Managing service performance in NoSQL distributed storage systems

Maria Chalkiadaki

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, Crete, GR-70013, Greece

Thesis Advisor: Prof. *Christos Nikolaou*

This work has been performed at the **University of Crete, School of Science and Engineering, Computer Science Department** and at the **Institute of Computer Science (ICS) - Foundation for Research and Technology - Hellas (FORTH), Heraklion, Crete, GREECE.**

The work is partially supported by the **PaaSage (FP7-317715) EU project**.

University of Crete
Computer Science Department

Managing service performance in NoSQL distributed storage systems

Thesis submitted by
Maria Chalkiadaki
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Maria Chalkiadaki

Committee approvals: _____
Christos Nikolaou
Professor, University of Crete, Thesis Supervisor

_____
Kostas Magoutis
Researcher, Foundation for Research and Technology - Hellas

_____
Dimitris Plexousakis
Professor, University of Crete

Departmental approval: _____
Angelos Bilas
Professor, Director of Graduate Studies, University of Crete

Heraklion, October 2013

Abstract

A new class of distributed data stores, often referred to as NoSQL key-value stores, have recently been developed to support large-scale data-centric services. Cloud service offerings of these technologies combine their performance and availability benefits with the flexible economic model of Cloud computing. Enterprises that have invested into Cloud technologies are now raising their expectations from best-effort to guaranteed levels of service. Responding to this need, the research community is now focusing on data-centric services that support controlled performance, predictable reliability, and guaranteed I/O throughput.

In this thesis we describe the architecture of a quality-of-service (QoS) infrastructure for achieving controlled application performance over the Apache Cassandra distributed key-value store. We present an implementation of our architecture and provide results from an evaluation using an extended version of Yahoo Cloud Serving Benchmark (YCSB) on the Amazon EC2 Cloud. A key focus of this thesis is on a QoS-aware measurement-driven provisioning methodology. Our evaluation provides evidence that the methodology is effective in estimating application resource requirements and thus in achieving the type of controlled performance required by data intensive performance-critical applications. While our architecture is implemented and evaluated in the context of the Cassandra distributed storage system, its principles are general and can be applied to a variety of NoSQL systems.

Περίληψη

Τα συστήματα υπηρεσιών μεγάλης κλίμακας απαιτούν υψηλή απόδοση, διαθεσιμότητα, και αξιοπιστία για να ικανοποιήσουν τις ανάγκες της παγκόσμιας ηλεκτρονικής αγοράς. Η πρόσφατη τάση σχεδίασης νέων υπηρεσιών βασισμένων σε υπολογιστικές υποδομές νέφους (Cloud computing) αποσκοπεί στο συνδυασμό ενός ευέλικτου οικονομικού μοντέλου με την υψηλή απόδοση και διαθεσιμότητα των τεχνολογιών κατανεμημένου υπολογισμού και αποθήκευσης που εστιάζουν σε περιβάλλοντα νέφους. Μια τέτοια σημαντική τεχνολογία είναι και οι μη-σχεσιακές (NoSQL) κατανεμημένες βάσεις δεδομένων, οι οποίες σήμερα χρησιμοποιούνται από παρόχους υπηρεσιών όπως οι Google, Yahoo, Facebook, κά. Καθώς οι ανάγκες των παρόχων επεκτείνονται στην προσφορά εγγυήσεων απόδοσης της υπηρεσίας, η ερευνητική κοινότητα επικεντρώνεται σε μηχανισμούς που παρέχουν ελεγχόμενη απόδοση, προβλέψιμη αξιοπιστία, και εγγυημένη απόκριση ή/και ρυθμοαπόδοση εισόδου / εξόδου.

Στην εργασία αυτή, παρουσιάζουμε ένα σύστημα παροχής ποιότητας υπηρεσιών εστιασμένο στον έλεγχο της απόδοσης εφαρμογών πάνω από το κατανεμημένο NoSQL σύστημα αποθήκευσης Apache Cassandra. Παρουσιάζουμε την αρχιτεκτονική και υλοποίηση του συστήματος και αναλύουμε τα αποτελέσματα αποτίμησης του χρησιμοποιώντας μια επέκταση του Cloud Yahoo Serving Benchmark (YCSB) σε εικονικές μηχανές του Amazon EC2. Το σύστημά μας βασίζεται σε μια μεθοδολογία πρόβλεψης της ποιότητας των υπηρεσιών με βάση συστηματικές μετρήσεις. Η πειραματική αποτίμηση αποδεικνύει ότι η μέθοδος είναι αποτελεσματική για την εκτίμηση των απαιτούμενων πόρων για μια εφαρμογή και κατά συνέπεια, για τον έλεγχο της απόδοσης των εφαρμογών. Η υλοποίηση που παρουσιάζουμε είναι επικεντρωμένη στο Apache Cassandra, ωστόσο οι βασικές της αρχές είναι γενικές και μπορούν να εφαρμοστούν και σε άλλα NoSQL συστήματα.

Acknowledgements

Firstly, I need to express my gratitude to the University of Crete and the Department of Computer Science for providing me with proper education, as well as the Institute of Computer Science of the Foundation for Research and Technology (ICS-FORTH) for supporting me financially through the graduate scholarship. Also, I offer my sincerest appreciation to my supervisor Prof. Christos Nikolaou for supervising and having trust in this work.

In addition, I would like to deeply thank my co-supervisor Dr. Kostas Magoutis for the valuable guidance, the useful advice, and his willingness to help me at every stage of this thesis. During my graduate studies he introduced me to the notion of research, fortifying the foundations for my further career in computer science.

I would also like to thank my friends: Katerina Tzompanaki, Ira Chrysochoou, Sofia Karfi, Maria Sartzetaki, Angeliki Nouvaki, Maria Koutentaki, Ioannis Sfakianakis, Nikos Stamatakis, Charis Souris, Panos Tselios, Christos Asaridis, Giorgos Papadimitriou, and Konstantina Konsolaki. My warmest appreciation to Markos Fountoulakis; our discussions and his useful advice were crucial for the completion of this thesis. Special thanks to Vassilis Papakonstantinou for encouraging me and endured my disappointments with patient and fortitude.

Last but not least, I am grateful to my parents Giorgos and Anna that always encourage and support me to fulfill my dreams.

Maria Chalkiadaki

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The new breed of *NoSQL* distributed storage systems has dramatically changed the landscape of how information is represented, manipulated, and stored in large-scale infrastructures today. These systems are currently at the forefront of academic research and industrial practice, primarily due to their high scalability and availability features. The demand for inexpensive scalability in data-intensive analytics (web search, data warehouse analysis, etc.) has led to the adoption of *NoSQL* systems (contrasted to *SQL* systems or traditional relational databases). Such systems implement simple interfaces to non-relational data representations and are well integrated with data programming platforms such as MapReduce [1]. A number of such platforms are currently implemented in Cloud infrastructures and offered to applications developers as utility services. Managing service performance over these systems is an area that attracts significant interest as manifested by the success of early service offerings in this space (e.g., Amazon DynamoDB [2]).

In this thesis we present a quality of service (QoS) architecture and prototype that offers managed service performance over a prominent NoSQL system, Apache Cassandra. Our architecture is able to address the *storage configuration* problem, namely how to appropriately provision initial storage resources for a target workload given a simple description of its characteristics. The architecture continuously monitors performance and controls resource allocations in order to achieve stated goals by utilizing *storage elasticity* mechanisms. It is also able to address the *dynamic adaptation* problem by monitoring service performance at runtime and adjusting to short-term variations by either throttling the application or by expanding the set of resources assigned to it. Our focus in this thesis is primarily on the first problem: our solution to it is based on a methodology using targeted measurements to produce a set of tables expressing benchmark performance over different configurations of Cassandra (number, type of servers). Using the pro-

duced tables, we can estimate the resources required to achieve the service-level objective (SLO) of an application by interpolation from the baseline measurements.

Our key contributions in the thesis are the following:

- A novel methodology for configuring Cloud-based Cassandra storage clusters for specific application SLOs.
- A dynamic Quality of Service monitoring and adaptation mechanism to control short-term workload variations.
- An extensive evaluation of our methodology in Amazon Web Services' EC2 Cloud.

Our exposition proceeds as follows: In Chapter 2, we provide the background for this thesis. In Chapters 3, and 4, we describe our architecture and implementation in the context of Apache Cassandra. In Chapter 5, we present our evaluation, and in the Chapter 6, we discuss related works in this space. Finally, in Chapter 7, we conclude.

Chapter 2

Background

2.1 SQL versus NoSQL

Relational databases were invented in the 1970s to offer applications the ability to store structured data and to retrieve it using a high-level query language(Structured Query Language, or SQL) allowing construction of powerful queries and operations on that data. Relational databases guarantee atomicity, consistency, isolation, and durability (collectively referred to as ACID properties) in data transactions.

The new breed of NoSQL databases trade off several aspects of the relational database model for increased scalability and speed. NoSQL systems are also referred to as "Not only SQL" to emphasize that in many cases offer SQL-like query languages. They offer dynamic schema less data storage, making it unnecessary to set a predefined schema in advance. In addition, they offer the ability to scale horizontally by adding servers instead of concentrating more capacity on a single server as usually is the case in SQL databases. The motivation for NoSQL systems therefore comes from their design simplicity, control over availability, and higher performance by relaxing data consistency. By offering weaker data consistency models such as eventual consistency, they trade off consistency for efficiency. This means that given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually through the system. Many NoSQL systems also feature versatile load balancing and high availability mechanisms: data and query load are balanced across servers; when a server goes down, it can be quickly and transparently replaced with no application disruption through replication. This attribute offers high availability and easier recovery without involving separate applications to manage these tasks. The storage environment is virtualized from the developer's perspective. Lastly, many NoSQL database technologies feature advanced caching capabilities, keeping frequently-used data in

system memory as much as possible.

The following table [3] summarizes the main differences between SQL and NoSQL databases.

Feature	SQL Databases	NoSQL Databases
Types	One type (SQL database) with minor variations.	Many different types including key-value stores, document databases, wide-column stores, and graph databases.
Development History	Developed in 1970s to deal with first wave of data storage applications.	Developed in 2000s to deal with limitations of SQL databases, particularly concerning scale, replication and unstructured data storage.
Examples	MySQL, Postgres, Oracle Database	MongoDB, Cassandra, HBase, Neo4j, etc.
Schemas	Structure and data types are fixed in advance. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.	Typically dynamic. Records can add new information on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary. For some databases (e.g., wide-column stores), it is somewhat more challenging to add new fields dynamically.

Data Storage Model	Individual records (e.g., "employees") are stored as rows in tables, with each column storing a specific piece of data about that record (e.g., "manager," "date hired," etc.), much like a spreadsheet. Separate data types are stored in separate tables, and then joined together when more complex queries are executed. For example, "offices" might be stored in one table, and "employees" in another. When a user wants to find the work address of an employee, the database engine joins the "employee" and "office" tables together to get all the information necessary.	Varies based on NoSQL database type. For example, key-value stores function similarly to SQL databases, but have only two columns ("key" and "value"), with more complex information sometimes stored within the "value" columns. Document databases do away with the table-and-row model altogether, storing all relevant data together in single "document" in JSON, XML, or another format, which can nest values hierarchically.
Scaling	Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required.	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The NoSQL database automatically spreads data across servers as necessary.
Development Model	Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database).	Open-source.
Support Transactions	Yes, updates can be configured to complete entirely or not at all.	In certain circumstances and at certain levels (e.g., document level vs. database level).
Data Manipulation	Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE...	Through object-oriented APIs.
Consistency	Can be configured for strong consistency.	Depends on product. Some provide strong consistency (e.g., MongoDB) whereas others offer eventual consistency (e.g., Cassandra).

Table 2.2: SQL vs. NoSQL Summary [3]

The last few years, the NoSQL storage systems became extremely popular. So, there are many different implementations and types for NoSQL systems. In general, we could assume that there are the following categories:

- **Wide-Column store or Column Families store**: optimized for queries over large datasets, and store columns of data together, instead of rows. Examples of wide-column store systems: Cassandra [4], HBase [5], Hypertable, Accumulo, Amazon SimpleDB, Cloudata, Cloudera, HPCC, etc.

- **Document store**: each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents. Examples of document store systems: MongoDB, RethinkDB, CouchDB, ElasticSearch, Couchbase Server, RavenDB, MarcLogic Server, etc.

- **Key-Value store**: single item in the database is stored as an attribute name, or key, together with its value. Examples of key-value store systems: DynamoDB [6], BigTable [7], Riak, Voltemort, Azure Table Storage, Redis, FoundationDB, BerkleyDB, LevelDB, etc.

- **Graph store**: store information about networks, such as social connections. Examples of graph stores systems: Neo4J, HyperGraphDB, Infinite graph, InfoGrid, DEX, GraphBase, etc.

  In this thesis, we focus on the open-source NoSQL distribute storage system, Apache Cassandra (more details about the architecture of Apache Cassandra in section 2.4).

2.2 Quality of service

In computer science, functional requirements define a function of a system or its component. They deal with what the system should do or what the system provides to the users and they include implementation and deployment issues. Functional requirements are supported by non-functional requirements which specify characteristics for detail constraints, targets or control mechanisms for the system. Quality of Service refers to the important criteria for non-functional requirements. The goal of the QoS is to provide guarantees on the ability of a system to deliver predictable results for their clients. The characteristics are some necessary requirements in order to understand the behavior of the system or service. It is worth noting that while QoS in networking is a relatively mature field whose numerous

research results have progressed in many cases into formal protocol specifications and products, storage QoS is a less mature area due to the significantly more challenging technical issues involved (such as for example non-linear behavior due to caches and the strong dependance on workload characteristics).
In general, Quality of Service [8] is a combination of several qualities or properties of a system, such as:

- **Availability**: represents the percentage that the system is available, operates without downtimes.
- **Reliability**: is when a system is consistent, operates correctly and maintains its performance.
- **Response time**: is the time a system responds to different types of requests. More specific, response time is a function of load intensity and it usually is measured as rate of requests per second or number of concurrent requests.
- **Throughput**: represents the rate that a system processes its incoming requests.
- **Security**: includes authentication and authorization mechanisms, message integrity, and confidentiality of the system.

2.3 Service Level Agreements

An SLA (service-level agreement) is part of the contract between two parties, usually the service provider and customer in order to define the quality of service. SLAs have been used in IT organizations, service-oriented architectures environments and generally in businesses for many years. The main role of the SLA is to set the expectations, requirements, obligations, quality attributes, and important aspects of the system or service between the two parties. They usually are not guarantee that all promises and constraints are kept but they set the penalties when the promises are not satisfied. The SLA specification can be either plain-text document or machine-readable format, specifically using XML language and it might be signed by all parties. There are many standards such as WS-Agreement by the Open Grid Forum and WSLA by IBM [9] specifying SLA requirements and obligations in machine-readable form. The main difference between this specification and WSLA, is WS-Aggreement extends Web Services standards functionality allowing the establishment of an agreement. On the other hand, WSLA offers a complete language for the defining, monitoring, and evaluating Service Level Agreements.

2.3.1 WSLA framework

The WSLA framework [10] uses XML language and defined as XML schema. The goal of WSLA is to allow the creation of machine-readable SLAs for services implemented using Web services technologies.
The SLA that has created using WSLA framework usually contains the following sections [9]:

- A description of all parties.
- A description of the WSDL interfaces of the web services that the parties implement.
- A definition of the service parameters, their operations, and their metrics.
- Obligations of the agreement.
- The action guarantees that all parties commit to perform.

2.3.2 Web-Service Agreement specification

Web Service-Agreement specification [11](WS-Agreement) is a protocol to establish an agreement between two parties. This specification developed with Grid Resource Allocation Agreement Protocol working group (GRAAP-WG) and Web Service Level Agreement specification(WSLA) [12] by the OGF. As WSLA, it provides an XML-language and XML Schema for defining the structure of the SLA document.

The SLA document created by WS-Agreement contains the following sections [9]

- A mandatory unique ID for the agreement and optionally a name.
- The context of the agreement.
- The metadata about the agreement, such as template agreement ID, expiration time, etc.
- The service terms that describes the service.
- The guarantee terms that the parties commit to perform.

WS-Agreement produces a contract between customer and provider that specifies the rights and obligations in order to achieve availability of resources and/or on service qualities. Each agreement is created based on agreement templates.

An agreement template is developed using extensible XML language and it describes what the agreement responder is willing to accept. It is similar document to agreement but it additional contains the ability to create constraints. Constraints describe the structure, the valid ranges, and the values of the terms for an acceptable agreement for the agreement responder. The agreement also uses XML language. When two parties finally create an agreement, the WS-Agreement defines also the type of monitoring needed for the validation of the agreement. The WS-Agreement model defines two different types of services: the agreement factory and the agreement service. The agreement factory is used to create agreements between a service consumer and provider and for instantiating the associated service with the agreed Quality of Service. The monitoring of the agreement and agreement service, through WS-Agreement is out of scope of this thesis. We focus in agreement factory service that are modeled as web service resource using Web Services Resource Framework (WSRF) specification. Generally, a web service resource is a web service instance that is uniquely identified by an endpoint reference (EPR). Endpoint references are defined in the WS-Addressing specification.

There are two parties in WS-Agreement [13] [11], the service customer or initiator and the service provider or service responder. The service initiator requests the available templates from the service responder using the WSRF GetResourceProperty method. The WSRF get resource property is a method that the agreement factory service publishes the agreement templates. The service initiator chooses a template and then it creates a new agreement offer from it. An offer describes the service to provide the guarantees, obligations and requirements for each party. After that, the service initiator sends back the offer to the service provider to create the service agreement. The service initiator is bound to the offer. If the service provider accepts the offer, then it returns a reference to a new agreement instance, otherwise it returns an error. In the WS-Agreement specification, there are two ways to create an agreement in the agreement factory. The synchronous method (createAgreement) specified by the agreement factory port type and the service provider has the obligation to accept or reject immediately the offer. On the other hand, the asynchronous method (createPendingAgreement) that is specified by the PendingAgreementFactory port type and the service provider could respond to service customer at a later time. Figure 2.1 shows the message exchange between service initiator and service responder using synchronous create of agreement.

All these standards can be straightforwardly leveraged for expressing SLAs over NoSQL technologies such as Cassandra.

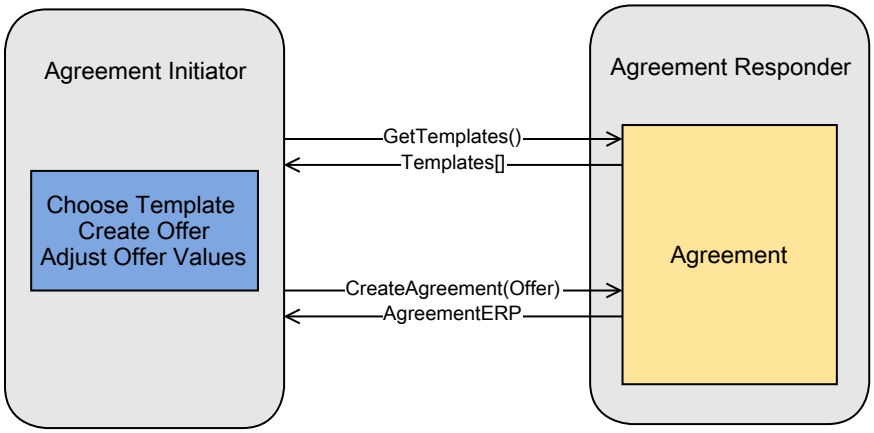


Figure 2.1: The process to create an agreement using synchronous WS-Agreement [13].

2.4 The architecture of Apache Cassandra

Apache Cassandra [4] [14] is an open-source distributed storage system that designed to handle large amounts of data across many servers, providing high availability with no single point of failure. Apache Cassandra combines features from Amazon DynamoDB [2] (in infrastructure), and Google BigTable [7] (especially in the data model).

Apache Cassandra [15] was developed at Facebook for the Inbox Search from Avinash Lakshman (one of the authors of Amazon's Dynamo) and Prashant Malik. In July 2008, it was released as an open source project on Google code. In March 2009, it became an Apache Incubator project and on February 17, 2010 it graduated to a top-level project. Now, it is an Apache Software Foundation project, under Apache License.

Apache Cassandra [14] has a peer-to-peer distributed architecture that is easy to set up and maintain. Apache Cassandra offers no single-point-of-failure and continuously availability. There is no master-server and slave-server node. Any number of commodity servers can be grouped into a Cassandra cluster in any datacenter, without having to worry about the type of machines. Every server accepts requests from any client. Every server is equal and all nodes communicating with each other via a gossip protocol. Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about. Apache Cassandra's built-for-scale architec-

ture means that it is capable of handling petabytes of information and thousands of concurrent users/operations per second (across multiple data centers). When a node first starts up, it looks at its configuration file to determine the name of the Cassandra cluster it belongs to and which nodes, called seeds, to contact to obtain information about the other nodes in the cluster. Another important aspect of the gossip process is the heartbeats from the other nodes in the cluster directly or indirectly. On the other hand, when a node goes down and then comes back, it may have missed writes for the replica data it maintains. The missed writes are stored by other replicas for a period of time providing hinted handoff is enabled. Moreover, in order to know which range of data it is responsible for, a node must also know its own token and those of the other nodes in the cluster. All the above parameters are customized in Cassandra's configuration file by the programmer or administrator of the Cassandra cluster.

The Apache Cassandra offers column-oriented data model. Unlikely to relational databases, it is not necessary to model all of the columns of a row. Each row is not required to have the same set of columns. The *keyspace* in Cassandra is the container of the data and it is similar to schema in a relational database. Each keyspace has one or more *column family* objects, that are similar to tables in relation databases. Column families contain columns that are identified by a row key. It is important to mention that each row in a column family is not required to have the same set of columns. The columns of a column family are accessed together. It is not supported joining column families at query time and there are no foreign keys in Cassandra.

2.4.1 Distribution and replication data

The data of a cluster in Apache Cassandra [14] are distributed across all nodes of the *ring*. The data partitioning across all nodes is either random or by lexicographical ordered (the default is Random Partitioning). Apache Cassandra also provides a built-in replication mechanism, to store copies of data across different nodes that participate in the same ring. The copies stored, called replicas, based on the row key. The total number of replicas across the cluster is referred to as the replication factor in the configuration file of Cassandra server. This means that if any node in a cluster goes down, one or more copies of that node's data is available on other machines in the cluster. All replicas are equally important, that means, there is no primary or master replica. Replication options are provided that also allow for data to be automatically stored in different physical racks (thus ensuring extra safety in case of a full rack hardware failure), multiple data centers, and cloud platforms. Replication ensure reliability and fault tolerance. There are two

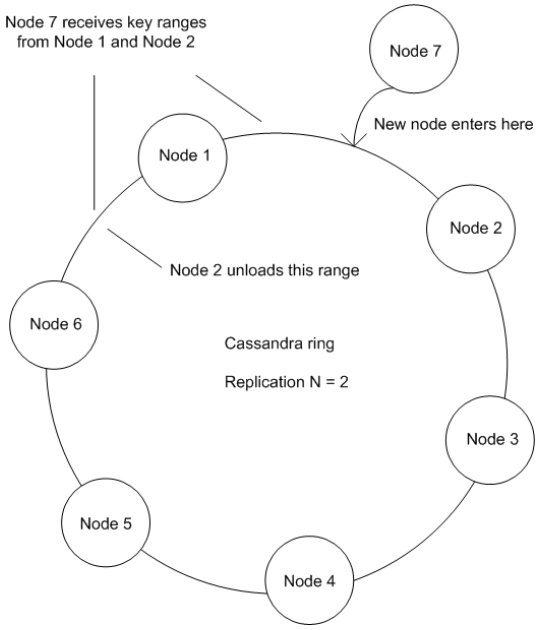


Figure 2.2: Data repartitioning in Cassandra.

different implementations for replica placement strategies [4]: Simple Strategy and Network Topology Strategy. The default strategy is Simple Strategy that places the first replica on a node determined by the partitioner and the other replicas are placed on the next nodes clockwise in the ring without considering rack or data center location. On the other hand,the Network Topology Strategy is used when there are multiple data centers per cluster. This strategy specify how many replicas you want in each data center. Cassandra partitions the key space onto system nodes using consistent caching. When a new node enters the system (e.g., Node 7 in Figure 2.2) data movement takes place only between the neighbors of the new node. For example in the setup of Figure 2.2 (where replication factor is equal to two), Node 7 will receive data from Nodes 1 and 2 and Node 2 will drop all data from the key range (between Nodes 6 and 1) no longer served by it.

2.4.2 Read and write operations

Any node may read or write data into the Cassandra cluster([4],[14]). When a client connects to a node and issues a read or write request, that node serves as the coordinator for that particular client operation. The basic role of coordinator is to act as a proxy between client and nodes or replica nodes that own the data being requested.

Furthermore, in the implementation of Apache Cassandra there is a utility which is a command line interface, named *nodetool*, that helps to manage a cluster. The nodetool utility has a number of options, for display statistics for every *keyspace* and *column family*, display compaction statistics, for telling a live node to decommission itself (streaming its data to the next node on the ring), enable or disable gossip protocol or thrift server, flush memtables to disk, enable/disable row cache or key cache capacity, etc.

Write operations

A responsible Cassandra cluster first writes the new record into a stable commit log (synchronously, or by explicit user choice, asynchronously) and then appends it to an in-memory buffer (*memtable*). When memtables reach a certain size (or at regular intervals) they are written to ordered SSTable files on disk. Write performance is normally unaffected by SSTable creation activity, unless write traffic exceeds the ability of a Cassandra node to buffer while writing to disk. Cassandra performs a number of background operations that may at times affect a node's response time, namely compactions that merge SSTables into fewer and larger files. Taking write intensity of a workload into account, one has to factor in (amortize) the periodic costs of compaction into the average cost of writes.

Read operations

The Cassandra read path starts at the client. A client can send operations to any node in the cluster and becomes the coordinator for that operation. The coordinator node contacts a configurable number of replicas to perform the read or write operation. A read first looks up a *row cache*, then a *key cache*, and then (if it misses in both caches) it tries to locate the key/value pair in the node's underlying storage system. In the worst case, a number of SSTables will have to be brought in memory to find the requested key. The cost of this path involves a number of network hops (depending on whether the coordinator is also one of the replicas serving the key sought) and disk accesses (none if we have a hit in the row cache), one or more if we either hit in the key cache or have to bring in indices from SSTables on disk. The use of compactions and Bloom filters narrows down the choice among SSTables, reducing I/O operations. Disk accesses in Cassandra go to a local file system.

Row and key caches

Cassandra's built-in key and row caches can provide very efficient data caching. Several Cassandra's clients use these two types of caches in order to improve read performance. The key cache holds the location of keys in memory on a per-column family basis. Each hit on a key cache can save one disk seek per SSTable. Unlike the key cache, the row cache holds the entire contents of the row in memory. Row cache is the best choice when you have data that is frequently accessible. Row caching saves the system from performing any disk seeks when fetching a cached row. Figure 2.3 depicts the path of two read operations. The one read operation hits the row cache, returning the requested row. The second read operation requests a row that is not present in the row cache but is present in the key cache. After accessing the row in the SSTable, the system returns the data and populates the row cache with this read operation.

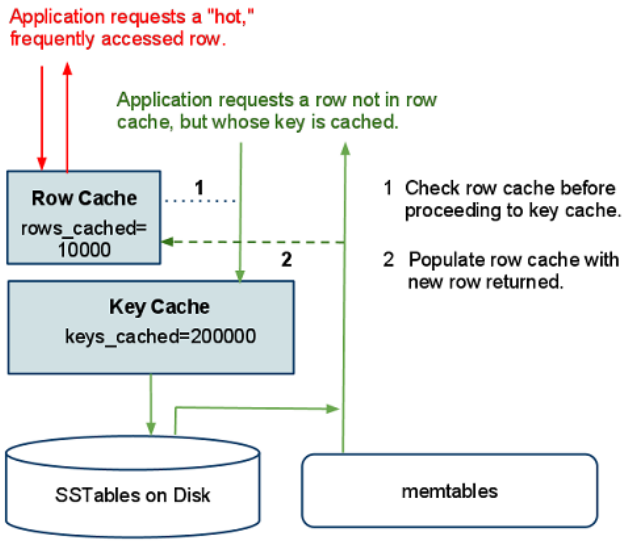


Figure 2.3: Two read operations in Apache Cassandra [4].

2.4.3 Apache Thrift

Cassandra's client API is built in the top of Apache Thrift API. Apache Thrift [16] is a library that offers the ability for creating high-performance services that can be called from multiple languages. It offers the ability to generate code to be used to easily build remote procedure calls (RPCs) for clients and servers that

communicate seamlessly across different programming languages. For that reason, Apache Thrift API creates a set of files that are used to create clients and servers. In addition, Apache Thrift is efficient through a unique serialization mechanism, in order to achieve interoperability. In more details, there is a software stack for creating clients and servers. Figure 2.4 depicts this stack. The top level of the stack is the generated code produced by Thrift definition file. In the two below levels, the Apache Thrift generated code results to Thrift services and data structures. Moreover, the protocol and transport are part of the Apache Thrift runtime library. Apache thrift also includes an infrastructure to tie the protocols and transports together.

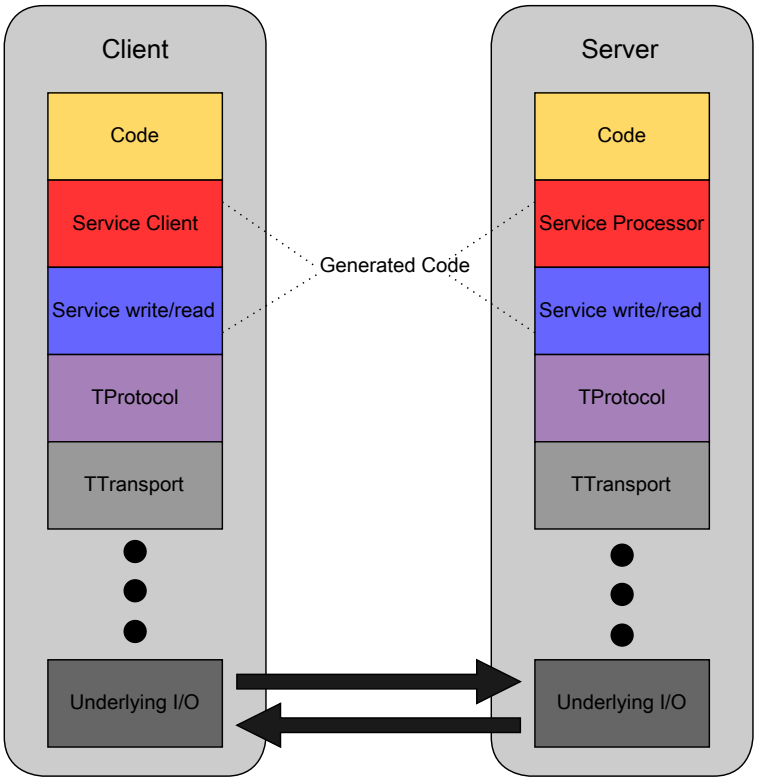


Figure 2.4: The software stack for creating clients and servers using Apache Thrift.

2.5 Load generator: YCSB

In this thesis, we use Yahoo! Client Serving Benchmark as a load generator. Yahoo! Client Serving Benchmark (YCSB) [17] is a framework and includes a common

set of workloads for evaluating the performance of different "key-value" stores, e.g. Cassandra, HBase, MongoDB, etc. This project comprises with a client, which is a workload generator and core workloads, that is a set of workload scenarios to be executed by the generator. YCSB performs 1KB accesses with configurable read or write ratio using Zipf or uniformly-random probability distributions.

Especially, in uniform distribution the items are uniformly random. That means, all rows are equally likely to be chosen. On the other hand, in Zipfian distribution the items are chosen according to the Zipf's distribution. That means some records will be extremely popular (which is the head of the distribution) while most records will be unpopular (that is the tail of the distribution).

In the core package of YCSB [17] there is a variation of different types of workloads. In these types of workloads, there is a table of records with a number of fields. Each record is identified by the row key, which is a string like *user234123*. Each field is named *field0*, *field1* and so on. The values of each field are a random string of ASCII characters of length L.

Chapter 3

Architecture

Our architecture for service-level management over the Cassandra distributed storage system is depicted in Figure 3.1. In the Figure 3.1, the existing components are with solid boxes and the dotted boxes denote our extensions. In this chapter, we describe the main aspects of our infrastructure, for monitoring, auto elasticity and provisioning. For these reasons, we developed a run-time adaptation to user-specified performance goals. The QoS controller is the core component of this architecture. Its key functionalities are to:

- setup SLAs with application clients, requesting their column families profiles (data set size and a coarse characterization of their degree of locality, such as random, zipf-like, etc. per column family) and performance requirements (currently focusing on satisfying response-time targets at certain throughput rates)
- effect initial resource allocations for the application
- periodically collect monitored response-time and throughput metrics from Cassandra clients and plan and effect changes in resource allocations to better align with requested targets
- perform admission control by estimating overall resource utilization and level of satisfaction of requirements for current commitments.

In this thesis, we primarily focus on the problem of initial resource allocation. We describe a provisioning policy based on predictions of service capacity requirements for applications. Our methodology relies on a set of performance tables (such as Table 3.1) of measured performance results (throughput and response time) produced by a configurable load generator. We use as load generator the Yahoo Client Serving Benchmark (YCSB) [17], 2.5 configured to produce a specific

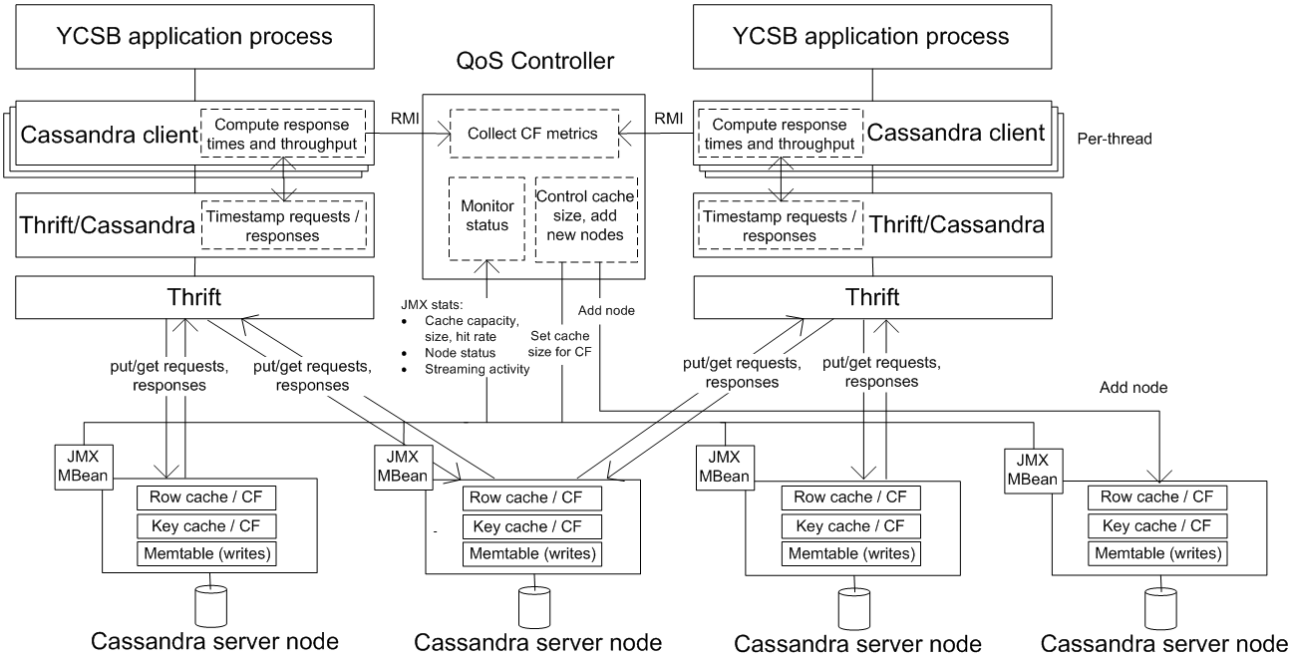


Figure 3.1: The Cassandra QoS architecture.

access pattern, I/O size, and read/write ratio (collectively termed a workload W) and server (VM) type (S).

	Workload: W; Server type: S				
# Servers / # Clients	1	2	3	4	...
clients_1	$r_1,\ t_1$	$r_2,\ t_2$	$r_3,\ t_3$	$r_4,\ t_4$	$r_5,\ t_5$
clients_2	$r_1,\ t_1$	$r_2,\ t_2$	$r_3,\ t_3$	$r_4,\ t_4$	$r_5,\ t_5$
clients_3	$r_1,\ t_1$	$r_2,\ t_2$	$r_3,\ t_3$	$r_4,\ t_4$	$r_5,\ t_5$
...	$r_1,\ t_1$	$r_2,\ t_2$	$r_3,\ t_3$	$r_4,\ t_4$	$r_5,\ t_5$

Table 3.1: Response time, throughput for variable load levels, service capacities (for given workload, server types).

Given an application's access pattern and desired load level, we can determine the number of servers of a given type required to achieve the desired throughput without exceeding a response time threshold. When the desired application characteristics or load levels do not exactly match a table entry we apply interpolation from neighboring table entries.

Each Cassandra client (usually embedded into an application) performs per-thread measurements of current throughput and of response time and computes exponentially-weighted moving averages (EWMAs) of response-time values using the following formula, where r*(T)* is the response time sampled at time T and α=0.125.

$$\mathrm{EWMA}(T) = (1 - \alpha) * \mathrm{EWMA}(T - 1) + \alpha * \mathrm{r}(T)$$

Each process computes response time EWMA and aggregate throughput across all its threads and communicates both to the QoS controller. The QoS controller combines the reported metrics across YCSB processes belonging to the same user. For a particular user, the total value of throughput across YCSB processes are the sum of the separate YCSB processes for the specific cassandra cluster and the total value of response time is the average of the EWMA response time of the YCSB processes.

The QoS controller is able to simultaneously interface and control multiple independent users (representing different workoads or different clusters). Since multiple applications executing over the same Cassandra cluster cannot normally be isolated in terms of elasticity policies (e.g., allow application A grow the Cassandra cluster by one server while application B sees its previous configuration), our design assigns each application to its own independent Cassandra cluster (still allowing Cassandra servers to share Cloud VMs) as shown in Figure 4.1. Finally, for high availability we support a primary-backup scheme using a shadow QoS controller to replicate the state of the primary. Details of this approach are beyond the scope of this thesis.

Chapter 4

Implementation

Our implementation is based on Apache Cassandra version 1.0.10. On the other hand, YCSB (version 0.1.4) is used as the canonical example of an application throughout this section. But we would like to mention that the architecture is general and applies to any application that can run over the Cassandra client library. Our implementation extends YCSB Cassandra Client version 1.0.10 (hereafter referred to as CassandraClient10). A YCSB application accesses a single Cassandra cluster and column family and involves multiple concurrently executing load-producing threads. Each thread uses a unique CassandraClient10 object. On the other hand, the Cassandra server-side code is unmodified. For simplicity reasons, we assume single datacenter and rack, one column family per cluster, single seed node (simple snitch), random partitioner, and replication factor one. Morever, key cache is disabled and we set enable only row cache. In Figure 3.1 solid boxes denote existing components while dotted boxes denote our extensions and our implementation.

To support our enhanced functionality (QoS attributes of multiple users accessing different CFs and clusters), we have added the following YCSB command-line arguments:

- "-rt" : desired response time for read/write requests.
- "-throughput" : desired throughput for read/write requests.
- "-QoS Port" : RMI connection port between YCSB user and QoSController.
- "-QoS Name, IP" : RMI connection name between YCSB user and QoSController.
- "-QoS IP" : RMI connection IP address between YCSB user and QoSController.

- "-dataPort" : RMI port for data connection between YCSB user and Cassandra cluster (each user is mapped to a separate cluster).
- "-cf" : column family name for the cluster.
- "-distribution" : the statistical distribution of requests, which can be "uniform" for uniformly random or "zipfian" for Zipf distribution in order to have locality.

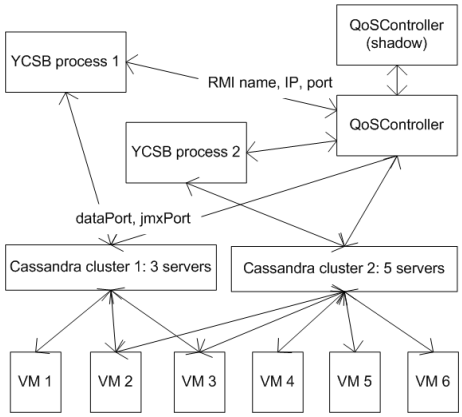


Figure 4.1: Relationships between application processes, QoS controller, Cassandra clusters, and the Cloud infrastructure. The shadow QoS controller forms a primary-backup pair with the main QoS controller for high availability.

In the following sections of this chapter, we present our implementation about monitoring, elasticity, caching and QoS controller process.

4.1 Monitoring

We modified the Cassandra Thrift implementation depicted in Figure 3.1 to timestamp read and write operations. As we mentioned above, the YCSB application has a number of concurrently threads, that is clients and each client has a unique CassandraClient10 object and has a Cassandra Thrift implementation. Each CassandraClient10 computes read or write latency (in ms) and read or write throughput (MB/s) (using the Cassandra Thrift timestamps). These values are stored them into a per-thread CassandraQoS object (that is per thread). The role of CassandraQoS object is to compute the EWMA of response times and estimates throughput by dividing the bytes transferred for completed operations over a given time period. That means, each thread has information about read or write latency

(EWMA response time) and read or write throughput of its requests.

Moreover, we have extended the YCSB implementation and we have added a *StatGathering* thread that periodically (every 30 seconds) collect from all CassandraQoS objects their response-time EWMAs and throughput values. *StatGathering* computes the average of EWMAs and aggregate throughput across YCSB threads. These values are the total EWMAs response-time and throughput per YCSB application. QoSController collects those numbers from each YCSB process via periodic (every 30 seconds) RMI calls.

4.2 QoS controller

The QoSController is a separate process executing on a dedicated node. The QoSController maintains a separate thread for different cassandra clusters. To enable the QoS controller to simultaneously control independent YCSB workloads we:

- Allow each YCSB process to have access to a different Cassandra server cluster. To allow sharing of VMs across clusters, network ports (data and JMX) used by a cluster are set per YCSB process.
- Give each YCSB process a separate RMI connection to the QoSController.

Each YCSB process communicates with the QoSController over its assigned RMI name, IP, and port, and passes to it configuration information followed by the user's SLA. Configuration information includes parameters of its Cassandra cluster, which are the following:

- Name of cluster
- Initial number of Cassandra servers based on the provision tables and user's SLA
- Maximum number of Cassandra servers for that cluster
- IP addresses of all Cassandra servers
- The number and the IP of seeds Cassandra servers
- The data port for (write and read operations)
- The JMX port for monitoring and managing Cassandra servers

We would like to mention that Cassandra cluster initialized for a specific YCSB process takes as parameters the ports all servers should listen to, e.g. data port 9160, JMX port 7199 for *Cluster0*; data port 9161, JMX port 7198 for *Cluster1*; etc.

The QoSController starts JMX MBeans connections (in particular ports) to each Cassandra server in a cluster. For read-only workloads, QoSController, through JMX's connections, disables key cache and sets row cache capacity at 500MB per server. Furthermore, it collects periodically (every 30 seconds) statistics on row cache capacity, its current size, and its hit ratio per cassandra server. When row caches fill up (past a ramp up phase after an elasticity action), the QoSController goes over a 10 minute period during which it checks I/O response times and throughput (20 times) for compliance with the user-specified SLO. For a specific cluster, the QoSController thread collects EWMAs responses-time and throughput from YCSB processes. Then it calculates the average of EWMAs and aggregate throughput across YCSB processes.

As we already mention (2.4.2) the write path does not use row or key caching. For that reason, QoS controller checks I/O response times and throughput for a period over a 10 minutes (that is 20 times) for check the compliance of the user's SLO. We use the same formula, as read requests, in order to calculate the total response time and throughput at a specific time in write requests. If the SLO is violated, it decides to start a new server to further distribute the load in the cluster. That means, the elasticity policy is time based.

4.3 Elasticity

When the QoSController starts a new server in a cluster, it initiates a JMX connection to that server and uses it to set specific attributes for that server such as row cache capacity and to get information about the server it will stream data from to balance the ring. The new cassandra server receives data from the cassandra server that has the majority of the data in the ring in order to balance the load. When data streaming is complete, the new server transitions into normal mode in the cluster (from joining mode during bootstrapping phase) and is ready to receive client requests. At that point, a cleanup thread in the QoSController deletes keys from offloaded nodes using the JMX `cleanup` API.

A challenge we faced early on was that the standard version of YCSB (as of version 0.1.4) does not take elasticity into account: it statically binds to an initial set of Cassandra servers and cannot dynamically redistribute load to an expanding

cluster. In our implementation, we check in YCSB whether a new server has been inserted in the cluster (ring) before each read or write operation (by checking the *host* attribute in the list of properties) and if so we include the new server. When the new server is at normal mode the QoSController informs, through RMI the YCSB's clients. Each CassandraClient10 client stores data structures of all clients of this cluster, structures of all TCP protocols and TCP transports for all servers in order to use them when elasticity takes place. In addition, each CassandraClient10 re-selects the Cassandra server to which it binds to and sends its requests. To spread the client load uniformly over the servers we map each client to a server by taking the modulo of the client identifier over the total number of servers. This dynamic reassignment of clients to servers is performed each time a new Cassandra server enters the cluster.

4.4 Caching

At the Cassandra server side, we use fixed-size row caches per column family, set using the JMX `setCapacity` method exported by storage servers. In addition, we use the same JMX method to disable key cache per server. In our earlier versions of our implementation using JVM heap for cache memory we were careful regulating these caches to avoid exceeding a certain fraction of the total heap size. Our experience indicates that exceeding that limit triggers frequent garbage collection (GC) activity and leads to automatic cache-size reduction by Cassandra.

Our current implementation configures Cassandra servers to use off-JVM heap memory (and thus not GC'ed) for its caches (row cache, key cache, and memtable). We thus avoid some of the cache-related memory pressure effects that impact Cassandra performance in unpredictable ways. We do not however fully prevent such activity, which is inherent in Java implementations of data-intensive distributed systems.

Chapter 5

Evaluation

We evaluated our system on the Amazon Web Services (AWS) EC2 Cloud using two different Cassandra clusters. The first cluster (referred to as SMALL) consists of 7 servers of type AWS *m1.small* featuring 1 virtual core with Intel Xeon processors, 1.7GB DRAM, 160GB local (instance) storage. The second cluster (MEDIUM) consists of up to 5 servers of type AWS *m1.medium* featuring 1 virtual core with Intel Xeon processors, 3.75GB DRAM, 410GB local storage. In both cases, the server operating system is Linux Ubuntu 10.4.1 LTS, 64 bits. The Cassandra software version (baseline) is 1.0.10 using the OpenJDK 1.6.0-24 Java runtime environment with heap size of 1GB.

Our evaluation workload is the Yahoo Cloud Serving Benchmark (YCSB) version 0.1.4. The YCSB workload executes on an EC2 instance of type AWS *m1.large* (2 virtual CPUs, 7.5 GB DRAM, 840GB local storage). The QoSController process executes on a dedicated AWS *m1.small* EC2 instance. In addition, we have used the Random Partitioner (the default partitioning strategy using consistent hashing) for mapping rows to Cassandra servers and set the replication factor to one. For simplicity, we have one datacenter, one rack per cluster, and have disabled the key cache to focus on the characteristics of the row cache alone.

To exhibit our QoS-aware provisioning methodology we focus on two distinct types of applications: those that exhibit locality in table accesses and those that do not. We model both by configuring YCSB to produce accesses based on:

- a Zipf probability distribution. According to the Zipf distribution, some records are extremely popular while most records are unpopular.

- a uniformly-random probability distribution.

In the first part of our evaluation we produce instances of Table 3.1 by progressively expanding I/O path parallelism between the application and storage servers (via elasticity actions) as much as needed to match the selected workloads (no SLO is set in this phase). Our evaluation considers AWS *m1.small* and *m1.medium* types; our methodology however extends to coverage of other VM types. We use our extended version of the YCSB benchmark configured for 128, 256, and 512 concurrent client threads to produce read-only uniformly-random or Zipf workloads. In a real setting, the QoS controller would build a larger set of tables for better coverage. However the current set of points are sufficient for demonstrating our approach. YCSB is initially setup over a Cassandra cluster of two servers. Progressively, the QoS controller grows the cluster to five (MEDIUM) or seven (SMALL) servers[1].

5.1 Zipfian distribution

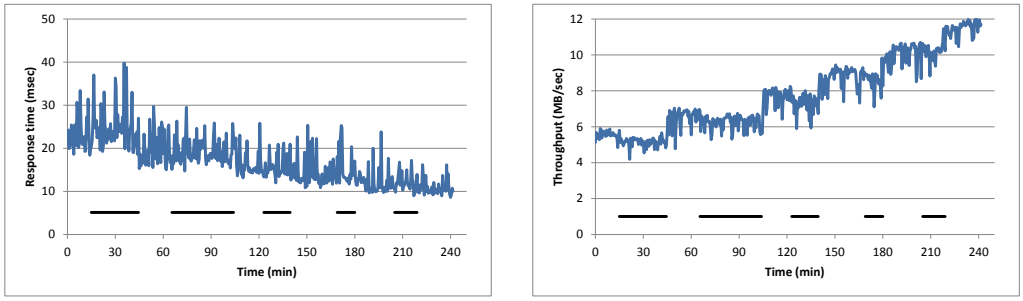
In the first set of experiments, we configure YCSB to produce a workload of ZIPF-distributed reads to 15 million 1KB records (a 15GB dataset). The QoS controller sets the row cache size to 500MB per Cassandra server on the initial cluster. The server row cache capacity is periodically checked (every 30 secs) through the JMX `getSize` method exported by storage servers. The QoS controller maintains the cluster size until all server row caches have filled up and then on for about 10 min. At that point the QoS controller triggers an elasticity action. In this phase, the QoS controller is configured to scale the system continuously as long as there is performance benefit from doing so.

Figures 5.1–5.3 depict EWMA response time (a) and throughput (b) in the SMALL cluster with 128, 256, and 512 concurrent client threads. Figures 5.4–5.6 depict EWMA response time (a) and throughput (b) for the MEDIUM cluster. The horizontal bars designate periods of data streaming during which a new (bootstrapping) node receives data from offloaded nodes. The sharp throughput drops in Figure 5.4b, 5.5b, and 5.6b are due to brief freezes observed in the Cassandra VMs (we are unaware of the cause of these freezes). Tables 5.1 and 5.2 summarize our results for the SMALL and MEDIUM clusters respectively (at steady state) for 128, 256, and 512 concurrent client threads. As cluster size grows, performance benefits come from increased I/O path parallelism as well as to the larger aggregate cache capacity available (each new server adds 500MB of cache in the cluster). Bootstrapping has a performance hit, but this is typically small due to throttling

[1] Our goal in sizing the two clusters was to provide comparable performance at their maximum capacity. Since *m1.medium* VMs are more powerful than *m1.small*, five *m1.medium* VMs are sufficient to match and exceed the maximum performance possible with seven *m1.small* VMs.

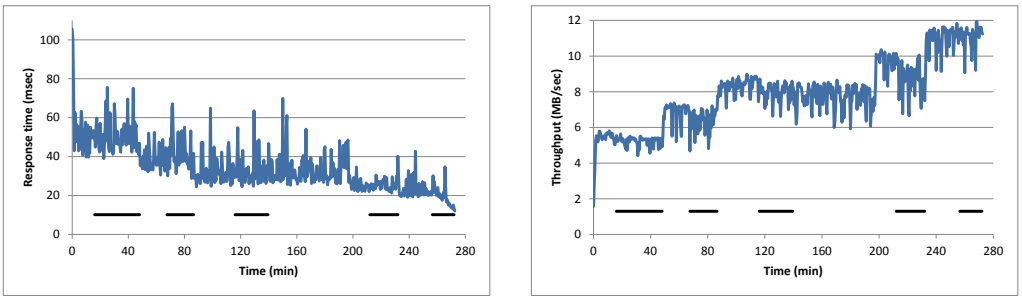
on streaming throughput applied by Cassandra.

Figures 5.8 and 5.9 show that the throughput increases and response time reduces proportionally with the number of Cassandra servers in both two clusters. As anticipated, the MEDIUM cluster can achieve a certain level of performance with fewer servers compared to SMALL. For example, a response time of 34ms for 512 client threads is achievable with either 7 *m1.small* VMs or with 4 *m1.medium* VMs. Similarly, for the same load and service capacity level the MEDIUM cluster achieves a higher level of throughput (up to 45%) compared to SMALL. Figure 5.7 depict the response time vs. offered-load relationships in the two clusters with growing service capacities. We also observe that results with the SMALL cluster exhibit higher variation compared to the MEDIUM cluster; additionally, variation grows within the SMALL cluster with increasing load level. This can be attributed to the fact that average CPU utilization is lower (and thus more CPU cycles available to absorb spurious activity) in the MEDIUM cluster vs. the SMALL cluster.



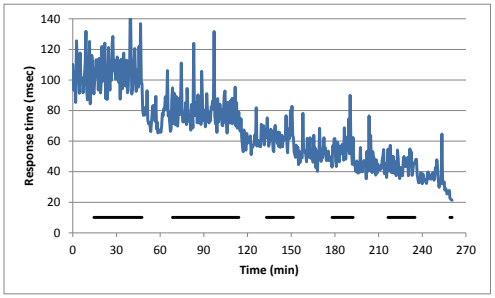
(a) Response time (b) Throughput

Figure 5.1: AWS m1.small, Zipf distribution, 1 client with 128 threads.

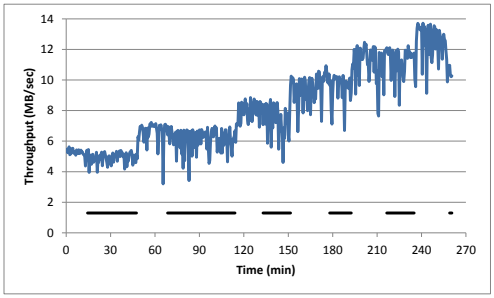


(a) Response time (b) Throughput

Figure 5.2: AWS m1.small, Zipf distribution, 1 client with 256 threads.

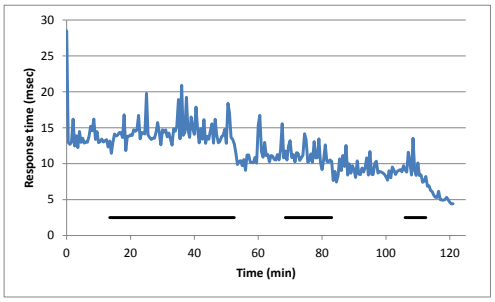


(a) Response time

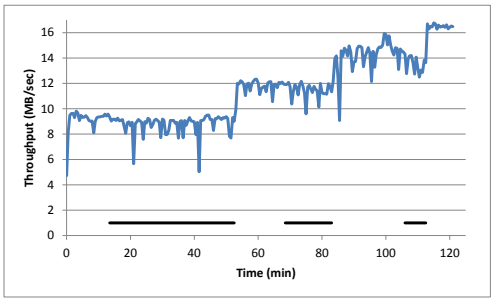


(b) Throughput

Figure 5.3: AWS m1.small, Zipf distribution, 1 client with 512 threads.

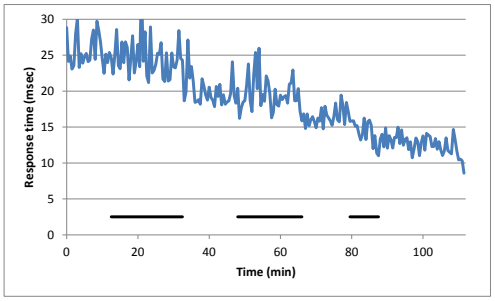


(a) Response time

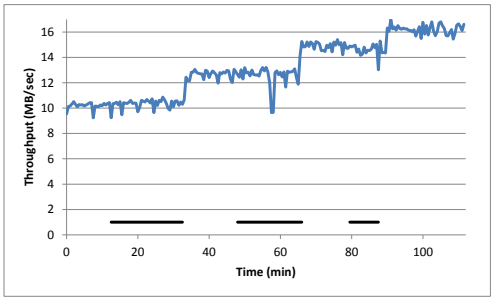


(b) Throughput

Figure 5.4: AWS m1.medium, Zipf distribution, 1 client with 128 threads.

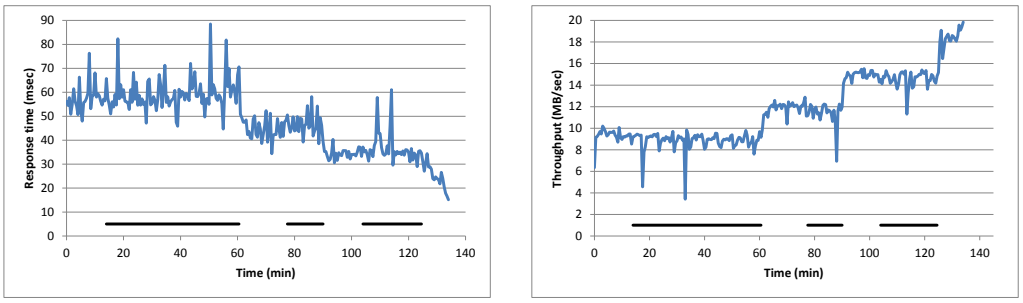


(a) Response time



(b) Throughput

Figure 5.5: AWS m1.medium, Zipf distribution, 1 client with 256 threads.



(a) Response time (b) Throughput

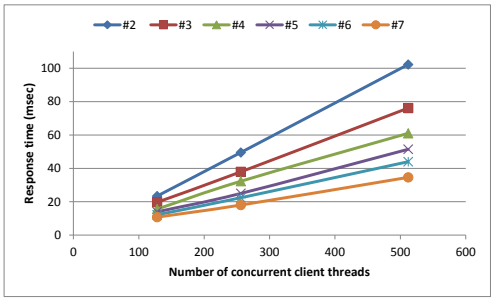
Figure 5.6: AWS m1.medium, Zipf distribution, 1 client with 512 threads.

# Clients \ # Servers	ZIPF-100% READS: AMAZON M1.SMALL					
	2	3	4	5	6	7
128	23.4, 5.5	19.7, 6.7	15.6, 7.7	14, 8.8	12.2, 10.2	10.7, 11.6
256	49.5, 5.5	37.9, 6.9	32.3, 8.5	25, 9.6	22.4, 11.1	18, 11.1
512	102.2, 5.2	76.1, 6.6	61, 8	51.4, 9.8	44, 10.9	34.6, 12.2

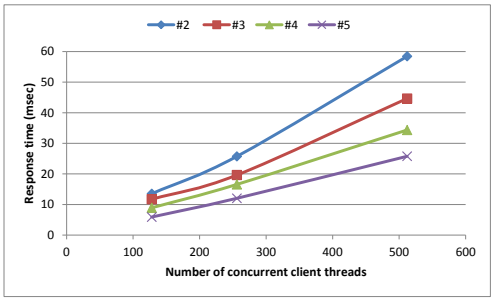
Table 5.1: Response time (ms), throughput (MB/sec) for ZIPF access pattern on Amazon's M1.SMALL.

# Clients \ # Servers	ZIPF-100% READS: AMAZON M1.MEDIUM			
	2	3	4	5
128	13.53, 9.27	11.73, 11.85	8.93, 14.7	5.89, 15.9
256	25.73, 10	19.64, 11.67	16.57, 14.88	12.02, 16.20
512	58.44, 9.33	44.63, 11.95	34.36, 15	25.77, 17.95

Table 5.2: Response time (ms), throughput (MB/sec) for ZIPF access pattern on Amazon's M1.MEDIUM.



(a) Amazon's M1.SMALL



(b) Amazon's M1.MEDIUM

Figure 5.7: Response time (ms) vs. offered load for different cluster capacities in the ZIPF workload.

(a) Response time

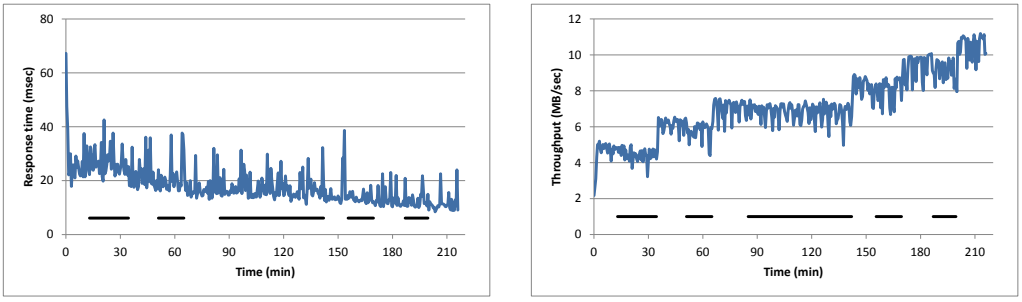
(b) Throughput

Figure 5.8: Response time (ms), throughput (MB/sec) for ZIPF on Amazon's M1.SMALL, having 128, 256, and 512 concurrent client threads.

(a) Response time

(b) Throughput

Figure 5.9: Response time (ms), throughput (MB/sec) for ZIPF on Amazon's M1.MEDIUM, having 128, 256, and 512 concurrent client threads.



(a) Response time (b) Throughput

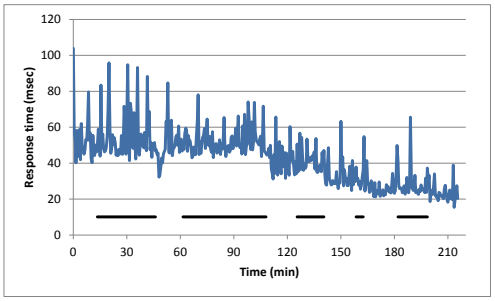
Figure 5.10: AWS m1.small, Uniformly random distribution, 1 client with 128 threads.

5.2 Uniformly random distribution

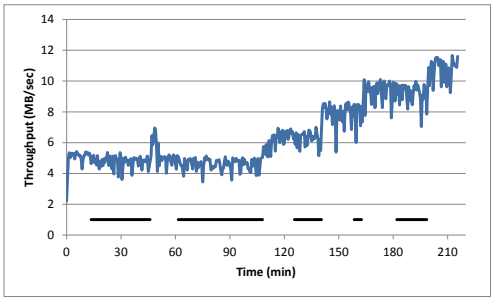
Similarly to the case of the Zipf distribution, we configure YCSB to produce a workload of uniformly-random reads over the same 15GB dataset with 128, 256, and 512 concurrent client threads. Figures 5.10–5.12 depict EWMA response time (a) and throughput (b) in the SMALL cluster configured to scale continuously as long as there is performance benefit from doing so. Figures 5.13–5.15 depict EWMA response time (a) and throughput (b) in the MEDIUM cluster. Tables 5.3 and 5.4 summarize our results from these experiments (at steady state). The throughput drop at Figures 5.14b and 5.15b is due to brief freeze of the Cassandra VMs and it is Cloud related. Figure 5.16 depicts the response time vs. offered-load relationships in the both cluster. Similar to the Zipf distribution, Figures 5.17 and 5.18 show that throughput increases and EWMA response time decreases with growing cluster size, although less so (up to 20%) due to smaller benefit from caching in this case (18% hit ratio vs. 60% for Zipf). The MEDIUM cluster can (just as in the case of Zipf) achieve a given level of performance with fewer servers compared to SMALL: a response time of 45ms for 512 client threads is achievable with either 7 *m1.small* VMs or with 4 *m1.medium* VMs. Similarly, the MEDIUM cluster achieves a higher level of throughput compared to SMALL for the same load level.

5.3 Validation of the methodology

At the initial stage of the YCSB benchmark the user sets up an SLA for the CF created and accessed by YCSB. In the SLA the user specifies the dataset size (15GB), degree of locality (ZIPF), the requested maximum average response time

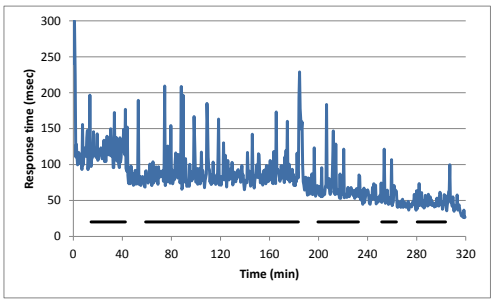


(a) Response time

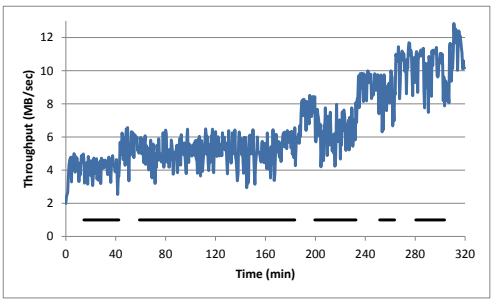


(b) Throughput

Figure 5.11: AWS m1.small, Uniformly random distribution, 1 client with 256 threads.

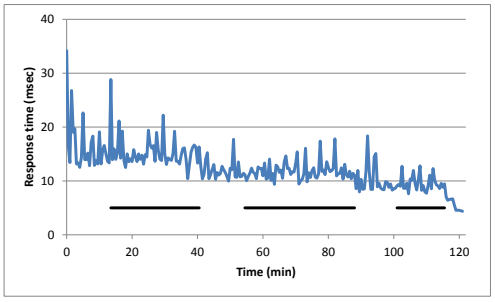


(a) Response time

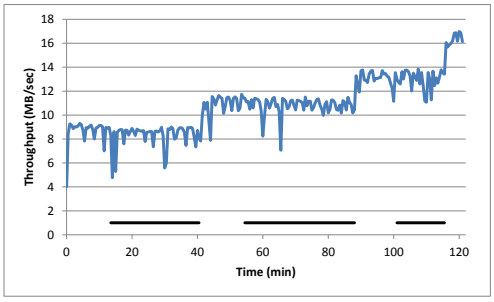


(b) Throughput

Figure 5.12: AWS m1.small, Uniformly random distribution, 1 client with 512 threads.

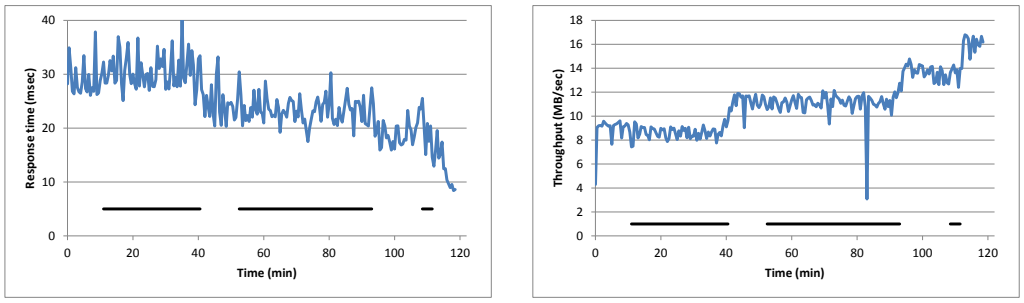


(a) Response time



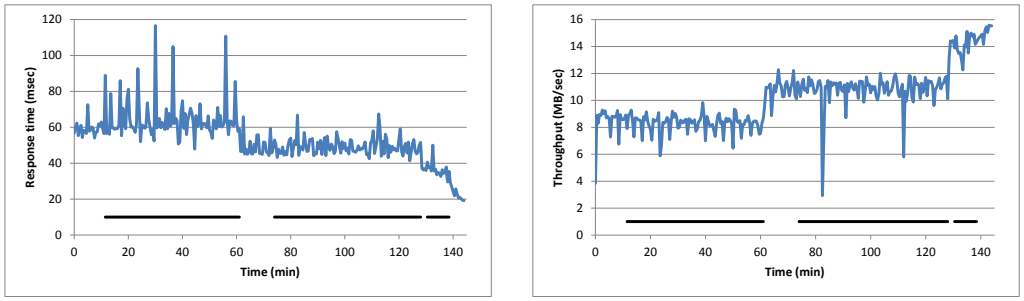
(b) Throughput

Figure 5.13: AWS m1.medium, Uniformly random distribution, 1 client with 128 threads.



(a) Response time (b) Throughput

Figure 5.14: AWS m1.medium, Uniformly random distribution, 1 client with 256 threads.



(a) Response time (b) Throughput

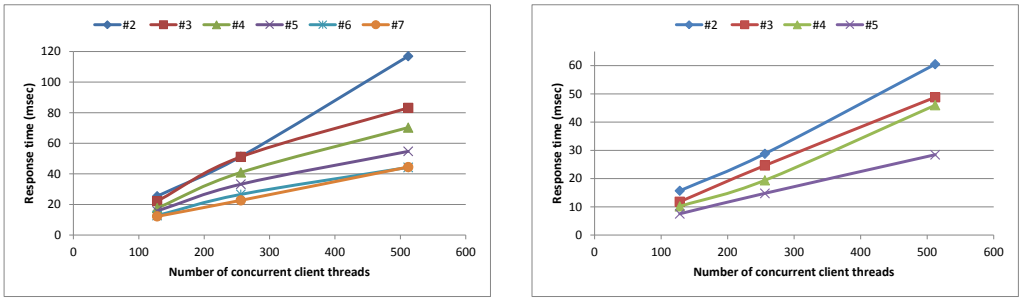
Figure 5.15: AWS m1.medium, Uniformly random distribution, 1 client with 512 threads.

	Uniform-100% Reads: Amazon m1.small					
# Servers / # Clients	2	3	4	5	6	7
128	25.4, 4.8	22.2, 6.1	17.6, 7.1	15.8, 8.2	12.8, 9.5	12.2, 10.4
256	51.3, 5.2	51.1, 4.9	40.9, 6.5	33.2, 7.8	26.6, 9.4	22.7, 10.7
512	116.9, 4.4	83.1, 5.6	70.3, 7.7	54.7, 9.2	44.2, 10.6	44.5, 10.6

Table 5.3: Response time (ms), throughput (MB/sec) for Uniform access pattern on Amazon's m1.small.

# Clients \ # Servers	UNIFORM-100% READS: AMAZON M1.MEDIUM			
	2	3	4	5
128	15.69, 8.78	11.8, 11.18	10.16, 13.06	7.52, 14.75
256	28.79, 9.05	24.64, 11.24	19.39, 13.51	14.77, 15.22
512	60.5, 8.64	48.8, 11.08	45.99, 12.77	28.44, 14.8

Table 5.4: Response time (ms), throughput (MB/sec) for UNIFORM access pattern on Amazon's M1.MEDIUM.



(a) Amazon's M1.SMALL

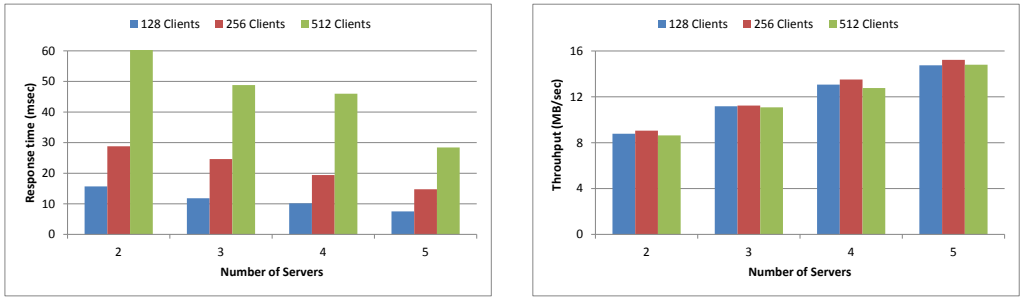
(b) Amazon's M1.MEDIUM

Figure 5.16: Response time (ms) vs. offered load for different cluster capacities in the UNIFORM workload.

(a) Response time

(b) Throughput

Figure 5.17: Response time (ms), throughput (MB/sec) for UNIFORM on Amazon's M1.SMALL, having 128, 256, and 512 concurrent client threads.



(a) Response time

(b) Throughput

Figure 5.18: Response time (ms), throughput (MB/sec) for UNIFORM on Amazon's M1.MEDIUM, having 128, 256, and 512 concurrent client threads.

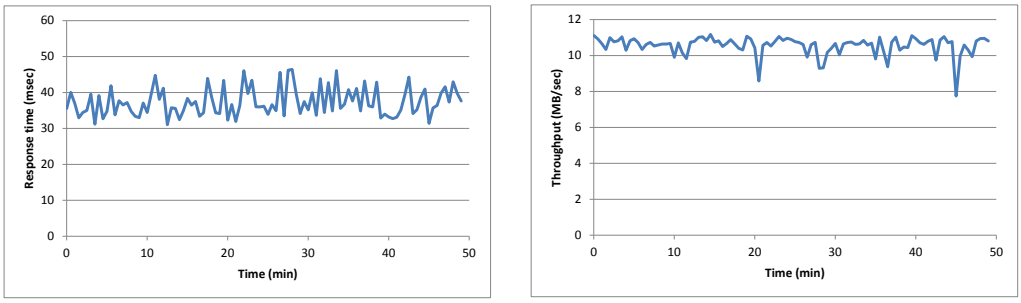
for read operations (40ms), an upper limit on throughput (384 threads), and row size (1KB). The QoS controller uses Table 5.1 to estimate the capacity to achieve the requested SLA. It uses weighted-average interpolation to produce the new row for 384 threads shown in Table 5.5. Other approaches to estimation have been explored in the past [18, 19]; a more thorough exploration of such techniques however is outside the scope of this thesis.

Using the predictions of Table 5.5, the QoS controller provisions a 5-node Cassandra cluster of EC2 *m1.small* type VMs, creates a CF on it and periodically monitors the achieved response time and throughput. Figure 5.19 shows that response time and throughput closely approximate the levels predicted by Table 5.5. Although average response time is below 40ms, throughput is slightly higher than expected (10.55 vs 9.72 MB/s). Since the user-requested SLA is achieved, the QoS controller does not trigger any further elasticity actions.

Although we have focused on read-only workloads in this thesis, we provide some insight to the characteristics of write workloads. Figure 5.20 depicts YCSB response time and throughput in an identical setup to the one used in Figure 5.19 (384 threads, 5 servers). We observe that response time is higher (61ms vs. 37.3ms) while throughput is lower (4.9MB/s vs. 10.6MB/s) with both metrics exhibiting more noise compared to the read-only workload. Cassandra's default write policy (unstable writes to commit log and memtable with periodic syncs to disk) largely decouples write performance from the disk device. However interference with frequent memtable/SSTable compaction activity (especially intensive in a 100% write workload) hurts performance due to increased I/O activity as well as increased CPU needs.

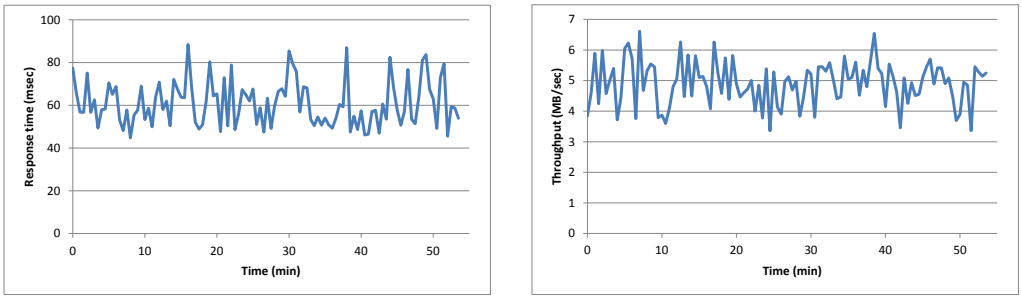
# Clients \ # Servers	ZIPF-100% READS: AMAZON M1.SMALL					
	2	3	4	5	6	7
256	49.5, 5.5	37.9, 6.9	32.3, 8.5	25, 9.6	22.4, 11.1	18, 11.1
384	*75.9, 5.3*	*57, 6.7*	*46.6, 8.2*	*38.2, 9.7*	*33.2, 11*	*26.3, 11.7*
512	102.2, 5.2	76.1, 6.6	61, 8	51.4, 9.8	44, 10.9	34.6, 12.2

Table 5.5: The row on 384 clients is a weighted average of the rows on 256 and 512 clients.



(a) Response time (b) Throughput

Figure 5.19: Zipf read-only distribution, 1 client with 384 threads, 5 servers.



(a) Response time (b) Throughput

Figure 5.20: Zipf write-only distribution, 1 client with 384 threads, 5 servers.

Chapter 6

Related Work

Distributed data stores (often referred to as key-value stores) that implement distributed tabular structures with configurable access semantics have recently been developed as research prototypes as well as commercial systems to support a number of rapidly-growing large-scale data-centric enterprises. Examples of such systems include Dynamo [6], Bigtable [7], and their open-source variants Cassandra [4] and HBase [5]. Cloud service offerings of these technologies are currently widely available, offering a broad range of performance and dependability characteristics.

As enterprises that have invested into Cloud computing are now raising their expectations from best effort to guaranteed levels of service, Cloud providers are beginning to offer versions of their data-centric services that support controlled performance, reliability, etc. Recently Amazon Web Services (AWS) announced two new versions of existing services that offer guaranteed read/write I/O throughput on a key-value store (this service is branded *DynamoDB*) and provisioned I/O throughput over its elastic block storage (service branded *provisioned IOPS*).

Providing quality of service over distributed storage has been an active area of research for at least two decades. Work at HP labs (a retrospective by John Wilkes provides a good overview of this work [20]) addressed a wide range of concerns, from specifications of workloads, QoS goals, and device capabilities, to mappings of workload onto underlying storage resources, and to run-time management of storage I/O flows.

Work by Goyal et al. [21] in the context of the CacheCOW system contributed algorithms for dynamically adapting storage cache space allocated to different classes of service depending on observed response time, temporal locality of reference, and the arrival pattern for each class. The focus of this work was on cen-

tralized storage controllers rather than distributed servers typically used in NoSQL systems. More recently, Magoutis et al. [22] presented a self-tuning storage management architecture that allows applications and the storage environment to negotiate resource allocations without requiring human intervention. The authors of this work aim to maximize the utilization of all storage resources in a storage area network subject to fairness (rather than user-defined service-level objectives, as we do in this thesis) in the allocation of resources to applications.

With AWS being the current industry leader in guaranteed performance over distributed Cloud storage, it is worth taking a deeper look into their published and commercial work. Their SOSP paper [6] describes their (internal at the time) Dynamo key-value data store service which offered service-level agreements (SLA) on the response-time of put/get operations (e.g., service-side completion within 300ms) offered by the service measured on the 99.9^{th} percentile of the total number of requests, assuming the client does not exceed a peak level of load (e.g., 500 requests / sec). The recently introduced DynamoDB [2] is based on the published design of Dynamo with the introduction of new technologies such as solid-state storage (SSD) to address reliability issues.

DynamoDB departs from the original Amazon design in its SLA specification. Namely, a user specifies performance requirements on a database table in terms of *request capacity* or number of 1KB read or write operations (also known as units of read or write capacity) desired to be executed per second. DynamoDB allocates dedicated resources to tables to meet performance requirements, and automatically partitions data over a sufficient number of servers to meet request capacity. If throughput requirements change, the user can update a table's request capacity on demand. Average service-side latencies for Amazon DynamoDB are reported to be in the single-digit milliseconds range [2]. Applications whose request throughput exceeds their provisioned capacity may be throttled. DynamoDB does not seem to provide any guarantees on the response time offered nor on the distribution of requests on which their offered performance is evaluated (e.g., 99.9^{th} percentile over some time range).

Two of the most widely deployed NoSQL distributed storage systems are HBase [5] and Cassandra [4]. HBase tables contain rows of information indexed by primary key. The basic unit of data is the column, which consists of a key and a value. Sequences of columns (an arbitrary number) collectively form a row. A number of logically-related columns can be grouped into column families (CFs), which are kept physically close in both memory and disk. HBase partitions data using a distributed multi-level tree that splits each table into Regions and stores Region

data in the HDFS distributed file system using a scheme similar to LSM trees [23].

Cassandra is an open source clone of Dynamo, combining some features (such as column families, and storage management based on LSM trees over local storage) from HBase. Each node in a Cassandra cluster maps to a specific position on a ring via a consistent hashing scheme [4]. Similarly, each row maps to a position on the ring by hashing its key using the same hash function. Each node is in charge of storing all rows whose keys hash between this node's position and the position of the previous n nodes on the ring when replicating n times. Cassandra leverages an LSM-tree like scheme similar to that used by HBase to store data except that individual files (called SSTables) are stored in each node's local file system as opposed to a distributed file system. When reading a row stored in one or more SSTables, Cassandra uses a row-level column index (and optionally a Bloom filter) to find the necessary blocks on disk.

The idea of measurement-based performance modeling has been previously proposed by Anderson [18] in the context of storage system design and configuration of RAID arrays [20]. Complex performance evaluations of systems have been studied by Westermann et al. [24, 19], including methods for predicting application performance based on statistics over a space of measurement data. Our methodology is closely related to these approaches as we rely on a guided exploration of system configuration space under different workload assumptions. We differ from them on our focus on service-level management of scalable and elastic NoSQL technologies such as Cassandra.

Chapter 7

Conclusions and Future Work

In this thesis, we describe a QoS infrastructure geared towards scalable NoSQL storage systems and current implementations of the infrastructure within the Apache Cassandra. Our evaluation of the Cassandra-based implementation controls of server-side caching as an effective solution to regulating application response time when the application exhibits strong data-access locality. Control over I/O path parallelism via elasticity mechanisms is a complementary and effective solution for matching user performance requirements. The impact of elasticity actions on performance varies depending on their intensity and the hardware characteristics of the underlying platform, warranting further investigation.

Moreover, we presented a methodology for QoS-aware provisioning of Cassandra clusters based on application SLAs. Our evaluation demonstrates that the methodology is effective in predicting server capacity requirements given simple application workload descriptions. Part of the simplicity of our approach stems from the scalability and elasticity mechanisms built into NoSQL systems such as Cassandra; we believe that our work is more broadly applicable to such systems.

A full exposition of write-intensive workloads and the handling of interference between workloads feature prominently on this list. Based on our experience with write workloads we believe that our methodology can straightforwardly extend to them. Workload interference has been deemed to be an important parameter in the past [20], especially over disk drives. We believe that with the proliferation of flash drive (SSD) technology, the importance of interference at the disk-drive level is less critical today than it was ten years ago. However, contention for other resources (CPU, memory) still needs to be taken into account in provisioning concurrent workloads over Cassandra clusters.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] Amazon Web Services, "DynamoDB," http://aws.amazon.com/dynamodb/, August 2012.

[3] "What is nosql?" http://www.mongodb.com/nosql.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," in *Proceedings of 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, Big Sky, MT, October 2009.

[5] Apache Software Foundation, "HBase," http://hbase.apache.org/, August 2012.

[6] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proceedings of 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[7] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[8] D. A. Menascé, "Qos issues in web services," *Internet Computing, IEEE*, vol. 6, no. 6, pp. 72–75, 2002.

[9] P. Bianco, G. Lewis, and P. Merson, "Service level agreements in service-oriented architecture environments," Software Engineering Institute, Tech. Rep. CMU/SEI-2008-TN-021, September 2008.

[10] A. Ludwig, P. Braun, R. Kowalczyk, and B. Franczyk, "A framework for automated negotiation of service level agreements in services grids," in *Business Process Management Workshops*. Springer, 2006, pp. 89–101.

[11] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web services agreement specification (ws-agreement)," in *Global Grid Forum*, vol. 2, 2004.

[12] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck, "Web service level agreement (wsla) language specification," *IBM Corporation*, pp. 815–824, 2003.

[13] "The ws-agreement protocol," http://packcs-e0.scai.fraunhofer.de/wsag4j/.

[14] "Datastax corporation, introduction to apache cassandra, white paper," http://www.datastax.com/wp-content/uploads/2012/08/WP-IntrotoCassandra.pdf.

[15] "Apache cassandra wiki," http://en.wikipedia.org/wiki/Apache_Cassandra.

[16] "The apache software foundation, thrift," http://thrift.apache.org.

[17] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC '10)*, Indianapolis, IN, June 2010.

[18] E. Anderson, "Simple table-based modeling of storage devices," HP Laboratories, Tech. Rep. HPL–SSP–2001–4, July 2001.

[19] D. Westermann, J. Happe, R. Krebs, and R. Farahbod, "Automated Inference of Goal-Oriented Performance Prediction Functions," in *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, September 2012.

[20] J. Wilkes, "Traveling to Rome: A retrospective on the journey," *Operating Systems Review (OSR)*, vol. 43, no. 1, pp. 10–15, January 2009.

[21] P. Goyal, D. Jadav, D. S. Modha, and R. Tewari, "CacheCOW: QoS for Storage System Caches," in *Proceedings of 11th International Workshop on Quality of Service (IWQoS 03)*, Monterey, CA, June 2003.

[22] K. Magoutis, P. Sarkar, and G. Shah, "OASIS: Self-Tuning Storage for Applications," in *Proceedings of 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST)*, College Park, MD, May 2006.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[24] D. Westermann, J. Happe, M. Hauck, and C. Heupel, "The Performance Cockpit Approach: A Framework For Systematic Performance Evaluations," in *Proc. of 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Lille, France, September 2010.