, but does not have significant variations. A very important moment is the initialization of the fifth clients, after 240 seconds. At this point the total target throughput of the five clients is more than Cassandra can handle. As a result, we observe that the latency becomes very unstable and on the other hand, the request rate generated by the client decreases. When the sixth client start, the instability of the system increases,while the request rate generated by the client decreases even more. The same behavior can be observed when the last client starts.

Based on the performance models that we have presented we will analyze in the next section how the behavior of Cassandra influences the resource provisioning process, what are the limitations of the current resource provisioning algorithms, and how can we adapt these algorithms to be able to deal with web applications that use NoSQL data stores.

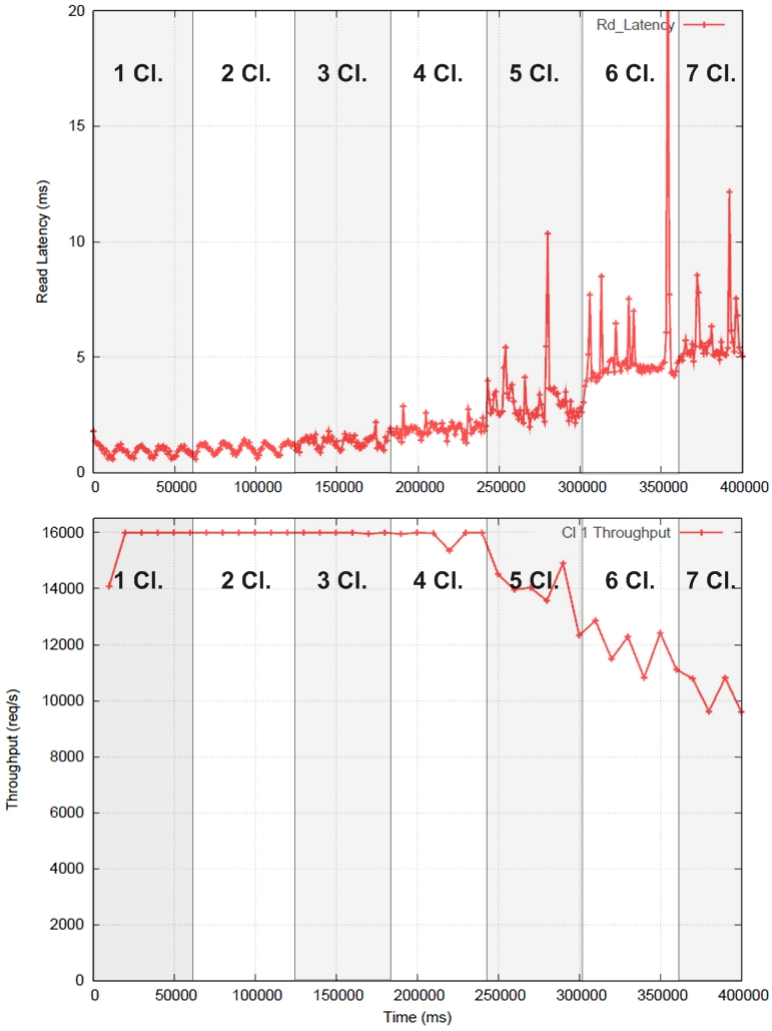

Figure 5.6: Update latency model for Cassandra

Chapter 6

Resource Provisioning for Cassandra

The performance models introduced in the previous sections provide not only accurate prediction tools, but they also highlight very important aspects that influence the resource provisioning process. In this chapter we analyze the impact of the proposed performance models from the resource provisioning perspective.

6.1 Existing Resource Provisioning Algorithms

NoSQL data stores are typically used by the most popular web applications like Google search engine, Facebook or Amazon. This is very important as such applications are very complex and involve tens of service invocations for each user action. This aspect narrows the range of the resource provisioning algorithms that are suitable for web applications based on NoSQL data stores.

Most of the existing resource provisioning algorithms assume that web applications have single-tier [10, 1] or multi-tier applications [18, 27, 28]. This assumption does not hold for the class of applications that we are interested in. Typically, large web applications are organized as a collection of dependent services that can be represented as a directed acyclic graph. Thus, we need a provisioning algorithm that is able to provide very good performance for arbitrary complex applications. We chose the one presented in [17].

This dynamic resource provisioning algorithm for service based web applications claims that a service level agreement (SLA) should only be assigned to the front-end service [17]. Based on this assumption, the reprovisioning process is triggered once the front-end SLA is violated. This approach provides very good results for web applications that do not rely on NoSQL data stores similar to Cassandra. However, in our case, a new approach is mandatory as we are going to see next.

6.1.1 Limitations

The most important problem of using a single SLA for the front end service is that services like Cassandra provide very good response times for request rates lower then a maximum value (the maximum throughput), but for higher request rates, the system crashes as we have seen in the previous chapter. This behavior can easily be highlighted by the read/update latency model proposed in the previous section, if we compute the limit response time for a value very close to the maximum throughput:

$$L_{lim} = \lim_{\lambda \to n \cdot \mu} L(\lambda, n). \tag{6.1}$$

According to Formula 5.25, we can express L_{lim} as:

$$L_{lim} = \lim_{\lambda \to n \cdot \mu} \left(\left[1\frac{1}{n} + 2\left(1 - \frac{1}{n}\right) \right] \left(\frac{1}{\mu - \frac{\lambda}{n}} \right) + N_{Lat} + a\lambda \right) \tag{6.2}$$

$$L_{lim} = \left[1\frac{1}{n} + 2\left(1 - \frac{1}{n}\right) \right] \lim_{\lambda \to n \cdot \mu} \left(\frac{1}{\mu - \frac{\lambda}{n}} \right) + N_{Lat} + a\lambda \tag{6.3}$$

But,

$$\lim_{\lambda \to n \cdot \mu} \left(\frac{1}{\mu - \frac{\lambda}{n}} \right) = \frac{1}{0_+} \to +\infty \tag{6.4}$$

Thus, we obtain

$$L_{lim} \to +\infty \tag{6.5}$$

This last result highlights that for request rates very close to the maximum throughput value, μ in our case, the system crashes.

6.2 A Resource Provisioning Algorithm for NoSQL Data Stores

Based on this result, a high performance resource provisioning algorithm for NoSQL data stores should also take into account the maximum throughput that can be obtained using a specific data store configuration. Basically, the algorithm proposed in [17] has to be extended with a second SLA for the data storing service. This second SLA should define a maximum request rate that can be handled by the storing service.

Deciding which is the optimal upper bound for the request rate specified in the Cassandra service SLA is extremely challenging and represents a very good direction to be explored in the future. This value is very important as it offers an estimation of the magnitude of a flash crowd that can be handled by the

application. Basically, an upper bound very close to the maximum throughput value can easily lead to a system crash in case of a sudden increase in request rate. Moreover, in deciding a proper value for the SLA should also take into account the impact of provisioning new resources over the performance of the system and the transition time to the system stabilization.

Assuming that we set the SLA to 95% of the maximum throughput of a Cassandra cluster containing 3 servers, the upper bound defined by the SLA would be 69350 requests per second, considering that the maximum throughput is 73000. If we have a 5% increase in the request rate, according to the read/update latency model, the latency would increase more than 5 times. For a request rate increase of more than 10% Cassandra would crash.

In the presence of a small scale flash crowd, a sudden increase of more than 4000 requests per second is very likely to happen. Thus, in this case, a lower value for the SLA would be a better choice. On the other hand, if we set the SLA to a much lower value we have efficiency problems due to the low utilization of the servers. Thus, deciding a proper value for the SLA is very challenging as it controls the trade-off between the server utilization and the magnitude of a flash crowd that could be gracefully handled.

The violation of the SLA corresponding to the storing solution does not have to lead to a global reprovisioning decision, but to a local reprovisioning for the NoSQL data store. On the other hand, for the violation of the front service SLA the provisioning algorithm should have its normal behavior according to [17]. Typically, this algorithm decides which is the best place to perform the reprovisioning based on the performance model of each service. More precisely, it computes, for each service, the effect of provisioning, and then, the algorithm chooses for provisioning the service that optimizes the overall performance of the system.

Computing the effect of adding (or removing) one or more resources to a Cassandra cluster can easily be done using the performance models introduced in the previous section. For instance, the effect of adding or removing one server over the maximum throughput of Cassandra can be expressed as:

$$\Delta Th = Th(n \pm 1) - Th(n). \tag{6.6}$$

Using maximum throughput model from the previous chapter (Formula 6.8) we obtain:

$$\Delta Th = a + b(n \pm 1) + c\frac{1}{n \pm 1} - a - bn - c\frac{1}{n}. \tag{6.7}$$

Going one step further, we can express the influence over the maximum throughput as:

$$\Delta Th = \pm b + c\left(\frac{1}{n \pm 1} - \frac{1}{n}\right), \tag{6.8}$$

or as:

$$\Delta Th = \pm b + c\frac{\mp 1}{n(n \pm 1)}. \tag{6.9}$$

Next, we can easily generalize the previous formula in order to compute the effect of (de-)provisioning k resources:

$$\Delta Th(\pm k) = \pm kb + c\frac{\mp k}{n(n \pm k)} \tag{6.10}$$

Using a similar approach we can also compute the effect of (de-)provisioning for the read or write latency of Cassandra.

Chapter 7

Conclusions

During the last few years, web applications have become very popular and today, the most popular applications have tens of millions users. For instance, Facebook, the most popular social network, has more than 500 million active users [25]. Moreover, Google+ [14], probably the future most important competitor of Facebook, has exceeded the 10 million users barrier in just 16 days after the release, 50 times faster than Facebook [12]. The immense impact of Google+ is even more impressive if we take into account that the application has not been officially released. Currently, new users must be invited by existing users to join the network.

To understand the impact of poor resource provisioning algorithms, we can analyze what happened to Amazon.com in May 2011. The website had a one night special offer for the latest album of Lady Gaga where users could download the album for just one dollar. Unfortunately Amazon did not expected Lady Gaga to be so popular. As a result, the very high number of download requests crashed the Amazon's servers [29].

Current research on resource provisioning assumes either that web applications have simple architectures (single-tier or multi-tier), or that they use traditional databases to store and manage date. In the context of popular web applications, neither of these assumptions is true. Typically, this type of web applications have very complex architectures that are organized as dependent services. Moreover, modern web applications rely on NoSQL data stores as they have to manage large volumes of data that go far beyond the limits of traditional databases.

The goal of this research was to provide a new perspective for resource provisioning by analyzing how NoSQL data stores influence the provisioning process. To achieve this goal, we analyzed the particularities of different NoSQL implementation and how different design decisions influence the resource provisioning process.

As a proof of concept we have chosen Cassandra [19], one of the most popular open source NoSQL solutions. Our first objective has been to understand how

Cassandra works and how different design decisions influence the performance of the system. To achieve this objective, we used the Yahoo! Cloud Serving Benchmark to build a performance profile of Cassandra, necessary for a very good understanding of the data store's behavior.

The performance profile of Cassandra provides valuable information from the resource provisioning perspective. The most important aspect highlighted by the performance profile is the fact that each Cassandra configuration has a maximum request rate that it can sustain. For higher request rates, the system crashes.

Next, based on the design of Cassandra, we built performance models for different parameters of Cassandra including the generated traffic, the maximum throughput and the read/write latency. Moreover, we used the information provided by the performance profiles in order to prove the corectness of the proposed performance models. The performance models proposed by this research provide very accurate predictions, having an error less then 5% for all the tests we have performed.

Moreover, based on the proposed models, we prooved that dynamic resource provisioning algorithms should assign a SLA to the data storing service. This SLA imposes an upper bound for the request rate that represents a percentage of the maximum throughput that can be achieve with the current configuration. The violation of this SLA should lead to the provisioning to the NoSQL data store service.

The most important part of this research is the understanding of how different design decisions used by NoSQL data stores influence the behavior of the system and the performance modeling. Even if all the models presented in this research ars built specifically for Cassandra, the steps that we have presented and followed in order to develop these models are the same for all NoSQL data stores. Of course, the performance model for Cassandra, will not work for other solutions like Bigtable[5], but they represent a example of how the design decisions used by Cassandra designers influence the performance models.

7.1 Future Work

Resource provisioning for NoSQL data stores represent a very important topic and our research is the first one to explore this domain. The principle we have stated and the models that we have introduced in the previous sections represent just the first steps towards high performance resource provisioning for complex web applications relying on NoSQL data stores. This domain is extremely vast and challenging due to the large variety of NoSQL solutions and the scale of the applications depending on them.

Unfortunately, the performance profile that we have built has several limitations. First of all, the benchmarking tool that we have used is not capable of maintaining a fixed request rate. As we have already mentioned, if the target

request rate is higher than the maximum throughput of the Cassandra cluster, the benchmarking tool reduces the request rate to avoid the system crash. From the performance profiling perspective, this behavior is not benefic, as we are not able to find the exact maximum throughput point. Thus, we made multiple experiments and we have chosen the maximum throughput value as the lowest encountered maximum throughput value. We have chosen this approach because, from the perspective of a large scale web application, is better not to utilize all the servers at maximum capacity, then to overload the servers and crash the system. Taking all these into account, a more accurate benchmarking tool is necessary. Such solution should be capable of maintaining a fixed request rate independent of the data store capacity.

The workloads that we used throughout the profiling of Cassandra and evaluation of the proposed models have been limited by the resources and shared nature of the machines used to perform the experiments. Thus, more realistic experiments could improve the quality of the proposed models.

A very challenging problem that could be addressed by future research is the problem of deciding how to adapt SLA corresponding to the NoSQL service to the size of the cluster and request rate history.

One of the most challenging research directions is represented by the analysis of the impact of dynamic resource provisioning for the performance of Cassandra. Even more challenging is to build models capable of deciding which is the best moment to provision new resources in order to reduce the impact of provisioning over the performance of the system.

To sum up, this research addresses the problem of resource provisioning in the context of NoSQL and presents the steps to be followed in order to build an efficient performance model for NoSQL data stores. As a proof of concept, we built the performance model for Cassandra and we have proved its correctness. Moreover, we have shown the necessary steps that should be followed in order to adapt existing resource provisioning algorithms for large scale web applications based on NoSQL data stores similar to Cassandra. Finally, we suggested future research directions toward efficient resource provisioning in the presence of NoSQL data stores.

Bibliography

[1] Tarek Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13:2002, 2001.

[2] Cassandra @ Twitter an Interview with Ryan King. http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king. Accessed on June 30h, 2011.

[3] Eric A. Brewer. Towards robust distributed systems. (invited talk). *Principles of Distributed Computing, Portland, Oregon*, July 2000.

[4] Apache Cassandra. http://incubator.apache.org/cassandra/. Accessed on June 14th, 2011.

[5] 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 Proc. OSDI, pages 15–15, 2006.

[6] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In Proc. 34th VLDB, 2008.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. SoCC, pages143–154, 2010.

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

[9] Distributed ASCI Supercomputer 4. http://www.cs.vu.nl/das4/home.shtml. Accessed on July 13th, 2011.

[10] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-based resource provisioning in a web service utility. In *Proc USITS*, page 5, 2003.

[11] Jeremy Elson and Jon Howell. Handling flash crowds from your garage. In *USENIX 2008 Annual Technical Conference*, pages 171–184, Berkeley, CA, USA, 2008.

[12] Chart: Google+ Hit 10 Million Users 50 Times Faster Than Facebook. http://www.theatlanticwire.com/technology/2011/07/chart-google-reached-10million-users-50-times-faster-facebook-or-twitter/40295/. Accessed on July 25th, 2011.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proc. SOSP*, pages 29–43, 2003.

[14] Google+. https://plus.google.com/. Accessed on July 25th, 2011.

[15] N. J. Gunther. Analyzing Computer System Performance with Perl::PDQ. Springer, 2005.

[16] Apache HBase. http://hadoop.apache.org/hbase/. Accessed on June 14th, 2011.

[17] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *The 18th International World Wide Web Conference*, pages 471–480, 2010.

[18] Abhinav Kamra. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *International Workshop on Quality of Service*, pages 47–56, 2004.

[19] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *The 28th ACM symposium on Principles of distributed computing*, 2009.

[20] Charlene O'Hanlon. A Conversation with Werner Vogels. *ACM Queue*, 4:14–22, May 2006.

[21] Jim Gray Pat, Dennis Shasha (nyu, Jim Gray, Jim Gray, Pat Helland, Pat Helland, and Dennis Shasha. The dangers of replication and a solution. In *Proc. SIGMOD-*, pages 173–182, 1996.

[22] Forrester Research. An updated look at customer reaction to a poor online shopping experience. 2009.

[23] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced paxos commit for transactions on dhts. In *Proceedings of the 10th*

IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pages 448–454, Washington, DC, USA, 2010.

[24] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of The 7th ACM SIGPLAN workshop on ERLANG*, New York, NY, USA, 2008.

[25] Facebook Official Statistics. www.facebook.com/press/info.php?statistics.

[26] Saying Yes to NoSQL; Going Steady with Cassandra. http://about.digg.com/node/564. In *Accessed on July 3rd, 2011.*

[27] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 291–302, New York, NY, USA, 2005. ACM.

[28] Daniel Villela, Prashant Pradhan, and Dan Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Trans. Internet Technol.*, 7, February 2007.

[29] Amazon Offers 99-Cent Lady Gaga Album Again (with New No-Crash Promise). http://techland.time.com/2011/05/26/amazon-offers-99-cent-lady-gaga-album-again-with-new-no-crash-promise/. Accessed on July 25th, 2011.