Adaptive Quorums for Cloud Storage Systems

Pilana Withanage Gayana Ranganatha Chandrasekara

Dissertation for the Degree of Master of
Information Systems and Computer Engineering

Jury

President:	Prof. Doutor Miguel Nuno Dias Alves Pupo Correia
Advisor:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Member:	Prof. Doutor Rodrigo Seromenho Miragaia Rodrigues

July 2015

Acknowledgements

I am obliged to my advisor, Professor Luís Rodrigues for giving me the opportunity to work on this thesis under his supervision. During this period I could gain a vast experience regarding how to do fruitful research. His advices and motivation is the key to produce this thesis.

Moreover, I would like to thank Prof. Paolo Romano, Manuel Bravo, Maria Couceiro and João Paiva for all of their valuable ideas and instructions in order to make my thesis a success. Specially I should be thankful to Nuno Diegues for all the hardships he went through in order to provide a better infrastructure to perform all the experiments.

A special thanks also to my family and friends for their support during this thesis.

Lisboa, July 2015

Pilana Withanage Gayana Ranganatha Chandrasekara

For my family,

Resumo

Os sistemas de armazenamento na nuvem usam replicaąão para aumentar a fiabilidade; são mantidas várias cópias de cada objecto de forma a assegurar que os dados permanecem acessíveis na presença de falhas. Os sistemas de quorums emergiram como uma das ténicas mais promissoras para garantir que os clientes observam um estado coerente mesmo quando algumas das réplicas estão inacessíveis ou lentas. Trabalho anterior demostrou que o desempenho de um sistema de quoruns depende da caracterizaąão da carga, e que ganhos significativos podem ser obtidos através da seleção criteriosa dos quoruns de escrita e de leitura. Neste trabalho, estamos interessados em sistemas de armazenamento com diferentes consumidores e padrões de carga diversos. Nestes sistemas é pode ser vantajoso usar quoruns diferentes para objectos diferentes. No entanto, as vantagens que dai advêm necessitam de ser poderadas com os custos inerentes à manutenção de meta-dados para um largo número de objectos. Para equilibrar estas duas forças, esta tese propõe uma nova estratégia que consiste em identificar os objectos acedidos com maior frequência de forma a ajustar individualmente os quoruns apenas para estes objectos, tratando os restantes por atacado. Foi desenvolvido um protótipo do sistema resultante que foi avaliado experimentalmmente de modo a aferir as vantages da estratégia proposta.

Abstract

Cloud storage systems rely on replication for reliability. Typically, each data object is stored in multiple nodes to ensure that data remains available in face of node or network faults. Quorum systems are a practical way to ensure that clients observe consistent data even if some of the replicas are slower or unavailable. Previous work has shown that the performance of a quorum based storage system can vary greatly depending on the workload characterisation and that significant gains can be achieved by carefully selecting the size of write and read quorums. In this work we are interested in multi-tenant storage systems, that are faced to heterogeneous works. In these systems, for optimal performance, different quorums may need to be applied to different data. Unfortunately, keeping different quorum systems for different objects significantly increases the amount of metadata that the storage system needs to manage. The challenge is to find suitable tradeoffs among the size of the metadata and the performance of the resulting system. The thesis explores a strategy that consists in identifying which tenants and/or objects are the major sources of bottlenecks in the storage system and then performing fine-grain optimization for just those objects, while treating the rest in bulk. We have implemented a prototype of our system and assessed the merits of the approach experimentally.

Palavras Chave
Keywords

Palavras Chave

Armazenamento na Nuvem

Desempenho

Tolerância a Faltas

Sistemas de quorum

Adaptação dinâmica

Keywords

Cloud Storage

Performance

Fault-tolerance

Quorum Systems

Dynamic Adaptation

Contents

List of Figures

List of Tables

Acronyms

ML Machine Learning

RM Reconfiguration Manager

AM Autonomic Manager

SDS Software Defined Storage

KPI Key Performance Indicator

1 Introduction

The number of cloud-based applications that require the storage of large data sets keep increasing. Notable examples include Google, Facebook, Twitter, among many others, but this trend is common to many other applications. As a result, distributed storage systems are a fundamental component of any cloud-oriented infrastructure and have received a growing interest, with the emergence of many private, commercial, and open source implementations of this abstraction. Apache Cassandra (Lakshman and Malik 2010), Amazon Dynamo (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels 2007), Hazelcast (Hazelcast 2008), Infinispan (Infinispan 2009), Openstack Swift (Openstack-Swift 2009) are some relevant examples of cloud storage systems that aim at providing high availability and throughput to the applications.

Most of the cloud storage systems, such as the ones listed above, have several configurations parameters those need to be carefully tuned for optimal performance. Naturally, the optimal configuration depends not only on application specific requirements (such as the intended reliability) but also of the workload characterization, such as the size of objects being read or written, distribution of the accesses to different objects, read/write ratio, among others (Wolfson, Jajodia, and Huang 1997; Couceiro, Ruivo, Romano, and Rodrigues 2013; Paiva, Ruivo, Romano, and Rodrigues 2013; Jiménez-Peris, Patiño Martínez, Alonso, and Kemme 2003). Furthermore, some of these factors may change in time, making hard of even impossible to resort to static or manual configuration. The dynamic adaptation of cloud storage systems, in face of observing changes in the workload, is therefore a challenging task that needs to be embraced (IMEX-Research 2015).

It is possible to find several works in the literature that address the dynamic adaptation of distributed storage systems. Some examples include the dynamic adjustments of the replication scheme of objects based on observed changes to the read/write access patterns (Wolfson, Jajodia, and Huang 1997), automated adaptation of the replication protocol based on current operational conditions (Couceiro, Ruivo, Romano, and Rodrigues 2013), dynamic data replica placement to improve locality patterns of data access with the help of machine learning techniques (Paiva, Ruivo, Romano, and Rodrigues 2013), and

techniques to optimize system wide quorum sizes for both read and write operations according to the read/write ratio (Jiménez-Peris, Patiño Martínez, Alonso, and Kemme 2003). Still, most of these approaches consider single-tenant systems, i.e., the case where the system is optimized as a whole. However, many cloud storage systems are multi-tenant, i.e., the same storage infrastructure is shared by different applications, each with its own specific requirements and workload characterization. Ideally, one could attempt to optimize the system for each and every tenant. Unfortunately, this may be too costly and induce a significant metadata overhead. Furthermore, the optimization of tenants that have a residual share of the total system usage may have no perceived gain provided that the overall system performance may be constrained by the configuration of the tenants that consume most of the systems bandwidth. This work address the dynamic adaptation of multi-tenant systems.

1.1 Motivation and Goals

Usage of distributed storage systems opens a large research arena to optimize the performance of the underlying system, monitoring various parameters such as access patterns, service level agreements, energy consumption etc. Based on our preliminary survey it was identified that even though the influence of quorum systems for the performance of cloud storage had been extensively researched, the effort to find the relationship between the access patterns and the optimal quorum configurations was lacking. Hence, when performing initial baseline tests to find those relationships it turned up that there is no linear co-relation between those two factors which in fact triggered the use of black box machine learning techniques to infer the suitable quorums at run time.

These initial findings were the motive to create the first prototype Global Q-Opt by Maria Couceiro. The performance of Global Q-Opt is sub optimal and specific to uniform single workloads which is not that realistic. Hence, the motivation in this work is to brainstorm new ways to make this system capable of working in real use cases and to get the maximum benefits from the initial findings of previous related work.

This work addresses the problem of adapting, in an automated manner, the quorum configurations according to the access patterns of objects in a distributed cloud storage system in order to maximise the throughput. More precisely:

The thesis aims at researching, designing, and implementing a system that can automate

the task of performing fine-grain adaptation of the quorum configuration of the most heavily accessed objects in a cloud storage system.

1.2 Contributions

To achieve the goals above the thesis contributes with two novel techniques, namely.

- A combination of top-k algorithms (to detect hotspots) and machine learning techniques (to automate quorum selection) that allows to optimize the system in an incremental way.
- A new algorithm that allows to change the quorum configuration on-the-fly, with minimum overhead for the on-going requests.

1.3 Results

We have integrated our mechanism in the Openstack Swift cloud storage system and used this prototype to do an extensive experimental evaluation of the proposed techniques. As a result, this work has produced:

- An implementation for Openstack Swift
- An extensive experimental evaluation using the resulting prototype.

1.4 Research History

When I started this work, a prototype of a system that was able to perform some level of quorum adaptation for Openstack Swift had already been developed by Maria Couceiro. This system considered only aggregated values, and performed a global quorum optimization, by applying the same quorum configuration to all objects. The goal of my work was to improve and expand that prototype in different ways. First, I added the possibility of applying different quorum-configuration to different objects. Given the scalability problems associated with keeping fine-grain information for every object, I have also implemented top-k mechanisms, able to identify for which objects, fine-grain optimization would provide

more significant results. I have also performed several optimizations to the base Swift implementation that were not implemented in the prototype developed by Maria Couceiro. In particular, in the original Swift implementation, write requests were directed to all replicas, regardless of the write-quorum in use. This proved to be a bottleneck on performance, and I have made the required changes to take full benefit of the selected quorum configuration. In my work I have benefited from the collaboration of my advisor, Prof. Luís Rodrigues and other members of the GSD team, namely from Prof. Paolo Romano, Maria Couceiro and Manuel Bravo.

1.5 Structure of the Thesis

In Chapter 2 we present all the background related to our work. Chapter 3 describes the proposed architecture and its implementation. Chapter 4 presents results from a experimental evaluation. Finally, Chapter 5 concludes the report.

2 Related Work

In this chapter we survey the related work that is relevant to our work. We start by doing a brief overview of cloud storage systems, then we address the use of replication in this class of systems, and subsequently we discuss the use of quorum systems in this context. After this overview, we describe the system that served the basis for our work. Finally, we address other techniques that are relevant to our work, namely the systems to perform top-k analysis and machine learning.

2.1 Cloud Storage

Storage systems made for the cloud have important characteristics that distinguish them from traditional distributed file systems, such as NFS (Sandberg, Goldberg, Kleiman, Walsh, and Lyon 1985) or AFS (Morris, Satyanarayanan, Conner, Howard, Rosenthal, and Smith 1986), and classical database systems (Ullman 1990). Among these differences, one of the most important aspects is the need to support extremely large amounts of information and very large numbers of clients. This is usually achieved by distributing the storage and processing load by a large number of machines. We can distinguish two main types of cloud storage systems: file systems and key-value stores. File systems, such as Google File System (GFS) (Ghemawat, Gobioff, and Leung 2003), or its open source implementation, HDFS (Shvachko, Kuang, Radia, and Chansler 2010), are typically optimised for sequential access. Key-value stores, such as Amazon Dynamo (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels 2007), Cassandra (Lakshman and Malik 2010), among many others, strore key-value pair tuples, and support random access to these tuples, with different degrees of consistency. Furthermore, most of these system are implemented on commodity hardware, that is more cost efficient, but also more prone to failures, than high-end servers. Therefore, cloud storage system heavily rely on data replication for reliability and availability. In this thesis, we mainly focus on key-value stores. In the following paragraphs, we provide a brief description of some relevant key-value stores for the cloud.

2.1.1 Quorums for Cloud Storage

When multiple replicas of the same data are maintained in a system, one needs to address the problem of coordinating write and read operations on those replicas. Maybe one of the most intuitive approaches consists in updating all replicas every time a write operations occurs. If the appropriate protocols are used to coordinate concurrent updates, applying all updates to all replicas ensures that the replicas remain mutually consistent and up-to-date. As a results, read operation can be directed to any replica. This approach is used, for instance, in the Google File System (GFS) (Ghemawat, Gobioff, and Leung 2003). The drawback of this approach is that, if one of the replicas fails or becomes temporarily slower than the remaining replicas, the write operation may stall until the replica recovers or is re-constructed. In fact, long write latencies have been observed in the Google File System when chunkservers failed.

An alternative option consist in relying in a quorum-system. In a quorum-based system (Gifford 1979), the execution of a read or write operation on a data item requires to contact some subset, called a quorum, of the set of nodes that replicate that item (Lakshman and Malik 2010; DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels 2007). In fact, the approach described earlier can be seen is a particular case of a quorum system, where the write quorum includes all the replicas and the read quorum includes a single replica. The usage of a quorum-system that does not require to contact all replicas, can provide better availability, given that an operation may terminate even if one or more replicas are unresponsive. In fact, quorum systems are the cornerstone of a large number of techniques that address distributed coordination problems in fault-prone environments, including mutual exclusion, transaction processing, data replication, and byzantine fault-tolerance (Herlihy 1987; Dolev, Keidar, and Lotem 1997; Lakshman and Malik 2010; Jiménez-Peris, Patiño Martínez, Alonso, and Kemme 2003; Malkhi and Reiter 1997). A *strict quorum system* (also called a coterie) is a collection of sets such that any two sets, called quorums, intersect (Garcia-Molina and Barbara 1985). In many distributed storage systems a further refinement of this approach is employed, in which quorums are divided into read-quorums and write-quorums, such that any read-quorum intersects any write-quorum (additionally, depending on the update protocol, in some cases any two write-quorums may also be required to intersect).

In this thesis we are primarily concerned with the selection of strict quorum systems that can optimize the system performance. There is a significant body of research that focused on which criteria should be used to select a "good" quorum configuration, which studied the trade-off between load, availability, and probe complexity (Naor and Wool 1998; Peleg and Wool 1995). In practice, existing

cloud-oriented storage systems, such as Dynamo (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels 2007), Cassandra (Lakshman and Malik 2010) or OpenStack's Swift, opt for a simple, yet lightweight approach, according to which:

- each data item is replicated over a fixed set of storage nodes N, where N is a user configurable parameter (often called *replication degree*) that is typically much lower than the total number of the storage nodes in the system.
- users are asked to specify the sizes of the read and write quorums, denoted, respectively, as R and W. In order to guarantee strong consistency (i.e., the strictness of the quorums) these systems require that $R+W > N$.
- write and read requests issued by clients are handled by a *proxy*, i.e., a logical component that may be co-located with the client process or with some storage node. Proxies forward the read, resp. write, operation to a quorum of replicas, according to the current quorum configuration (i.e., reads are forwarded to R replicas and writes to W replicas). For load balancing, replicas are selected using a hash on the proxy identifier. A proxy considers the operation *complete* after having received the target R, resp. W, replies. If, after a timeout period, some replies are missing, the request is sent to the remaining replicas until the desired quorum is ensured (this fallback procedure, occurs rarely, mainly when failures happen).
- for read operations, after collecting the quorum of replies, a proxy selects the most recent value among the ones returned by the storage nodes. For write operations, storage nodes acknowledge the proxy but discard any write request that is "older" than the latest write operation that they have already acknowledged. These solutions assume that write operations are totally ordered, which is typically achieved either using globally synchronized clocks (Liskov 1991) or using a combination of causal ordering and proxy identifiers (to order concurrent requests), e.g., based on vector clocks (Lamport 1978) with commutative merge functions (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels 2007).

This scheme minimizes the load imposed on the system for finding a live read/write quorum in the good case given that, typically, the selected quorum of replicas that are contacted first are alive and do reply; only if faults occur, the requests need to be forwarded to all replicas. Further, thanks to the strictness property of the quorum system, and given the assumption on the total ordering of the write

Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family
Primary Key	Row Key
Column name	Column name/key
Column value	Column value

Table 2.1: Cassandra data model w.r.t relational data model

operations, this scheme guarantees that a read operation running solo (i.e., without concurrent write operations) returns the value produced by the last *completed* write operation.

2.1.2 Cassandra

Cassandra is a distributed, highly scalable cloud storage system initially designed by Facebook in order to manage their huge dynamic storage requirements ensuring low latency and high availability (Lakshman and Malik 2010). The design adapts different important properties from both Amazon Dynamo and Google Big Table (Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, and Gruber 2008). Cluster management, replication and fault tolerance features are captured from Amazon Dynamo while the features such as columnar data model and storage architecture are captured from Google Big Table.

Even though the Cassandra data model is based on key value storage, it extends this basic key value model with two level of nesting. So that it is capable of providing the flavor of relational data model on top of key value storage. As shown in the Table 2.1 it shows a close relation with the relational database model to the outside world and the Cassandra Query Language (CQL) is the wrapper used to hide the complexity of data manipulation in a complex object storage system to the clients.

In practice, Cassandra is deployed on top of commodity hardware, in multiple servers in a datacenter. Its capability of making a certain node to work independently eliminates the single point of failure. It adapts the consistent hashing technique to partition the data across the storage cluster, therefore the departure and arrival of a node only affects its immediate neighbors. As shown in the Figure 2.1, it creates a virtual ring and assigns a predefined set of partitions to each of the nodes to be taken care of.

Each node is collocated with a proxy module and a storage module. The proxy module is responsible for contacting the actual storage nodes (may be the node itself) in order to fulfil a read/write request.

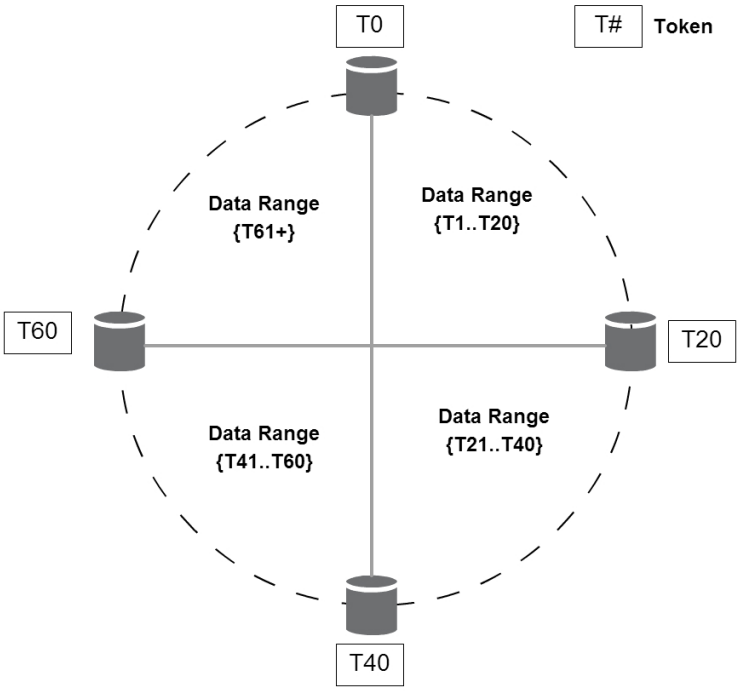

Figure 2.1: Cassandra data partitioning

Therefore, the proxy become the entrypoint and the interface to the outside world. For instance when the proxy receives a read request it will become the coordinator for that read request and contacts the replicas in order to satisfy the consistency guarantees requested by the user and gather the responses and send the response back to the client. Moreover, the write requests also handled in the similar manner. Because each node has a proxy module each of them can work as the entry point to the cluster and furthermore proxies follows the shared nothing architecture so that they can work independently.

It is important to capture the idea of how Cassandra handles read and write requests because these internal mechanisms have a direct impact on the behaviour of the whole system. As the Figure 2.2 depicts, when a write request arrives to a Cassandra storage node, first it writes to a commit log for durability and recoverability, then it writes to an in-memory data structure. In order to maximize the disk throughput, all the writes to the commit log are done sequentially. Once the in memory data structure exceeds a tunable limit, it dumps it to the disk. When a read request has arrived, it first checks the in-memory data structure. If it does not exist there, then it checks the disk. As a result of this way of writing sequentially to the disk Cassandra has achieved higher write throughput than the read throughput.

Cassandra has implemented many strategies to replicate the data locally and globally in order to make the data highly available. For instance, the “Simple Strategy” places the original row on a node

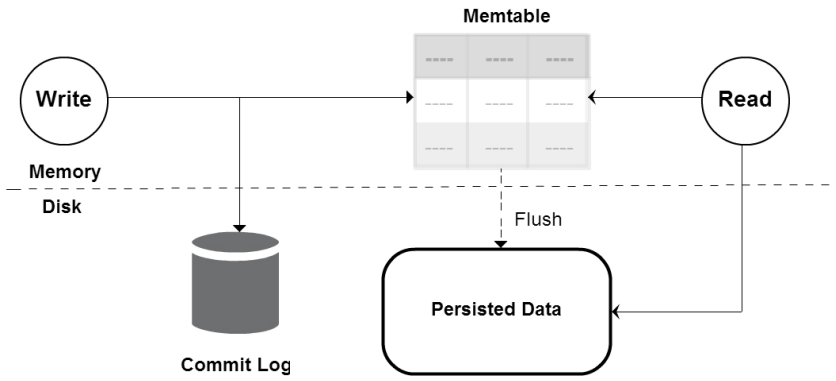

Figure 2.2: Cassandra disk read/write

determined by the partitioner. Additional replica rows are placed on the next nodes clockwise in the ring without considering rack or data center location. Contrary, the "Network Topology Strategy" allows for replication between different racks in a data center and/or between multiple data centers. This strategy provides more control over where replica rows are placed. In Network Topology Strategy, the original row is placed according to the partitioner. Additional replica rows in the same data center are then placed by walking the ring clockwise until a node in a different rack from the previous rack is found. If there is no such node, additional replicas will be placed in the same rack. To replicate data between multiple data centers, a replica group is defined and mapped to each logical or physical data center. This definition is specified when a keyspace is created in Cassandra.

Therefore, it is mainly designed to be eventual consistent and to provide high availability and partition tolerance. Cassandra has built in settings to specify how many replicas needs to be consulted before a request claimed to be successfully executed. Interestingly, it allows users to achieve strong consistency with the help of those existing settings specified as below,

- WRITE ALL + READ ONE
- WRITE ONE + READ ALL
- WRITE QUORUM + READ QUORUM

A write consistency level one (WRITE ONE) means that data needs to be written to the commit log and memory table of at least one replica before returning success to the client. Moreover a read operation with consistency level ALL (READ ALL) means that the read operation must wait for all the replicas to reply with consistent data in order to return the data to the client. For more clarity of both

strong consistent and eventual consistent scenarios Figure 2.3 shows how a client request is catered when system is setup to provide those consistency guarantees respectively.

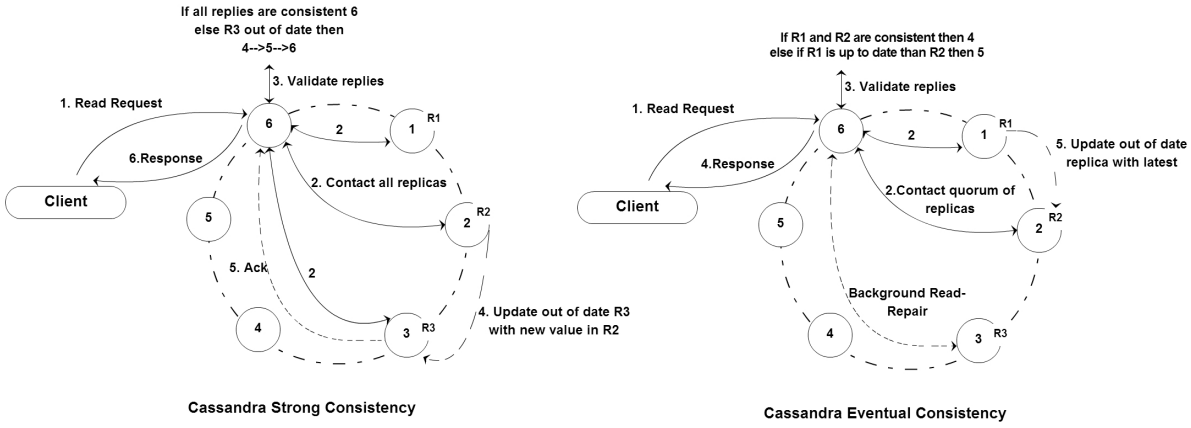

Figure 2.3: Cassandra strong and eventual consistency operation steps

2.1.3 Openstack Swift

Openstack Swift (Arnold 2014) is a redundant, scalable distributed object storage which is capable of managing a large amount of transactions guaranteeing the availability and the durability of data.

2.1.3.1 Swift Object Location

In order to keep the interactions more simple Swift introduces a RESTful API so that the object manipulation is easy to deal with the help of HTTP verbs. Furthermore, each object in the cluster has a unique url so that using the HTTP verbs such as GET, PUT, POST etc can be used to enforce the intended operation for the relevant object. For instance, this url 'https://Swift.example.com/v1/account/container/object' represents the format of such a url. Path segment 'Swift.example.com/v1/' represents the cluster location and meanwhile '/account/container/object' represents the storage location for an object.

- /account is the storage area where the list of metadata related to that account itself and the containers under that storage area is stored.
- /account/container is a storage area within the account which contains the metadata of the container itself and the objects under that storage area.

- /account/container/object is the storage location where the object and its metadata will be kept.

With the help of the above url format client applications can interact with them using below HTTP verbs,

- GET —downloads objects, lists the contents of containers or accounts
- PUT —uploads objects, creates containers, overwrites metadata headers
- POST —creates containers if they don't exist, updates metadata (accounts or containers), overwrites metadata (objects)
- DELETE —deletes objects and containers that are empty
- HEAD —retrieves header information for the account, container or object.

2.1.3.2 Swift Architecture

The overall system contains two types of nodes, proxy nodes and storage nodes. Figure 2.4 below shows the Swift architecture and how a request is catered. Proxy node contains the proxy process which is the interface to the external world and accepts client requests to manipulate data. Swift proxy nodes follow the shared nothing architecture so that multiple proxies can be established which serve different users but refers the same storage cluster.

Storage nodes are the servers those run multiple processes such as account, container and object. Account process keeps track of number of containers and the amount of storage has been occupied by the containers in a SQLite database. Container can be considered as directory created inside the account. There can be many containers such that container process tracks the number of objects and space used by them. (Important to note that containers do not know the location of the objects). Object process handles the actual objects and place them on the disk. In addition to these 3 main processes, storage node has several other complementary services such as replicator service which takes care of the object replication and maintenance, auditor service which checks for data corruption and repair.

Similar to many other promising object storage systems, Swift uses modified consistent hashing. This consistent hashing information is pre-generated for a certain cluster at the time of establishment and shared with all the nodes in the cluster (proxy and storage) which is called the 'Ring file'. Having this

Ring files distributed in all the nodes in the cluster, Swift has eliminated centralized data coordination becuase Ring files helps a node to find an object' s actual physical location deterministically.

This consistent hashing technique implies that it contains a fixed amount of partitions in a cluster and when the number of nodes are added or removed the amount of partitions held by a single node become decreased or increased respectively. Each node keeps two data structures in order to locate data in the cluster. First is the device list and second is the devices look up table. Device list contains the driver information such as ip, port, region, zone etc. Device lookup table is a table where a row represents a single unique replica and columns represent partitions. Hence it is a table with few rows and thousands of columns. Once a data placement algorithm decides the best place to keep a partition it updates this table. Therefore when a request arrive to read an object it will find the correct locations of it by this table and then check the device list to forward the request to it.

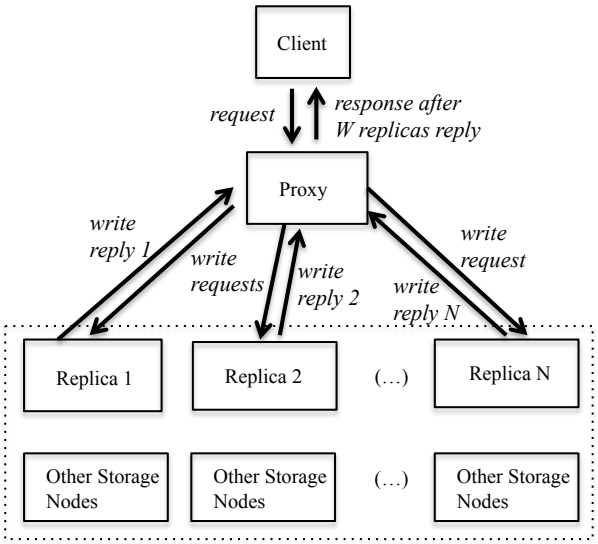

Figure 2.4: Openstack Swift Architecture

2.1.3.3 Swift Quorum Implementation

Openstack Swift is an eventually consistent object storage system where consistency is compromised in order to achieve the availability and partition tolerance. Therefore, it has a quorum configuration that helps the client applications to adjust the quorum settings based on the application requirement. For instance, if the application is prefered faster write operations but does not affected in case of data loss a configuration with write quorum 1 would be the best choice. With write quorum 1, Swift returns the write request as successful as soon as one of the replicas acknowledged the write operation. In case of a

storage node failure it is possible to observe a data loss because it only persists data to one of the replicas at the time it return success. But this event is highly unlikely to happen because the Swift replicator services are proved to be update those objects with in few milliseconds. For read intensive systems it is preferred to have a lower read quorum so that reads are much faster. In this way, Swift is capable of providing different flavours of object accesses based on the client requirements.

Swift Read Quorum implementation: Because it aims at eventual consistency, Swift can fulfil a read request if it has at least one correct response. Therefore, Swift read quorum implementation works lazily such that it sequentially query the replicas. It is highly unlikely that the very first read request would not yield correct results so that this sequential behaviour is not a bottleneck in current (original) Swift implementation. But in our goal of providing strong consistency by varying the read/write quorum sizes, this original implementation makes a huge overhead for read operations.

Swift Write Quorum implementation: Unlike in other similar quorum systems, Swift sends the write request to all the replicas in parallel and collects the replies until it satisfies the user specified write quorum size. This behaviour helps all the replicas to detect the object update quickly but at the same time generates an additional overhead in the network because each write request to an object spawns new connections to all of its replicas. It is worthy to improve the way of manipulating write requests in order to achieve better performance.

2.2 Quorum Adaptation

We aim at exploring opportunities for performing quorum adapation in a scalable manner for systems where objects are subject to diverse workloads. We now describe some system that address the autonomic configuration of quorum systems.

2.2.1 An Application-Based Adaptive Replica Consistency for Cloud Storage

As discussed, cloud storage often requires to distribute data across the globe in order to achieve high availability. As a result, the data may be inconsistent for a certain period of time until it propagates to all the replicas. Furthermore, applications developed on top of such distributed storage may also have different consistency requirements that are defined at run time. For instance, we may consider an auction system where the consistency of the prices the users see may not require full consistency until few

hours/minutes of the deadline of the auction. Then, when the auction is about to close, the application requires strong consistency to make the end users to enter the proper next bid.

Wang, Yang, Wang, Niu, and Xu (2010) propose a system where the consistency of the data is determined based on the requirement demand by the application rather than enforcing a consistency level for all the data in the cluster as a global configuration. The system uses the read and write frequency and the variation of these two parameters as the key input matrix to trigger automated consistency adaptation. Furthermore, the authors propose a hierarchical structure for the storage nodes in the cluster as shown in Figure 2.5. Initially all the nodes in the cluster are same and then when the system initializes one node is selected as the master and three other nodes as deputy nodes. Deputy nodes contain all the data as the master which works as the back up for the master node. Furthermore, deputy nodes are selected based on the geographical location. The rest of the nodes are child nodes and their main objective is to make the services available by handling application requests from different locations of the globe. If a deputy node is down, one of the child nodes in the same region would become the deputy node in order to maintain the balance of the hierarchy.

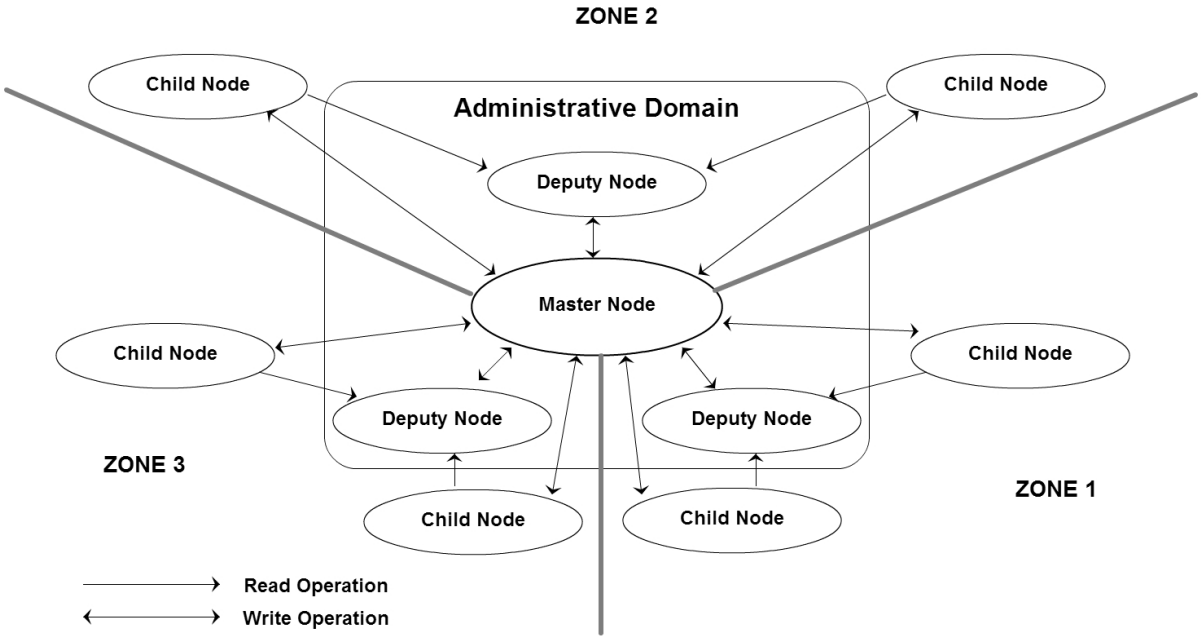

Figure 2.5: Model Structure

In the proposed model, all the nodes are capable of receiving operations from the client application but only the master node can handle the update requests. When a child or deputy node is received an update request it should be forwarded to the master because master decides where it needs to be replicated

based on the access frequency from that application. For read requests either child or deputy node can reply. If a child node does not have requested data it will fetch it from the nearest deputy node.

Based on the read/write access frequencies, the authors define 4 categories where the replication strategies are varied for each of them when a update request is handled by the master node.

- *Category 1:* the read frequency is high and the update frequency is low. Intuitively, because higher amount of operations are reads, they demand the strong consistency so that endpoints would not read old data. Even though the writes to all the replicas is expensive, due to the low amount of write operations cost of pushing updates to all the nodes is not too high. Hence, master push updates to every node and consequently the reads can be catered immediately with consistent data.
- *Category 2:* both read and write frequencies are high. In this case even though the high read frequencies demand consistent data, higher write frequencies incur a higher overhead if pushed updates to everybody jeopardizing the throughput. Hence, the master checks for the time difference of consecutive updates and if it is larger than a threshold which is a configurable parameter, then it allows to propagate updates to all nodes otherwise data is propagated only to the deputy nodes.
- *Category 3:* the read frequency is low and update frequency is high. In this case, they use a weaker consistency route which is forwarding the updates only to the deputy nodes. This decision is justified because frequent updates to all nodes tends to utilize the bandwidth fully which is indeed not that useful because the number of read operations are lower. Hence, use a weaker consistency and pull the latest updates from the nearest deputy node if child node needs to cater a read operation.
- *Category 4:* both read and write frequencies are low. In this case they use a kind of combination of both Category 2 and 3. Hence, updates will only be propagated to the deputy nodes and child nodes would check the time duration which it fetched the last update from the deputy nodes before cater a read operation. If the duration exceeds a certain threshold which is a configurable parameter it will fetch the latest from a deputy node otherwise it will use the data already fetched previously.

Using simulation, the authors have shown that their solution helps to significantly reduce the number of update operations pushed to the nodes to provide strong consistency but still can provide consistent data to satisfy the application requirement. Moreover, they have shown that the proposed solution can provide higher consistency level than eventual consistency paying slightly increased transaction cost.

Even though this simulation proves the concept, this model itself has some disadvantages in scalability point of view because the system has a centralized component. Furthermore, the number of 3 deputy nodes may be a bottleneck for a cluster of thousands of child nodes distributed across the globe. Intuitively, updating thousands of nodes by a single master to satisfy Category 1 could be a bottleneck in a real system. But their model and the view of application based adaptation have some good insight.

2.2.2 Harmony

Harmony (Chihoub, Ibrahim, Antoniu, and Perez 2012) is a recent attempt towards automated self-adaptive consistency in cloud storage systems. Harmony can be considered as an extension of the principle introduced in the previous section. Harmony has been integrated to Apache Cassandra. The eventual consistency approach has been adopted by vendors such as Amazon and Facebook because it allows to combine high availability with high throughput. As captured by the CAP theorem (Brewer 2012), in order to provide partition tolerance and high availability of their services, these systems sacrifice consistency. Interestingly, it has been estimated that under heavy read and write workloads, systems those use eventual consistency can return up to 66.61% of stale reads (Wada, Fekete, Zhao, Lee, and Liu 2011), which implies that two out of three reads provide inconsistent results. Hence it is important to have a system which is eventually consistent by default but that can provide stronger consistency when necessary. In fact, several of the underlying storage systems have the required mechanisms to provide stronger levels of consistency, as explained in Section 2.1.2. However, the required level of consistency is highly dependant on the application's properties; hence those settings needs to be updated properly, manually by a system administrator analysing the current system status.

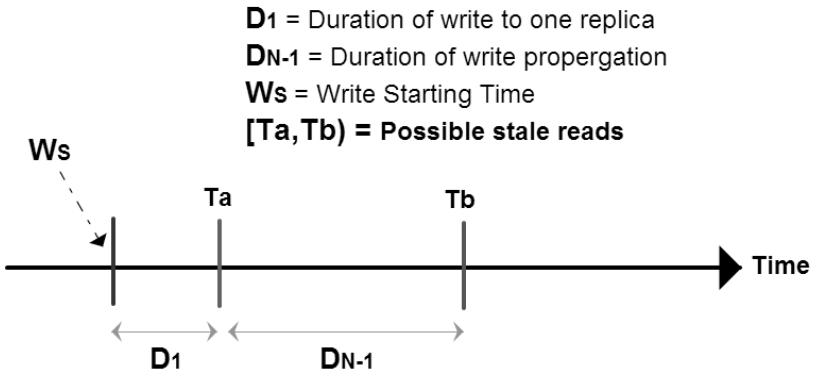

Figure 2.6: Stale read scenario

Harmony is a system to automate this tedious process of manual configuration of consistency levels, by monitoring the system status in real time and automatically changing the configuration accordingly. To this end, Harmony considers different factors, including the data access pattern, the network latency, and the consistency requirements of the application, in particular if the application can tolerate stale reads or not.

A stale read occurs when the read operation takes place after a write operation but the old value is still returned. Consider that a write operation takes place at time T_a. Since the default system operation uses eventually consistency, the update can take up to T_p time to be propagated to all replicas (completes propagation at time T_b). Therefore, stale reads can happen in the interval $[T_a, T_b)$. Figure 2.6 depicts this idea more clearly. Harmony is based on the observation that the application can specify a threshold for the rate of stale reads that can be tolerated and that the system quorum can be tuned to ensure that this threshold is not violated with high probability. The threshold value depends on the semantics of the application. For instance if the stale read rate is 0 it means the application needs strong consistency, on the other hand if it is 100% then application can operate under weak consistency. The probability that an application performs a stale read is derived considering the network delays and assuming that the arrival of requests follows a Poisson distribution, an assumption that is made in several works in the field (Wada, Fekete, Zhao, Lee, and Liu 2011; Tai and Meyer 1996). Figure 2.7 shows how Harmony has been integrated in the current Apache Cassandra implementation. Harmony, as an external component probes the Cassandra cluster to gather statistics and to decide the consistency level automatically and this new consistency level is instantly communicated to the Cassandra driver component used by the application.

Their prototype has shown 80% reduction in stale reads by only adding marginal latency to the responses. Interestingly, their system has yielded 45% throughput improvement while maintaining the appropriate consistency requirement of the application when compared to the strong consistency model in Cassandra.

2.2.3 Consistency Rationing in the Cloud

Consistency Rationing (Kraska, Hentschel, Alonso, and Kossmann 2009) is a system that classifies data into a small set of categories, according to their requirements, and then provides different consistency guaranties to each of these categories. For instance, consider a web-shop where different kinds of data such as customer profile information, credit card information, data about the sold product, user

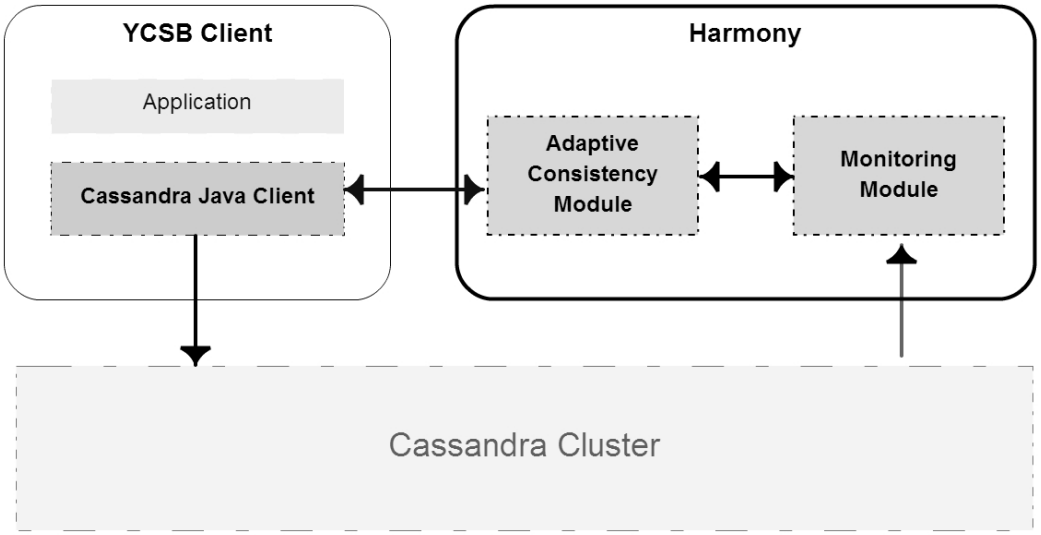

Figure 2.7: Harmony Architecture

preferences (like uses who purchased this item also purchased that item etc), and logging information. In this case the account information (credit card information) should be accessed or manipulated using strong consistency because the inaccuracies could end up with a loss of a revenue. Product inventory information can be handled with a weaker consistency if there is enough stock available but if few items left it needs to have strong consistency in order to prevent overselling of items (because overselling would be a high penalty for the merchant if they sell air tickets or tickets for popular games or shows). In contrast, user preference information and logging information may not affect the system if it remains stale for some period of time, hence can be replicated using eventual consistency. To cope with this type of applications, the following categories are considered:

- *Category A* represents strong consistency where the underlying protocols needs more interactions with the additional services such as lock services, queue services etc, which intuitively increase the cost of the operation and lower response time. Hence this consistency level should be used only when an up to date view is a must.

- *Category C* represents the eventual consistency where the data can be propagated with a delay. This does not need to interact with locking services and additional costly services to ensure serializability. Hence the cost of an operation is low instead the application should be able to cope with inconsistent data.

- *Category B* represents the smarter movement of consistency level between Category A and Cate-

gory C based on the cost to be paid due to an inconsistency. Based on the use case of the web shop in practice the cost of the penalty incurred due to an inconsistency can be quantified. For instance, refunds to a customer for wrong delivered item, reschedule an air ticket, give discounts or other benefits. Hence, they come up with a cost model that can help the system to automatically switch between consistency levels based on current system state.

The authors propose several adaptive policies in order to change the consistency level dynamically:

- *General policy* is based on the conflict probability. Assuming a poisson distribution (Wada, Fekete, Zhao, Lee, and Liu 2011; Tai and Meyer 1996) for the data access distribution of the cloud storage the probability that conflicting access can occur is computed. Initially, the storage system works with eventual consistency and once it detects the conflicting access probability exceeds a certain threshold, strong consistency needs to be applied over that data.
- *Time policy* is straightforward and it requires to input a specific date-time information to the system so that before this specified time system works as eventual consistent (Category C) and as soon as this time passed system switches its consistency to strong consistency (Category A). Hence, this is suitable for auction web sites because, when it is closer to the deadline strong consistency needs to be enforced in order to eliminate the inconsistencies of bidding.
- *Fixed threshold policy* is also straightforward which is simply a configuration that administrators can set when the system would switch consistency. At the beginning, eventual consistency is applied for all the data and once it noticed the monitoring parameter indicated a value below the threshold, system switches to strong consistency. Hence this kind of setting can be used effectively for inventory systems where overselling needs to be eliminated when low stock is available.
- *Demarcation policy* is an extension of the fixed threshold policy. Here in this case, without keeping a global configuration for the threshold each replica of storage can take the decision of when to enforce consistency switching based on the local proportion allocated to that specific replica out of the whole amount contains in the monitoring parameter. For instance, the total inventory amount is divided among the storage nodes and based on their local deduction of the assigned proportion due to the ordering of items they can request more inventory from available replicas before change the consistency. Hence, the system can work with less expensive eventual consistency for a extended period of time before switching to the strong consistency which intuitively makes the system more cost effective.

The authors have implemented their proposed solution on top of Amazon S3 and the experimental evaluation confirms that having probabilistic model to decide when to switch consistency levels has paid off rather than having a static policy. Furthermore, in order to get the statistics required for the decision making they use a sliding window technique so that the total amount of data needs to be accumulated has claimed to be small and not a bottleneck for the system resource utilization.

2.2.4 Global Q-OPT

My work is an evolution of an early prototype that aimed at adapting, in an automated manner, the quorum-system used by Openstack Swift. This prototype has been developed by Maria Couceiro, Matti Hiltunen, Paolo Romano, and Luís Rodrigues. Contrary to the work described in this thesis, this early prototype applied the same quorum-system to all objects of the key-value store. For this reason, we name this preceding version *Global Q-OPT*. In the next paragraphs we describe the main features of *Global Q-OPT*.

It is clear that based on the workload characteristics the correct quorum configuration would help to improve the throughput of a storage system. Identifying that correct quorum configuration is not obvious because of the highly dynamic nature of the data accesses in a particular system.

Global Q-Opt is a prototype developed as an extension to Openstack Swift where it tries to find the workload characteristics by gathering information and to infer the best quorum configuration, using machine learning techniques, for the whole system that would gain a higher throughput than the current throughput provides in the current configuration. Furthermore, Global Q-Opt focuses on providing strong consistency over the data it manages. This algorithm works in rounds which is described as below.

2.2.4.1 Optimization Algorithm

- At the beginning the quorum configurations are defined by the system administrator manually and any configuration that satisfies the $R + W > N$ is acceptable in order to provide strong consistency where R is the read quorum, W is the write quorum and N is the number of replicas or the replication degree.
- Once the system is started, a centralized component called Autonomic Manager (AM) initiates a information gathering phase where it asks all the proxy nodes to gather the information for a certain period of time and send them to AM. Total number of requested reads and writes, average time

duration for a read and a write, total number of successful read write operations are the statistics gathered by all the proxies.

- Once all the statistics are received from all the proxies, AM queries a black-box machine learning module which has already been trained with different workload characteristics to provide the best possible quorum configuration for the current scenario.
- If the proposed quorum configuration is different from the current settings, AM initiates a reconfiguration protocol which ensures non blocking installation of the new quorum configuration.
- Once the new quorum configuration is established, AM checks the performance of the past r rounds and see whether the performance gain is exceeding a threshold predefined by the user. If so, it loops the same steps, otherwise it will terminate the optimization rounds.

Another feature of Global Q-Opt is that, it periodically monitor the system workload and is possible to identify when there is a significant change in the system's workload characteristics. For instance, initially the system had read intensive workload and Global Q-Opt optimized the quorum configuration that best suits the read intensive workload characteristics. After some time the system changes its workload characteristics to write intensive. This change is detected by Global Q-Opt and initiate the above algorithm again in order to optimize the system to suit the current scenario.

2.2.4.2 Reconfiguration Protocol

The reconfiguration protocol of Global Q-Opt works as follows:

- Reconfiguration Manager (RM) sends an initial message to all the proxy nodes with the new quorum configuration.
- Upon receiving the initial message from RM all the proxy nodes apply a transition quorum where transition write quorum is the max value out of current write quorum and proposed write quorum, transition read quorum is the max value out of current read quorum and proposed read quorum.

This transition quorum is used to ensure that the concurrent write operations intersect with the read operations and if there are no concurrent write operations the previous write operation of an object performed with the old quorum also intersect with the current read operation.

Global Q-Opt does not change the current implementation of the Swift read/write implementation described in section 2.1.3.3. In fact it uses the inherent behaviour of writes to its own advantage. At the time a proxy enforces the transition quorum it knows what are the write requests spawned already using the previous quorum configuration. Therefore, when the proposed write quorum is larger than the current active write quorum (Because proposed read quorum size is reduced intuitively in this case so it is possible to have inconsistent reads unless wait for the completion of previous writes), it uses the transition quorum to handle current requests and waits until all the previous write requests return such that it can guarantee that the writes with the previous quorum is now written to all the replicas such that consistency would not compromise. Once all the pending writes are completed, proxy sends a message back to the RM notifying that it is ready to install the new configuration.

- Once all the proxies acknowledged, RM broadcasts a confirmation message to enforce the new quorum configuration.
- Upon receiving the confirmation message from the RM, proxy install the new quorum configuration and acknowledge the RM.

2.2.4.3 Limitations of Global Q-Opt

Global Q-Opt shows a significant through gain for read intensive workloads. Because it does not change the inherent read implementation of the system which queries the replicas serially, throughput increases drastically when reduce the read quorum size. On the other hand, change in the write quorum size shows a very small influence because every request send the write operation to all the replicas in parallel, in background. Hence, it tends to reply almost at the same time which shows a marginal throughput difference with different write quorum combinations.

Moreover, Global Q-Opt is suboptimal for mix workload situations. For instance if a certain application access a set of data in a read intensive manner and meanwhile another application accesses a separate set of data in a write intensive manner Global Q-Opt would hurt one of the applications throughput.

2.3 Top-k Analysis

Space efficient filtering of top-k items from the data accesses to the cloud storage is a vital complementary module essential to implement our proposed system. Hence, we searched for the available

solutions and found space saving top-k algorithm which is explained below.

Because our intention is to perform fine grained optimization for the individual objects in a cloud storage system there should be a space efficient way to filter out the hot objects in a data stream. Hence, the solution proposed by Metwally, Agrawal, and El Abbadi (2005) is quite useful to our work.

In their work, the authors use an approximate integrated approach for finding the most popular items in a data stream. It means that when the access distribution starts to show significant skewness for some data items in a data stream those items become much more accurately picked by the algorithm. Intuitively, this algorithm is effective to identify the popular objects represent in the head of the skewed access distribution and when move towards the tail of the distribution the error rate starts to increase,

The user can specify the number of elements (m) need to be tracked by the algorithm. Then, as soon as it noticed an item it add it to a counter list until it contains m number of items. When the same item is noticed the relevant counter is incremented. If it noticed a new element after the counter list is filled, the least frequent item is replaced by the new item and assume that new item also has see number of times similar to the least frequent item. Hence, it maintains another parameter which specify the possible error which in this case is the frequency of the removed item. For instance, Figure 2.8 depicts how the algorithm works in a situation where m=2 and below is the explanation of each scenario,

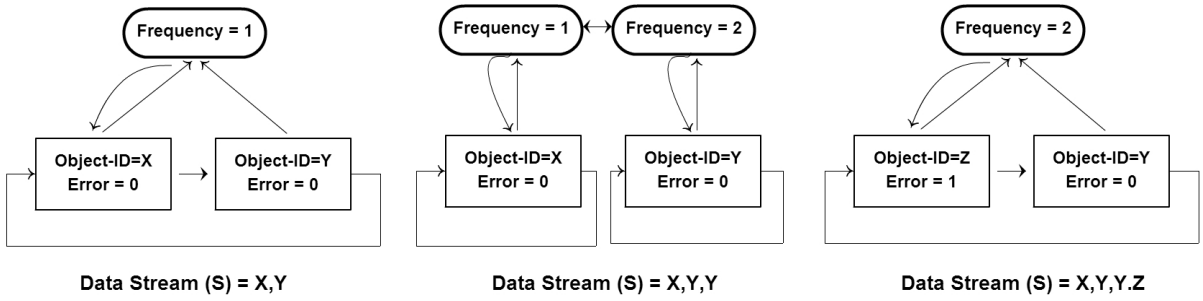

Figure 2.8: Updates to stream summary data structure

- scenario (a) represents the arrival of items X and Y and it has been added to the counter list and each item was seen only once.
- In scenario (b) item Y is seen again and this simply increments the counter of Y to 2 because it already seen and stay in the counter list.
- In scenario (c) a new item Z arrives and hence X is the least frequently seen item in the list it is

replace by Z and at the same time error is also incremented to 1 which is the frequency of the item X.

In addition to this, they proposed a special data structure named “Stream-Summary” to keep the frequencies, object-ID, error count etc which is ordered based on the popularity of the items. Hence, when an item become increasingly popular it can be noticed moving to the top of the data structure and it always returns an ordered list of items based on the popularity with the access frequency and possible error count.

2.4 Machine Learning

Since we suggest to integrate a Machine Learning (ML) based predictor to our system based on initial revelation which showed lack of linear correlation between best write quorum configuration and percentage of write operations as explained in section 1.1, it is important to understand suitable ML techniques and appropriate learning methods related to our work.

ML has many different facets of explanations. In our work, the target of ML is to infer the accurate write quorum configuration fits to the current workload characteristics of the cloud storage provided that a sufficient amount of previous known quorum/workload tuples available. Hence considering it as a technology which can infer a new data point accurately with a high probability based on the observed data up to now is best fit to our work and is of our interest.

Several learning techniques such as supervised, unsupervised etc are used to train different ML modules but we are interested in supervised learning. Supervised learning provides the ML module with a set of known data which is called the training set. Each data item represents an array of metadata called features which explains the scenario the system deals with along with best value that ML should infer if the same training set item is seen in the future. Since, we present the features-set/decision tuple to the learning system, it simply becomes a classification problem. Furthermore, there are algorithms already developed to classify data using decision trees for the provided data set so that for an arbitrary query with all the metadata, ML can suggest the best fit with high probability.

Decision tree is a simple, human readable, flow chart like data structure that can be used to classify data in such a way that a node represents a test on an attribute (i.e., tossing a coin can end up either head or tail), a branch which represents the results of the test leads to another similar node, which contains

another test on an attribute, or to a leaf node, which contains the final decision. This structure helps to form rules to classify different possible scenarios considering the path from root to leaf.

ID3 algorithm (Quinlan 1975) and later successor of it, C4.5 algorithm (Quinlan 1993), both well known to make the decision trees to classify the training sets. But the latest improved version C5.0 algorithm (Quinlan 2012) has many interesting features such as significant speed up, memory efficient, improved accuracy, smaller decision trees etc all in comparison to C4.5 algorithm. Hence, in our implementation we use the C5.0 algorithms as the ML module.

Summary

Due to the positive trend of using cloud storage as the back end for many services, it is a vital field of research to explore the possibilities of optimizing the throughput satisfying the consistency demands of the applications. Here in this chapter we discussed the most popular cloud storage systems widely used in the industry and how they have integrated with cutting edge features to improve user experience. Furthermore, we discussed several proposed solutions and prototypes already developed in order to automate the consistency levels capturing various parameters at run time those can be used as the base for our contribution for this area of research.

The next chapter will introduce the architecture and implementation details of our system 'Q-Opt'.

3 Q-Opt

3.1 Q-OPT: System Model & Overview

Q-OPT is designed to operate with a SDS that adheres to the architectural organization discussed in Section 2.1.3.2, namely: its external interface is represented by a set of proxy agents, and its data is distributed over a set of storage nodes. We denote the set of proxies by $\Pi = \{p_1, \ldots, p_P\}$, and the set of storage nodes by $\Sigma = \{s_1, \ldots, s_S\}$. In fact proxies and storage nodes are logical process which may be in practice mapped to a set of physical nodes using different strategies (e.g., proxies and storage nodes may reside in two disjoint sets of nodes, or each storage node may host a proxy). We assume that nodes may crash according to the fail-stop (non-byzantine) model. Furthermore, we assume that the system is asynchronous, but augmented with unreliable failure detectors which encapsulate all the synchrony assumptions of the system. Communication channels are assumed to be reliable, therefore each message is eventually delivered unless either the sender or the receiver crashes during the transmission. We also assume that communication channels are FIFO ordered, that is if a process p sends messages m_1 and m_2 to a process q then q cannot receive m_2 before m_1.

As illustrated in Figure 3.1, Q-OPT is composed of three main components: the Autonomic Manager, the Reconfiguration Manager, and the Oracle, that we briefly introduce below.

The *Autonomic Manager* is responsible for orchestrating the self-tuning process of the system. To this end, it gathers workload information from the proxy processes, and triggers reconfigurations of the quorum system. The optimization logic of the Autonomic Manager is aimed at maximizing/minimizing a target KPI (like throughput or latency), while keeping into account user defined constraints on the minimum/maximum sizes of the read and write quorums. This allows, for instance, for accommodating fault-tolerance requirements that impose that each write operation to contact at least $k > 1$ replicas. The Autonomic Manager is prepared to collect statistics for different data items and to assign different quorums to different objects. However, for scalability issues, we avoid to collect individual statistics for all objects. In fact, Q-OPT makes a fine grain optimization for the objects mostly accessed, and that

consume the largest fraction of the system resources and then treats in bulk all objects that are in the tail of the access distribution.

More precisely, Q-OPT starts to perform multiple rounds of fine-grain optimization, these work as follows. In each round, a top-k analysis is performed to i) identify the (next) top-k objects that have not been optimized yet, and ii) monitors the access to the top-k objects (identified in the previous round) and extract their read-write profile. Then, the read/write quorum for those objects is optimized (see more detail below). At the end of each round, the changes are applied and the effect of these, on the system performance, is checked. If the average performance improvement over the last γ rounds is above a predefined threshold θ, a new round of fine-grain optimization is performed (for the next top-k objects). When the improvement achieved with the fine-grain optimization is below the threshold, the fine-grain optimization of the system is considered to be terminated. At this point, the system uses the average information collected for the remaining objects (which are on the tail of the access frequency distribution) and a read/write quorum is selected for all those objects based on their aggregated profile.

For optimizing the access to a given data item (or to an aggregated set of data items), the Autonomic Manager relies on an *Oracle* that encapsulates a black-box machine-learning based predictor that is responsible for determining the best quorum, given the workload of the monitored SDS application.

Finally, the *Reconfiguration Manager* is in charge of coordinating a non-blocking reconfiguration protocol that aims at altering the read/write quorum size used by the proxies for a given data item. The reconfiguration is non-blocking in the sense that it does not require halting the processing of read/write operations during the transition phase from the old to the new configuration, even in presence of faults affecting some proxies and/or the Reconfiguration Manager. The reconfiguration protocol is oblivious to the specific quorum-based protocol used to replicate data in the SDS (which may, e.g., provide the semantics of regular or atomic register (Guerraoui and Rodrigues 2006)), but it guarantees that at any time (i.e., before, during, and after the reconfiguration process) the quorum used by a read operation intersects with the write quorum of concurrent write operations, or, in absence of concurrent writes, of the last completed write operation.

A detailed description of each of these three components is provided in the following sections. Note that, for convenience of conciseness, in the remaining of the thesis we describe the three components above as if they have centralized, non fault-tolerant, implementations. However, standard replication techniques, such as state-machine replication (Lamport 1978; Schneider 1990; Budhiraja, Marzullo, Schneider, and Toueg 1993; Wiesmann, Pedone, Schiper, Kemme, and Alonso 2000; Guerraoui and

Rodrigues 2006), can be used to derive fault-tolerant implementations of any of these components, such that they not become single points of failure in the system.

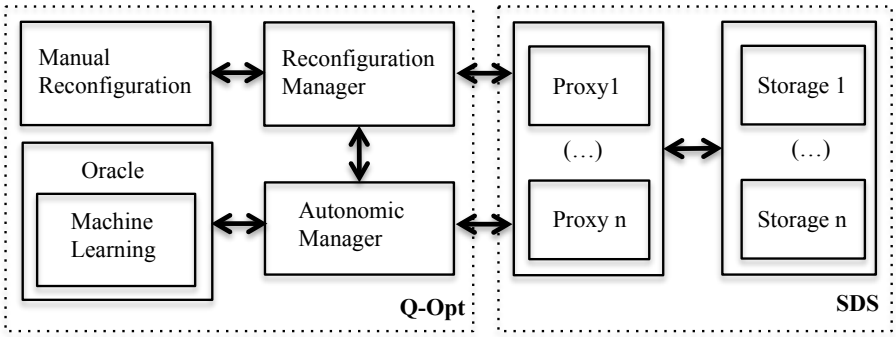

Figure 3.1: Architectural overview.

3.2 Quorum Optimization

There are three key components involved in the quorums optimization process: (i) the proxies (ii) the autonomic manager, and (iii) the oracle.

The Autonomic Manager is responsible for determining when to trigger a quorum reconfiguration. The pseudo code executed at the Autonomic Manager side is depicted in Algorithm 1. It executes the following tasks:

It first starts a new round by broadcasting the new round identifier, r, to all the proxies (line 5). Each proxy p_i then replies with (line 7):

- $topK_i^r$: A set of new "hotspots" objects that, according to the proxy's local accesses, should be optimized in the next round to obtain larger benefits. In order to be able to identify the "hotspots" on each proxy with low overhead, Q-OPT adopts a state of the art stream analysis algorithm (Metwally, Agrawal, and El Abbadi 2005) that permits to track the top-k most frequent items of a stream in an approximate, but very efficient manner.
- $statsTopK_i^{r-1}$: The ratio of write accesses and the size for each of the objects resulting in the top-k analysis of the previous round.
- $statsTailK_i^{r-1}$: Aggregate workload characteristics (see Section 3.8 for detailed information on the traced workload characteristics) for the objects whose quorum size has not been individually optimized, i.e., the objects in the tail of the access distribution.

1 int r=0; *// Round identifier*
2 *// Fine-grain round-based optimization.*
3 **do**
4 r=r+1;
5 **broadcast** [**NEWROUND**, r] **to** Π;
6 $\forall p_i \in \Pi$:
7 **wait received** [**ROUNDSTATS**, r, topK_i^r, statsTopK_i^{r-1}, statsTail_i^{r-1}, th_i^{r-1}] **from** $p_i \vee suspect(p_i)$;
8 statsTopK^{r-1}=*merge*(statsTopK_1^{r-1}, .., statsTopK_P^{r-1});
9 topK^r=*merge*(topK_1^r,...,topK_P^r);
10 **send** [**NEWSTATS**, r, statsTopK^{r-1}] **to** ORACLE;
11 **wait received** [**NEWQUORUMS**, r, quorumsTopK^{r-1}] **from** ORACLE;
12 **send** [**FINEREC**, r, $\langle \text{topK}^{r-1}, \text{quorumsTopK}^{r-1} \rangle$] **to** RM;
13 **wait received** [**ACKREC**, r] **from** RM;
14 **broadcast** [**NEWTOPK**, r, topK^r] **to** Π;
15 th^{r-1}=*aggregateThroughput*(th_1^{r-1},..., th_P^{r-1}) ;
16 $\Delta_{th}(\gamma)$ = *throughput increase over last* γ *rounds.*;
17 **while** $\Delta_{th}(\gamma) \geq \theta$
18 *// Tail optimization.*
19 statsTail^{r-1}=*merge*(statsTail_1^{r-1}, ..,statsTail_P^{r-1});
20 **send** [**TAILSTATS**, statsTail^{r-1}] **to** ORACLE;
21 **wait received** [**TAILQUORUM**, quorumTail^{r-1}] **from** ORACLE;
22 **send** [**COARSEREC**, quorumTail^{r-1}] **to** RM;
23 **wait received** [**ACKREC**, r] **from** RM;

Algorithm 1: Autonomic Manager pseudo-code.

- th_i: The throughput achieved by the proxy during the last round.

Once the information sent by the proxies is gathered and merged (line 8 and 9), the merged statistics of previous round top-k is fed as *input features* to the Oracle (line 10), that outputs a *prediction* (line 11) of the right quorum to use *for each* object in the top-k set (see details in Section 3.8). In the current prototype, the Oracle only outputs the size W of the write quorum and the size R of the read quorum is derived automatically based on the system's replication degree, i.e., $R = N - W + 1$. If the output of the Oracle is different from the current quorum system for that object, a reconfiguration is triggered. In this case, the Autonomic Manager interacts with the Reconfiguration Manager (lines 12 and 13), which is in charge of orchestrating the coordination among proxy and storage nodes and adapt the current quorum configuration for the top-k objects identified in the previous round. Otherwise, if the current configuration is still valid, no reconfiguration is triggered. As a final step of a fine-grain optimization round, the Autonomic Manager broadcast the current top-k set to the proxies. Thus, each proxy can start monitoring the objects that belong to the current top-k set in the next round.

At the end of each round, the Autonomic Manager, based on the average throughput improvements achieved during the last γ rounds, decides whether to keep optimizing individual objects in a fine-grain

manner or to stop. When the gains obtained with the fine-grain optimization of individual "hotspot" objects becomes negligible (i.e., lower than a tunable threshold θ), a final optimization step to tune the quorum configurations used to access the the remaining objects, i.e., the objects that fall in the tail of the access distribution. These objects are treated as bulk (lines 19 - 22): the same read/write quorum is assigned to all the objects in the tail of the access distribution based on its aggregate workload characterization.

The frequency with which the Autonomic Manager cycle is executed is regulated by a classical trade-off for any autonomic system: the more often the Autonomic Manager queries the machine learning model, the faster it reacts to workload changes. However, it also increases the risk to trigger unnecessary configuration changes upon the occurrence of momentary spikes that do not reflect a sustained change in the workload. In our current prototype we use a simple approach based on a moving average over a window time of 30 seconds, which has proven successful with all the workloads we experimented with. As with any other autonomic system, in our implementation there is also a trade-off between how fast one reacts to changes and the stability of the resulting system. In the current prototype, we simply use a fixed "quarantine" period after each reconfiguration, to ensure that the results of the previous adaptation stabilise before new adaptations are evaluated. Of course, the system may be made more robust by introducing techniques to filter out outliers (Hodge and Austin 2004), detect statistically relevant shifts of system's metrics (Page 1954), or predict future workload trends (Kalman et al. 1960).

3.3 Reconfiguration Manager

The Reconfiguration Manager (subsequently denoted RM) executes the required coordination among proxy and server nodes in order to allow them to alter the sizes of read and write quorums without endangering neither, consistency, nor availability during reconfigurations. This coordination enforced by Q-OPT is designed to preserve the following property, which is at the basis of all quorum systems that provide strong consistency:

Dynamic Quorum Consistency. *The quorum used by a read operation intersects with the write quorum of any concurrent write operation, and, if no concurrent write operation exists, with the quorum used by the last completed write operation.*

where two operations o_1, o_2 are concurrent if at the time in which a proxy starts processing o_2, the processing of o_1 by a (possibly different) proxy has not been finalized yet (or vice-versa). The availability

of a SDS system, on the other hand, is preserved by ensuring that read/write operations can be executed in a non-blocking fashion during the reconfiguration phase, even despite the crash of (a subset of) proxy and storage nodes.

As mentioned the RM supports both per-object as well as global (valid across all objects in the SDS) changes of the quorum configurations. To simplify presentation, we first introduce the protocol for the simpler scenario of global quorum reconfigurations. We discuss how the reconfiguration protocol can be extended to support per-object granularity in Section 3.7.

3.4 Algorithm overview

There are three different type of components involved in the execution of the reconfiguration algorithm: the storage nodes, the proxy nodes, and the RM. The purpose of the reconfiguration algorithm is to change the quorum configuration, i.e., the size of the read write quorums, used by the proxy servers.

The algorithm is coordinated by the RM. When the RM runs the configuration algorithm we say that the RM *installs* a new quorum system. We denote the quorum system being used when the reconfiguration is started as the *old quorum* and the quorum system that is installed when the reconfiguration is concluded the *new quorum*. Old and new write and read quorums are denoted, respectively, as *oldW*, *oldR*, *newW*, and *newR*. Each quorum is associated with an *epoch number*, a sequentially serial number that is incremented by the RM when some proxy is suspected to have crashed during a reconfiguration. We also assume that storage nodes maintain a variable, called *currentEpoch*, which stores the epoch number of the last quorum that has been installed by the RM. As it will be explained below, during the execution of the reconfiguration algorithm, proxy nodes use a special *transition quorum*, that is sized to guarantee intersection with both the old and new quorums.

We assume a fail stop-model (no recovery) for proxies and storage nodes, and that at least one proxy is correct. As for the storage nodes, in order to ensure the termination of read and write operations in the new and old quorum configuration, it is necessary to assume that the number of correct replicas is at least *max(oldR, oldW, newR, newW)*. For ease of presentation, we assume that the sets Σ and Π are static, i.e. nodes are not added to these sets, nor are they removed even after a crash. In order to cope with dynamic groups, one may use group membership techniques, e.g. (Birman and Renesse 1994), which are orthogonal to this work.

```
1  int epNo=0; // Epoch identifier
2  int cfNo=0; // Configuration round identifier
3  int curR=1, curW=N; // Sizes of the read and write quorums
4  // Any initialization value s.t. curR+curW>N is acceptable.
5  changeConfiguration(int newR, int newW)
6      wait canReconfig;
7      canReconfig = FALSE;
8      cfNo++;
9      broadcast [NEWQ, epNo, cfNo, newR, newW ] to Π;
10     ∀p_i ∈ Π :
11       wait received [ACKNEWQ, epNo] from p_i ∨ suspect(p_i);
12     if ∃p_i : suspect(p_i) then
13         tranR=max(curR,newR); tranW=max(curW,newW);
14         epochChange(max(curR,curW),tranR,tranW);
15     broadcast [CONFIRM, epNo, newR, newW ] to Π;
16     ∀p_i ∈ Π :
17       wait received [ACKCONFIRM, epNo] from p_i ∨ suspect(p_i);
18     if ∃p_i : suspect(p_i) then
19         epochChange(max(newR,newW),newR,newW);
20     curR=newR; curW=newW;
21     canReconfig = TRUE;
22 epochChange(int epochQ, int newR, int newW)
23     epNo++;
24     broadcast [NEWEP,epNo,cfNo,newR, newW ] to Σ;
25     wait received [ACKNEWEP, epNo] from epochQ s_i ∈ Σ;
```

Algorithm 2: Reconfiguration Manager pseudo-code.

We assume that the RM never fails — as mentioned before, standard techniques may be used to derive a fault-tolerant implementation of the RM — and that it is equipped with an eventually perfect failure detection service (Chandra and Toueg 1996) that provides, possibly erroneous, indications on whether any of the proxy nodes has crashed. An eventually perfect failure detector ensures *strong completeness*, i.e., all faulty proxy processes are eventually suspected, and *eventual strong accuracy*, i.e., there is a time after which no correct proxy process is ever suspected by the RM. The reconfiguration algorithm is indulgent (Guerraoui 2000), in the sense that in presence of false failure suspicions only the liveness of read/write operations can be endangered (as we will see they may be forced to re-execute), but neither the safety of the quorum system (i.e., the Dynamic Quorum Consistency property), nor the termination of the reconfiguration phase can be compromised by the occurrence of false failure suspicions. The failure detection service is encapsulated in the *suspect* primitive, which takes as input a process identifier $p_i \in \Pi$ and returns true or false depending on whether p_i is suspected to have crashed or not. Note that proxy servers are not required to detect the failure of storage servers nor vice-versa.

In absence of faults, the RM executes a two-phase reconfiguration protocol with the proxy servers, which can be roughly summarized as follows. In the first phase, the RM informs all proxies that a reconfiguration must be executed and instructs them to i) start using the transition quorum instead of

the old quorum, and ii) wait till all the pending operations issued using the old quorum have completed. When all proxies reply, the RM starts the second phase, in which it informs all proxies that it is safe to start using the new quorum configuration.

This mechanism guarantees that the quorums used by read/write operations issued concurrently to the quorum reconfiguration intersect. However, one needs to address also the scenario in which a read operation is issued on an object that was last written in one of the previous quorum configurations, i.e., before the installation of the current quorum. In fact, if an object were to be last written using a write quorum, say *oldW*, smaller than the one used in the current configuration, then the current read quorum may not intersect with *oldW*. Hence, an obsolete version may be returned, violating safety. We detect this scenario by storing along with the object's metadata also a logical timestamp, $cfNo$, that identifies the quorum configuration used when the object was last written. If the version returned using the current read quorum was created in a previous quorum configuration having identifier $cfNo$, the proxy repeats the read using the largest read quorum used in any configuration installed since $cfNo$ (in case such read quorum is larger than the current one).

Since failure detection is not perfect, in order to ensure liveness the two-phase quorum reconfiguration protocol has to advance even if it cannot be guaranteed that all proxies have updated their quorum configuration. To this end, the RM triggers an epoch change on the back-end storage nodes, in order to guarantee that the operations issued by any unresponsive proxy (which may be using an outdated quorum configuration) are preventively discarded to preserve safety.

3.5 Quorum Reconfiguration Algorithm

The pseudo code for the reconfiguration algorithm executed at the Replication Manager side is depicted in Algorithm 2. The reconfiguration can be triggered by either the Autonomic Manager, or by a human system administrator, by invoking the *changeConfiguration* method and passing as arguments the new sizes for the read and write quorums, *newQ* and *WriteQ*. Multiple reconfigurations are executed in sequence: a new reconfiguration is only started by the RM after the previous reconfiguration concludes.

Failure-free scenario. To start a reconfiguration, the RM broadcasts a **NEWQ** message to all proxy nodes. Next, the RM waits till it has received an **ACKNEWQ** message from every proxy that is not suspected to have crashed.

```
1  int lEpNo=0; // Epoch identifier
2  int lCfNo=0; // Configuration round identifier
3  set Q={}; // list of cfNo along with respective read/write quorum sizes
4  int curR=1, curW=N; // Sizes of the read and write quorums
5  // Any initialization value s.t. curR+curW>N is acceptable.
6  upon received [NEWQ, epNo, cfNo, newR, newW ] from RM
7      if lEpNo≤epNo then
8          lEpNo=epNo;
9          lCfNo=cfNo;
10         Q=Q ∪ < cfNo, newR, newW > ;
11         int oldR=curR; int oldW=curW;
12         // new read/writes processed using transition quorum
13         tranR=max(oldR,newR); tranW=max(oldW,newW);
14         wait until all pending reads/writes issued using the old quorum complete;
15         send [ACKNEWQ, epNo ] to RM;
16 upon received [CONFIRM, epNo, newR, newW ] from RM
17     if lEpNo≤epNo then
18         lEpNo=epNo;
19         curR=newR;  curW=newW;
20         send [ACKCONFIRM, epNo ] to RM;
```

Algorithm 3: Proxy pseudo-code (quorum reconfiguration).

Upon receipt of a **NEWQ** message, see Algorithm 3, a proxy changes the quorum configuration used for its future read/write operations by using a *transition quorum*, whose read, respectively write, quorum size is equal to the maximum of the read, respectively write, quorum size in the old and new configurations. This ensures that the transition read (tranR), resp. write (tranW), quorum intersects with the write, resp. read, quorums of both the old and new configurations. Before replying back to the RM with an **ACKNEWQ** message, the proxy waits until any "pending" operations it had issued using the old quorum completed.

If no proxy is suspected to have crashed, a **CONFIRM** message is broadcast to the proxy processes, in order to instruct them to switch to the new quorum configuration. Next, the RM waits for a **ACK-CONFIRM** reply from all the non-suspected proxy nodes. Finally, it flags that the reconfiguration has finished, which allow for accepting new reconfigurations requests.

The pseudo-code for the management of read and write operations at the proxy nodes is shown in Alg. 4 and Alg. 5. As already mentioned, in case a read operation is issued, the proxies need to check whether the version returned using the current read quorum was created by a write that used a write quorum smaller than the one currently in use. To this end, proxies maintain a set Q containing all quorum configurations installed so far[1] by the Autonomic Manager. If the version returned using the current read quorum was created in configuration $cfNo$, the proxy uses set Q to determine the value of

[1] In practice, the set Q can be immediately pruned whenever the maximum read quorum is installed.

```
1  upon received [Read, oId] from client c
2      while true do
3          broadcast [Read, oId, curEpNo] to Σ;
4          wait received [ReadReply, oId, val, ts, W] from Σ' ⊆ Σ s.t. |Σ'|=curR ∨ ([NACK, epNo,newR, newW ]
           ∧epNo > lEpNo);
5          if received [NACK, epNo, cfNo, newR, newW ] then
6              lEpNo=epNo; lCfNo=cfNo; curR=newR; curW=newW;
7              Q=Q ∪ < cfNo, newR, newW > ;
8              continue; // re-transmit in the new epoch
9          v= select the value with the freshest timestamp;
10         // Set of read quorums since v.cfNo till lCfNo;
11         S = {R_i :< q_i, R_i, · >∈Q ∧v.cfNo ≤ q_i ≤ lCfNo};
12         if max(S)≤curR then
13             // safe to use cur. read quorum
14             send [ReadReply, oId, v] to client c;
15         else
16             // compute read quorum when v was created.
17             int oldR=max(S);
18             // obtain a total of oldR replies.
19             wait received [ReadReply, oId, val, ts] from Σ' ⊆ Σ s.t. |Σ'|=oldR ∨ ( [NACK, epNo,newR, newW ]
               ∧epNo > lEpNo);
20             if received [NACK, epNo, cfNo, newR, newW ] then
21                 lEpNo=epNo; lCfNo=cfNo; curR=newR; curW=newW;
22                 Q=Q ∪ < cfNo, newR, newW > ;
23                 continue; // re-transmit in the new epoch
24             v= select the value with the freshest timestamp;
25             send [ReadReply, oId, v] to client c;
26             // write v using the current quorum
27             write(v,oId,v.ts);
28         break;
29     end
```

Algorithm 4: Proxy pseudo-code (read logic).

the largest read quorum used in any configuration since $cfNo$ till the current one. If this read quorum, noted oldR (see line 17 of Alg. 4), is larger than the current one, the read is repeated using oldR. Further, the value is written back using the current (larger) write quorum. Note that re-writing the object is not necessary for correctness. This write can be performed asynchronously, after returning the result to the client, and is meant to spare the cost of using a larger read quorum when serving future reads for the same object.

Coping with failure suspicions. In case the RM suspects some proxy while waiting for an **ACKNEWQ** or an **ACKCONFIRM** message, the RM ensures that any operation running with an obsolete configurations is prevented from completing. To this end, the RM relies on the notion of *epochs*. Epochs are uniquely identified and totally ordered using a scalar timestamp, which is incremented by the RM whenever it suspects the failure of a proxy at lines 11 and 16 of Alg. 2. In this case, after increasing the epoch number, the RM broadcasts the **NEWEP** message to the storage nodes. This message includes 1) the new epoch identifier, and 2) the configuration of the transition quorum or of the new quorum, depending on

```
1  upon received [Write, oId, value] from client c
2      write (val, oId, getTimestamp());
3      send [WriteReply, oId] to client c;
4  write(value v, objId oid, timestamp ts)
5      while true do
6          broadcast [Write, oId, val, ts, curEpNo] to Σ;
7          wait received [WriteReply, oId] from Σ' ⊆ Σ s.t. |Σ'|=curW ∨ ([NACK, epNo,newR, newW ]
           ∧epNo > lEpNo);
8          if received [NACK, epNo, cfNo, newR, newW ] then
9              lEpNo=epNo; lCfNo=cfNo; curR=newR; curW=newW;
10             Q=Q ∪ < cfNo, newR, newW > ;
11             continue; // re-transmit in the new epoch
12         break;
13     end
```

Algorithm 5: Proxy pseudo-code (write logic).

whether the epoch change was triggered at the end of the first or of the second phase.

Next, the RM waits for acknowledgements from an *epoch-change quorum*, whose size is determined in order to guarantee that it intersects with the read and write quorums of any of the configurations in which the proxies may be executing. Specifically, if the epoch change is triggered at the end of the first phase, the size of the epoch-change quorum is set equal to the maximum between the size of the read and write quorums in the old configuration. It is instead set equal to the maximum between the size of the read and write quorums of the new configuration, if the epoch change is triggered at the end of the second phase. As we shall discuss more in detail in Section 3.6, this guarantees that the epoch change quorum intersects with the read and write quorums of any operation issued by nodes that may be lagging behind, and not have updated their quorum configuration yet.

When a storage node (see Alg. 6) receives an **NEWEP** message tagged with an epoch identifier larger than its local epoch timestamp, it updates its local timestamp and *rejects* any future write/read operation tagged with a lower epoch timestamp. It then replies back to the RM with an **ACKNEWEP** message. Whenever an an operation issued by a proxy in an old epoch is rejected, the storage node does not process the operation and replies with a **NACK** message, in which it specifies the current epoch number and the quorum configuration of this epoch.

Upon receiving a **NACK** message (see Algs. 4 and 5), the proxy node is informed of the existence of a newer epoch, along with the associated quorum configuration. Hence, it accordingly updates its local knowledge (i.e., its epoch and read/write quorum sizes), and re-executes the operation using the new epoch number and the updated quorum configuration.

```
1  int lEpNo=0; // Epoch identifier
2  int lCfNo=0; // Configuration round identifier
3  int curR=1, curW=N; //Sizes of the read and write quorums
4  // Any initialization value s.t. curR+curW>N is acceptable.
5  upon received [NEWEP,epNo, cfNo, newR, newW ] from RM
6      if epNo ≥lEpNo then
7          lEpNo=epNo;
8          lCfNo=cfNo;
9          curR=newR; curW=newW;
10         send [ACKNEWEP, epNo] to Reconfiguration Manager;
11 upon received [Read, epNo, ...] or [Write, epNo, ...] from π_i ∈ Π
12     if epNo <lEpNo then
13         send [NACK, epNo, cfNo, newR, newW ] to p_i;
14     else
15         process read/write operation normally;
16         if operation is a write then
17             store lCfNo in the version metadata cfNo;
18         else
19             piggyback cfNo to the ReadReply message;
20         end
21     end
```

Algorithm 6: Storage node pseudo-code.

3.6 Correctness arguments

Safety. We need to show that the quorum used by a read operation intersects with the write quorum of any concurrent write operation, and, if no concurrent write operation exists, that a read quorum intersects with the quorum used by the last completed write operation. We denote with *oldR, oldW*, *newR, newW* and *tranR, tranW* the read and write quorums used, respectively in the initial, new and transition phase.

As already mentioned, since *tranR=max(oldR,newR)* and *tranW =max(oldW,newW)*, the read, resp. write, quorums used during the transition phase intersect necessarily with the write, resp. read, quorums in both the new and old phases. Hence, safety is preserved for operations issued using the transition quorums.

In absence of failure suspicions, if any proxy process starts using the new quorum, the protocol guarantees that there is no pending concurrent operation running with the old quorum. It remains to discuss the case in which a read operation uses the new quorum size and there are no concurrent write operations. If the version returned using the current read quorum, say v, was created in the current quorum configuration, then the last created version is necessarily returned. Let us now analyze the case in which the last write was performed using a previous quorum configuration with timestamp $cfNo$. In this case the proxy repeats the read with the largest read quorum of any configuration installed since $cfNo$. This read quorum is guaranteed to intersect with any write operation issued since the creation of

v. Hence, the read is guaranteed to return the latest value written by any non-concurrent write operation.

If the RM suspects a node, either at the end of first or of the second phase of the reconfiguration protocol, an epoch change is triggered. This ensures that the storage nodes commit to reject the operations issued in the previous epoch. The values of epochQ are chosen large enough to guarantee intersection of the epoch change quorum with the quorum used by any operation of a node that may be lagging behind, and may not have updated his quorum configuration as requested by the RM. This in its turn guarantees that the operation will gather at least a **NACK**, and will be re-executed in the new epoch. At the end of phase 1 epochQ is set to max(*oldR,oldW*), as the proxy may be executing using either the old quorum or the transition quorum. In fact, the transition quorum is guaranteed to intersect with a quorum of size epocQ=max(*oldR,oldW*) by construction; also, a quorum of size epocQ=max(oldR,oldW) necessarily intersects with both oldR and oldW. Since at the end of phase 2 epochQ, the proxies may be using either the new quorum or the transition quorum, the epoch change quorum is set to max(*newR, newW*) for analogous reasons.

Liveness. We start by showing that if a **changeConfiguration** is triggered at the RM, it eventually terminates.

In absence of failure suspicions, the first and second phase of the reconfiguration protocol complete since: i) we assume reliable channels; ii) the wait conditions at lines 11 and 16 of Alg. 2 eventually complete since we assume that the number of correct storage nodes is max(*oldR,oldW,newR,newW*); iii) no other blocking primitives are executed.

By the strong completeness accuracy of failure detection, if a process is faulty, it will be eventually suspected. This ensures that the waits at lines 12 and 17 of Alg. 2 will eventually unblock if some proxy fails. This is true clearly even if some proxy is falsely suspected to have crashed. In either case, an epoch change will be triggered at the end of phase 1 and/or phase 2. The epoch change phase is non-blocking, given our assumption on the number of correct storage nodes.

Let us now show that if a correct proxy node p_i receives a **NEWQ** message with a new quorum configuration *newR, newW*, it eventually installs the new configuration. As discussed above, the corresponding instance of the reconfiguration protocol eventually terminates, possibly triggering an epoch change. If no epoch change is triggered, eventually p_i receives a **CONFIRM** message by the RM and installs the new quorum. If an epoch change is triggered, there are 2 cases: i) p_i can either receive the **CONFIRM** message by the RM, or ii) it receives a **NACK** from a storage node while executing a read or

write operation. In both cases, p_i installs the new quorum (line 19 of Alg. 3, lines 6 and 20 of Alg. 4, and line 10 of Alg. 5).

Finally let us analyze the termination of read and write operations. Since we are assuming the availability of both the old and new quorum configurations, in absence of failure suspicions/epoch changes the read/write quorums are eventually gathered. Also the read/write quorums do not contain any **NACK** message and the read/write operations complete without retrying. In presence of failure suspicions, the reconfiguration protocol can trigger up to two epoch changes that may cause pending operations to be re-executed twice. However, at each re-execution, upon receiving a **NACK** message (lines 4-5 of Alg. 4 and 8-9 of Alg. 5), the proxy learns the new quorum configuration and updates it epoch number. In order to ensure the eventual successful termination of read/write operations, we need therefore to assume either that i) the faulty nodes are eventually removed from the system, which eventually causes the RM not to suspect any proxy, or that ii) the time interval between two subsequent quorum reconfigurations is sufficiently long to allow operations to execute successfully in the most recent epoch, possibly after a finite number of re-executions.

3.7 Per-object quorum reconfiguration

As discussed, the above presented protocol allows for altering the quorum size used by the entire data store. However, extending the above presented protocol to allow for tuning independently the quorum sizes used to access different objects in the data store is relatively straightforward.

In particular, during the initial, round based optimization phase described in Section 3.2, the RM is provided with a set of object identifiers and with their corresponding new quorum configurations. The RM forwards this information to the proxies via the *NewQ* message. The proxy servers, in their turn, shall store the mapping between the specified object identifiers and the corresponding write quorums, and use this information whenever they are serving a read or a write operation. Note that, in our prototype, we store this mapping in main memory, as only a reduced set of "hotspots" is individually optimized. This simple mechanism allows the proxy servers to determine the new and old quorum sizes to use on a per object basis. Also, when a fine-grain quorum reconfiguration is requested that only affects a set of data items D, the proxy should wait only, in the first phase of the reconfiguration algorithm, for the completion of any pending operation (using the old quorum) targeting some of the items in D.

Any (read/write) access to objects whose quorum size has not been individually optimized can use

a common, global quorum configuration and be treated exactly as shown in Section 3.5.

3.8 The Oracle

The Oracle is responsible for determining the best quorum configuration for the current workload conditions. We cast the selection of the optimal quorum configuration as a classification problem (Mitchell 1997), in which one is provided with a set of input metrics (also called features) describing the current state of the system and is required to determine, as output, a value from a discrete domain (i.e., the best performing quorum configuration among a finite set in our case). In Q-OPT we rely on black-box machine learning techniques to automatically infer a predictive model of the optimal quorum configuration.

To this end, Q-OPT exploits the C5.0 algorithm (Quinlan 2012). C5.0 builds a decision-tree classification model in an initial, off-line training phase during which a greedy heuristic is used to partition, at each level of the tree, the training dataset according to the input feature that maximizes information gain. The output model is a decision-tree that classifies the training cases according to a compact (human-readable) rule-set, which can then be used to classify (decide the best quorum strategy) future scenarios.

The accuracy achievable by any machine learning technique is well known to be strictly dependant on the selection of appropriate input features (Mitchell 1997). In Q-OPT we selected a restricted set of workload characterization metrics $\vec{\mathscr{W}}$ that can be measured using lightweight, non-intrusive monitoring mechanisms, and which capture both the proxies' and the storage nodes' current capacity to serve each type of request. The input features include: number of read and write operation requests received, execution time of read and write operations, number of read and write operations that were successfully served, percentage of read operations, average sizes of the read and written objects, number of proxy and storage nodes.

In order to build a training set for the classification model, Q-OPT relies on an off-line training phase during which we test a heterogeneous set of synthetic workloads and measure a reference KPI of the system (e.g., throughput or response time) when using different read/write quorum configurations. The training set is then composed by tuples of the form $\langle curW, \vec{\mathscr{W}}, bestW \rangle$, where $curW$ is the write quorum configuration used while executing workload $\vec{\mathscr{W}}$. $bestW$ is the best performing write quorum configuration for $\vec{\mathscr{W}}$ and the target class for the classification problem.

This model construction methodology allows us to accommodate in a simple way also for application-dependant constraints on the quorum sizes, e.g. imposing a minimum value for the size for the write quorums for fault-tolerance purposes. To this end, it suffices to restrict the tuples in the training set to include only admissible values of both *curW* and *bestW*. More in detail, tuples using illegal values of *curW* are discarded from the training set. Tuples that use legal values of *curW* and whose optimal quorum configuration, *bestW*, violates some application-dependant constraint are retained in the training set, but their *bestW* value is replaced with the best performing among the *feasible* write quorum configurations.

3.9 Integration in Swift

At the point of integration of the proposed system with Openstack-swift, several changes were done to the default code execution path of both read and write operations. As explained in the section 2.1.3.3 the default read implementation is sequential, hence to get a better performance with different read quorum configurations read operation was changed to send the requests in parallel to the quorum of replicas.

To be more specific, we included the python greenlet library which is recommended by the python developers to create concurrent micro threads called 'tasklets' which are lightweight and have more control over the time when each of them executed. Furthermore, instead of changing this implementation fully we made it as a tunable configuration so that switching between the original implementation and this new feature is possible even at runtime.

Similarly, the original implementation of the write operations was also changed and made as a tunable feature. Originally, all the write operations contacted all the replica nodes in parallel as explained in section 2.1.3.3. This was also inefficient when we have a quorum system where it is possible to contact lesser number of replicas and still achieve the consistency while replicator services can handle the background data replication.

In this case, this new implementation initially it spawn concurrent connections only to the quorum of write replicas and if anyone failed to answer, then only the next possible replica is selected and send the write request as a fallback mechanism.

Apart from the above two major control flow changes, small code snippets had to be included in the default execution path which is essential to gather statistics such as read/write request count, dura-

tions, pending write operations etc. And the function where the quorum of the system is checked was redirected to the AM which is the other important component integrated to the swift implementation as a separate module It contains all the logical components those responsible for statistic gathering, quorum map maintenance (quorum map contains the new quorum settings for the hot objects) querying top-k component, querying machine learning component etc. All the featured integrated to this module can be switch on/off at run time or can be fully eliminated by changing the configuration such that original implementation of the swift is unharmed due to any of these changes.

Overall for the swift integration there were about 1500 code lines were written.

Summary

In this chapter we have been through the design and implementation of Q-Opt. Initially we elaborated the architecture of our proposed solution and then the change of control flow of each component such as proxy and storage node. Furthermore, we discussed the reconfiguration algorithms and how we changed the same reconfiguration algorithm in order to acquire both per object optimization and tail optimization of the system's access distribution. Then we explained in detail the correctness of the proposed algorithm theoretically and finally it was detailed that how it was integrated to the Openstack Swift's original implementation.

In the next chapter we present the experimental evaluation performed using this prototype.

4 Evaluation

First in this section we present the results of an experimental study aimed at quantifying the impact on performance of using different read and write quorum sizes in OpenStack Swift, a popular SDS solution. Then we assess three main aspects: the accuracy of the ML-based Oracle; the effectiveness of Q-OPT in automatically tuning the quorum configuration in presence of complex workloads; and the efficiency of the quorum reconfiguration algorithm. Before presenting the results, we briefly describe the experimental platform and methodology that we used.

4.1 Environment Set up

The experimental test-bed used to gather the experimental results presented in this thesis is a private cloud comprising a set of virtual machines (VMs) deployed over a cluster of 20 physical machines. The physical machines are connected via a Gigabit switch and each is equipped with 8 cores, 40 GB of RAM and 2x SATA 15K RPM hard drives (HDs). We allocate 10 VMs for the storage nodes, 5 VMs to serve as proxy nodes, and 5 VMs to emulate clients, i.e., to inject workload. Each client VM is statically associated with a different proxy node and runs 10 threads that generate a closed workload (i.e., a thread injects a new operation only after having received a reply for the previously submitted operation) with zero think time. Each proxy and client VM runs Ubuntu 12.04, and is equipped with 8 (virtual) cores, 10GB disk and 16GB of RAM memory. On the other hand, each storage node VM is equipped with 2 (virtual) cores, 100GB disk and 9GB of RAM memory.

Furthermore, the top-k module which would filter out the popular objects based on the access distribution is collocated in every proxy node, hence each proxy can individually query the top-k module with the statistics of object accesses via that proxy. One of the proxy nodes would work as the master so that the ML module is collocated there. Hence in this set up proxy nodes independently query the top-k module to find the popular objects and then will send those objects' statistics to the master proxy where the ML module is queried (only by the master) in order to find the corresponding quorum configuration.

When we initiate the storage system we set the replication degree to 5 and use the default distribution policy that scatters object replicas randomly across the storage nodes (while enforcing that replicas of the same object are placed on different nodes)

In the first motivating experiment explained in section 4.2 we have used a single tenant and a workload where all objects are accessed with the same profile. In the evaluation of the full system we consider more complex scenarios with skewed non-uniform workloads.

4.2 Impact of the read/write quorum sizes

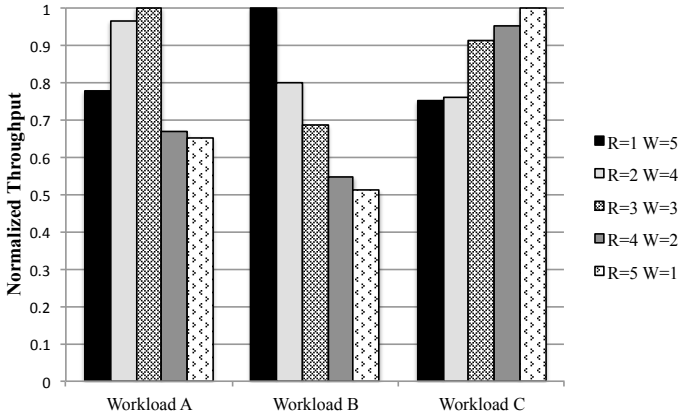

Figure 4.1: Normalized throughput of the studied workloads.

We start by considering 3 different workloads that are representative of different application scenarios. Specifically, we consider two of the workloads specified by the well known YCSB (Cooper, Silberstein, Tam, Ramakrishnan, and Sears 2010) benchmark (noted Workload A and B), which are representative of scenarios in which the SDS is used, respectively, to store the state of users' sessions in a web application, and to add/retrieve tags to a collection of photos. The former has a balanced ratio between read and write operations, the latter has a read-dominated workload in which 95% of the generated operations are read accesses. We also consider a third workload, which is representative of scenario in which the SDS is used as a backup service (noted Workload C). In this case, 99% of the accesses are write operations. Note that such write-intensive workloads are frequent in the context of personal file storage applications, as in these systems a significant fraction of users exhibits an upload-only access pattern (Drago, Mellia, M. Munafo, Sperotto, Sadre, and Pras 2012).

Figure 4.1 shows the throughput of the system (successful operations per second) normalized with respect to the best read/write quorum configuration for each workload. These results were obtained using

one proxy node and 10 clients. The results clearly show that when increasing the size of the predominant operation quorum, the number of served operations decreases: configurations favouring smaller read quorums will achieve a higher throughput in read-dominated workloads, such as Workload B, and vice-versa, configurations favouring smaller write quorums achieve a higher throughput in write-dominated workloads, such as Workload C. Mixed workloads such as Workload A, with 50% read and 50% write operations, perform better with more balanced quorums, favouring slightly reading from less replicas because read operations are faster than write operations (as these need to write to disk).

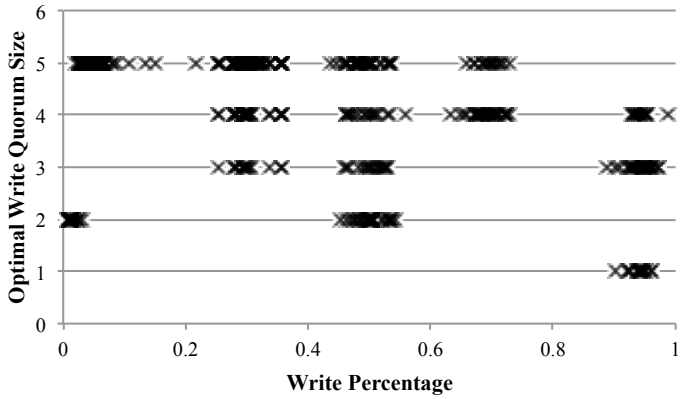

Figure 4.2: Optimal write quorum configuration vs write percentage.

In order to assess to what extent the relation between the percentage of writes in the workload and the optimal write quorum size may be captured by some linear dependency, we tested approx. 170 workloads, obtained by varying the percentage of read/write operations, the average object size, and the number of clients connected to the proxy in the domain. In Figure 4.2 we show a scatter plot contrasting, for each tested workload, the optimal write quorum size and the corresponding write percentage. The experimental data clearly highlights the lack of a clear linear correlation between these two variables. This happens mainly because of the variety of the object size distribution used when initializing the cluster. Smaller object sizes tends to read/write in a shorter time span and give higher throughput but if the object sizes are larger the results would be the opposit. Therefore, mix object sizes would make the final throughput not always predictable. This has motivated our choice of employing black-box modelling techniques (i.e., decision trees) capable of inferring more complex, non-linear dependencies between the characteristics of a workload and its optimal quorum configuration.

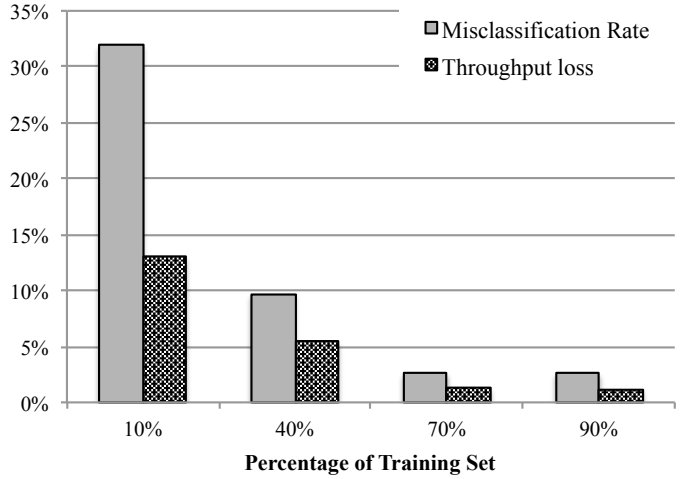

Figure 4.3: Oracle's misclassification rate and % throughput loss.

4.3 Accuracy of the Oracle.

In order to assess the accuracy of the Oracle, we consider the same set of 170 workloads used in Figure 4.2. Figure 4.3 reports the misclassification rate and throughput loss (w.r.t. the optimal solution) when we vary the size of the training set, using the rest of available data as test set. The results are obtained as the average of 200 runs, in which we fed the C5.0 with different (and disjoint) randomly selected test and training sets. The results show that the ML-based oracle achieves very high accuracy, i.e. misclassification rate is lower than 10% and throughput loss is about 5%, if we use as little as the 40% of the collected data set as training set. Interestingly, we observe that the throughput loss is normally less than half of the misclassification rate: this depends on the fact that, in most of the misclassified workloads, the quorum configuration selected by the oracle yields performance levels that are quite close to the optimal ones.

Figure 4.4 provides an alternative perspective on our data set. It reports the cumulative distribution functions of the accuracy achieved by our predictor with different training set sizes, and contrasting them with the normalized performances using all possible configurations for each considered workload. The plot highlights that the choice of the quorum configuration has a striking effect on Q-OPT's performance: for instance, in about 20% of the workloads, the selection of the third best quorum configuration is 40% slower than the optimal one; in the worst case, the plot also shows that the worst (i.e., the 5-th best) choice of the quorum configuration can be even more that 5x slower than the optimal for some workloads. In contrast, it shows that the oracle with even 10% training set helps to achieve higher performance than the 2-nd best choice. Moreover, it confirms that 90% training set strengthen the accuracy of the oracle so

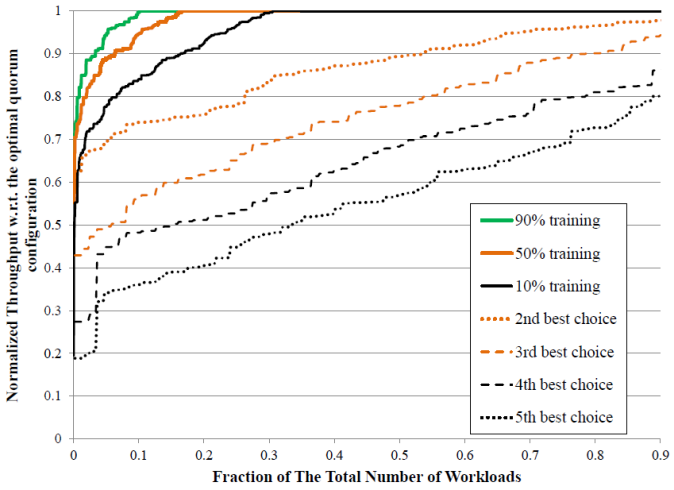

Figure 4.4: Cumulative distribution of the Oracle's accuracy.

that less than 10% of the workloads tends to have a sub optimal performance.

Figure 4.5 allows us to asses to what extent the selection of the features used to generate the ML-based models impacts the model's accuracy. We used five different configurations for this experiment, from A to E. Each configuration progressively adds extra features, including previous configuration features. As the plot shows, the configuration A, which only includes the percentage of write transactions as a feature, achieves poor accuracy, generating a misclassification rate of around 20%. This result confirms the relevance of using a multi-variate model, capable of keeping into account additional factors besides the write percentage (as suggested also by the plot in Figure 4.2). Indeed, the plot clearly shows that, as we include among the provided features also the object size (Conf. B) and throughput (Conf. C) the misclassification rate gets considerably reduced. Our experimental data show also that adding additional features (e.g., conf. D and E, which include statistics on the latencies perceived by get and put operations) does not benefit accuracy, but, on the contrary, can lead to overfitting (Mitchell 1997) phenomena that can ultimately have a detrimental effect on the learner's accuracy.

Finally, in Table 4.1, we report the accuracy achieved by the considered learner when using the, so called, *boosting* technique. The boosting approach consists in training a chain of N learners, where the learner in position i is trained to learn how to correct the errors produced by the chain of learners in position 1,2,...,i. This technique has been frequently reported to yield significant accuracy improvements when used with weak learners. Our experiments do confirm the benefits of this technique, although the relative gains in accuracy are, at least for the considered data set, not so relevant to justify its additional computational overheads.

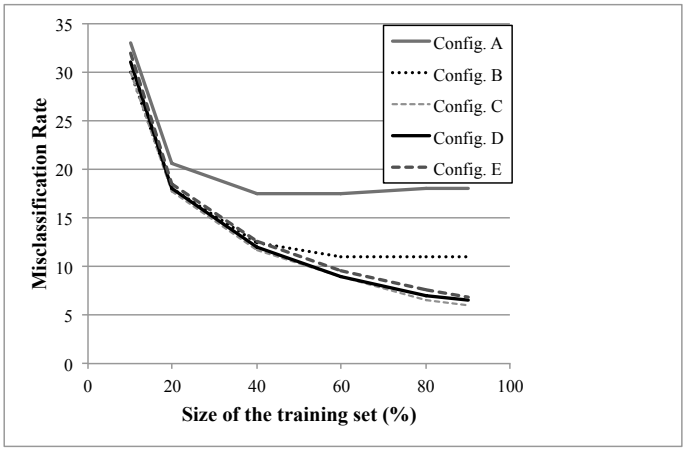

Figure 4.5: Oracle's misclassification rate when varying the set of features used.

	10% Training		50% Training		90% Training	
	unboosted	boosted	unboosted	boosted	unboosted	boosted
Avg. Misclassification (%)	15	14	9	7	6	5
Avg. Distance from optimal (%)	4.6	4	2.2	1.6	1.6	0.9
Avg. Distance when misclassified (%)	14.1	12.9	10.9	9.5	9.5	7.6

Table 4.1: Impact of boosting in the Oracle's performance when varying the training set size.

4.4 Reconfiguration Overhead

As presented in section 3.5, Q-OPT uses a two-phase quorum reconfiguration protocol whose latency is affected by the number of pending operations each proxy has to finish before completing the first phase of the protocol. Thus, we expect to observe an increase in latency as the number of clients issuing concurrent operations increases. Figure 4.6 shows the quorum reconfiguration latency in absence of faults varying the number of clients from 15 to 150 (i.e., close to system's saturation that we estimate at around 165 clients). As expected, the results show a correlation between the latency of the reconfiguration and the request's arrival rate; however, the reconfiguration latency remains, in all the tested scenarios, lower than 15 milliseconds, which confirms the efficiency of the proposed reconfiguration strategy.

Figure 4.7 and Figure 4.8 focus on evaluating the performance of the lazy write-back procedure encompassed by the quorum reconfiguration protocol. Recall that this happens when a read operation gathers a quorum of replies which reveals that the last write applied to that object has been performed using a smaller write quorum than the one currently used. In this case, the read operation is forced to wait for additional replies before returning to the client, and it has to write back the data item using the new (larger) write quorum. The experiment whose results are shown in Figure 4.7 and Figure 4.8 are

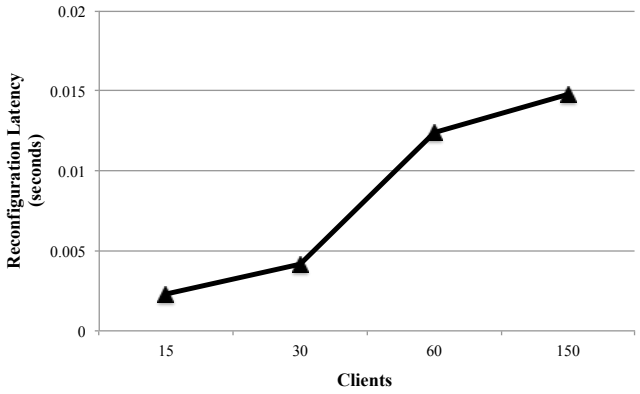

Figure 4.6: Reconfiguration latency while varying the number of clients.

focused precisely on evaluating the overhead associated with these additional writes.

To this end we consider a worst-case scenario in which: i) the system is heavily loaded, ii) data items are accessed with a uniform distribution, and iii) the write quorum increases from 1 to 5 while the application is running a read-dominated (95%) workload. At the beginning of the experiment, we allow the system to run for 3 minutes until it stabilizes and then we trigger the bulk reconfiguration (i.e., for the entire set of 20K data items in the SDS) of the quorum and report throughput over time. The baseline in both the plots represents the scenario where system is under a read-dominated (95%) workload and have the write quorum of 5 throughout the experiment depicting an upper bound for the performance.

Figure 4.7 depicts that as soon as the reconfiguration starts there is a significant performance loss due to the bulk rewrite operations that occur continuously for about 6 minutes until almost all the objects are rewritten with the new (larger) quorum. This behaviour is expected and indeed, we inferred that it would be possible to further reduce the overhead induced by the write back process via reducing the frequency in which write backs are issued, implementing it as an asynchronous background process and batching writes to amortize their cost.

Therefore, we have altered the rewriting module to batch write back requests and update the storage nodes asynchronously. As the Figure 4.8 depicts we could eliminate the dramatic throughput loss, instead it provided a performance gain with respect to the initial stable state. The only drawback in this scenario is that it takes a larger time duration (i.e., about 14 minutes) to reach the final stable state with maximum throughput. Thus, this would imply a trade-off between the time to complete the whole reconfiguration phase and the overhead incurred while this is in progress.

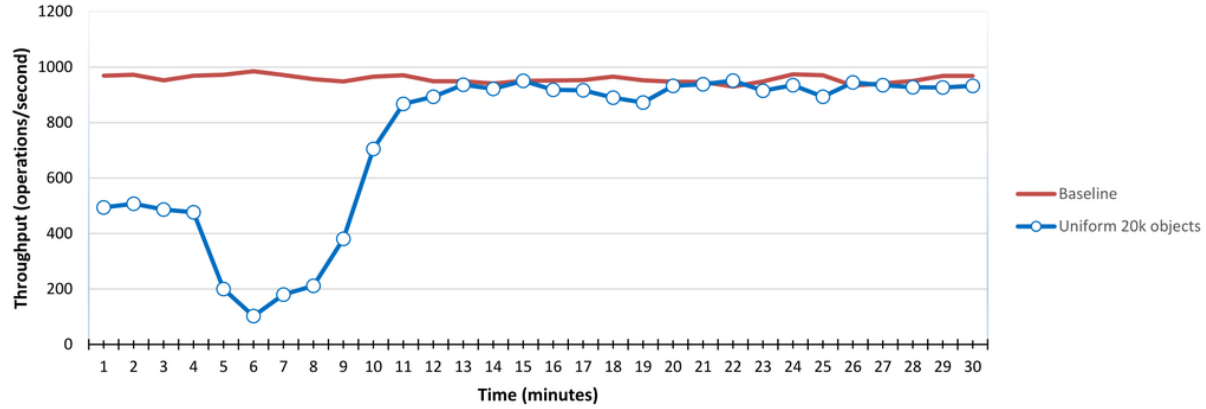

Figure 4.7: Evaluating the overheads associated with the bulk reconfiguration. Write quorum is increased from 1 to 5 in presence of a read-dominated workload.

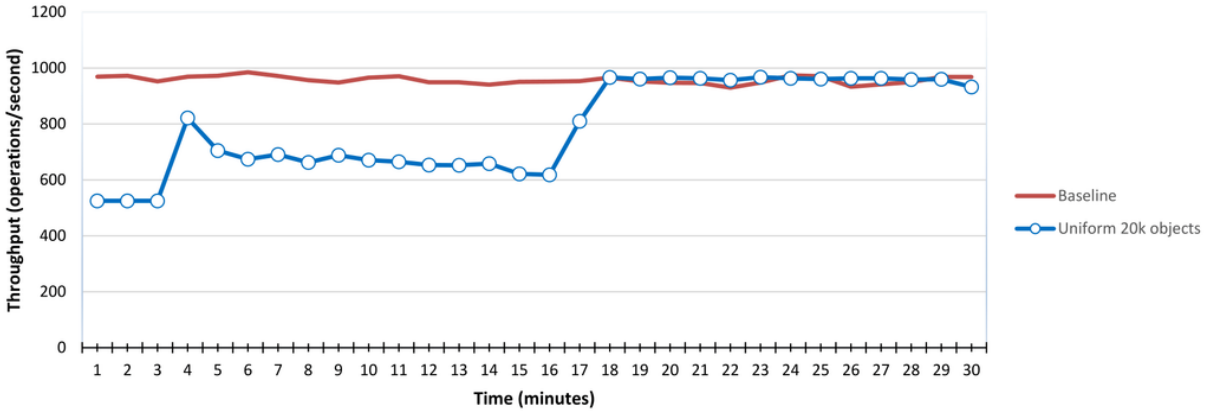

Figure 4.8: Evaluating the overheads associated with the bulk reconfiguration with batching and asynchronous updates. Write quorum is increased from 1 to 5 in presence of a read-dominated workload.

4.5 System Performance

Figure 4.9 and Figure 4.10 evaluate the effectiveness of Q-OPT when faced with time in the presence of complex workloads. We compare few static configurations against Q-OPT and performance of all the static configurations are shown in Figure 4.9. *R1W5*, *R3W3* and *R1W5* are basic static configurations that force the system to use the same quorum for all the objects throughout the experiment. *AllBest* uses the optimal quorum for each of the objects (i.e., an object which is read intensive will be accessed using read quorum 1 and write quorum 5). Finally, *Top10%* uses the optimal quorum for each of the 10% most accessed objects. Notice that *AllBest* and *Top10%* are unachievable configurations in practice since it would require precise pre-knowledge about the workloads of each object. Those configurations will serve us as baselines.

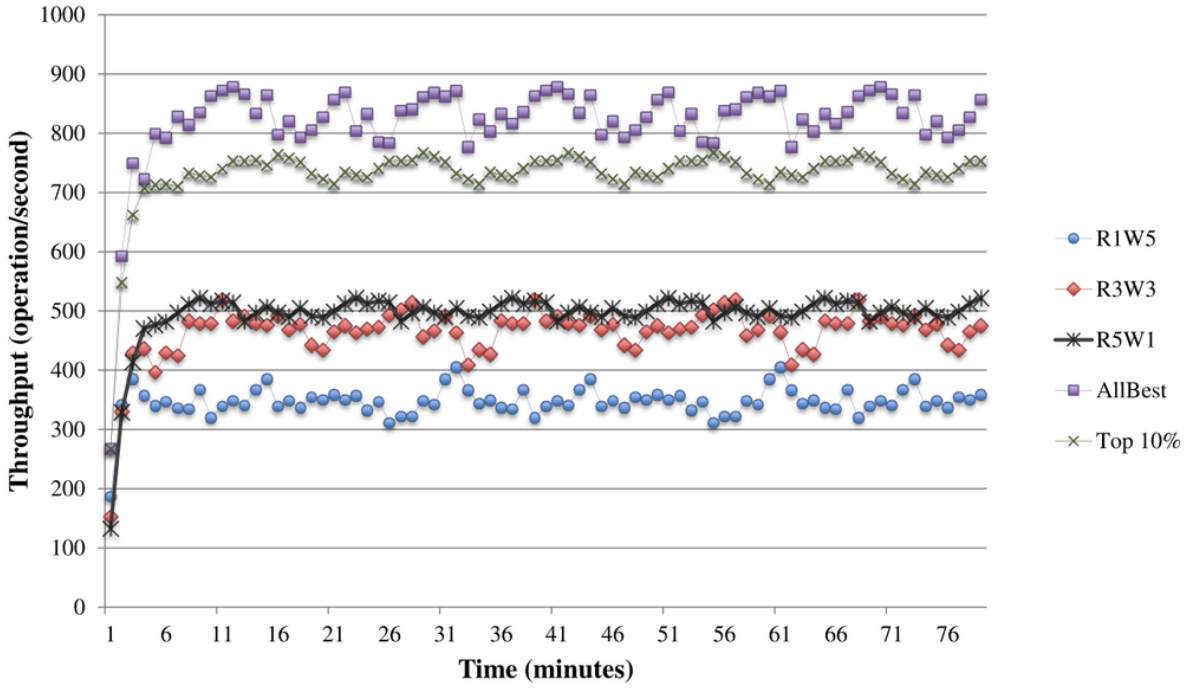

Figure 4.9: Performance of static configurations

In the experiment for Q-Opt depicted in Figure 4.10 we combine two workloads, one read intensive and one write intensive, each of them representing a different tenant. This means that each workload accesses a non-overlapping set of objects. After 40 minutes, we swap the type of workload. Therefore, the read intensive workload becomes write intensive and vice versa. The idea is to observe how Q-OPT reacts to changes in the workloads. For this experiment, Q-OPT runs a fine-grain optimization round every minute. After 20 minutes, the fine-grain optimization phase ends and Q-OPT optimizes the tail.

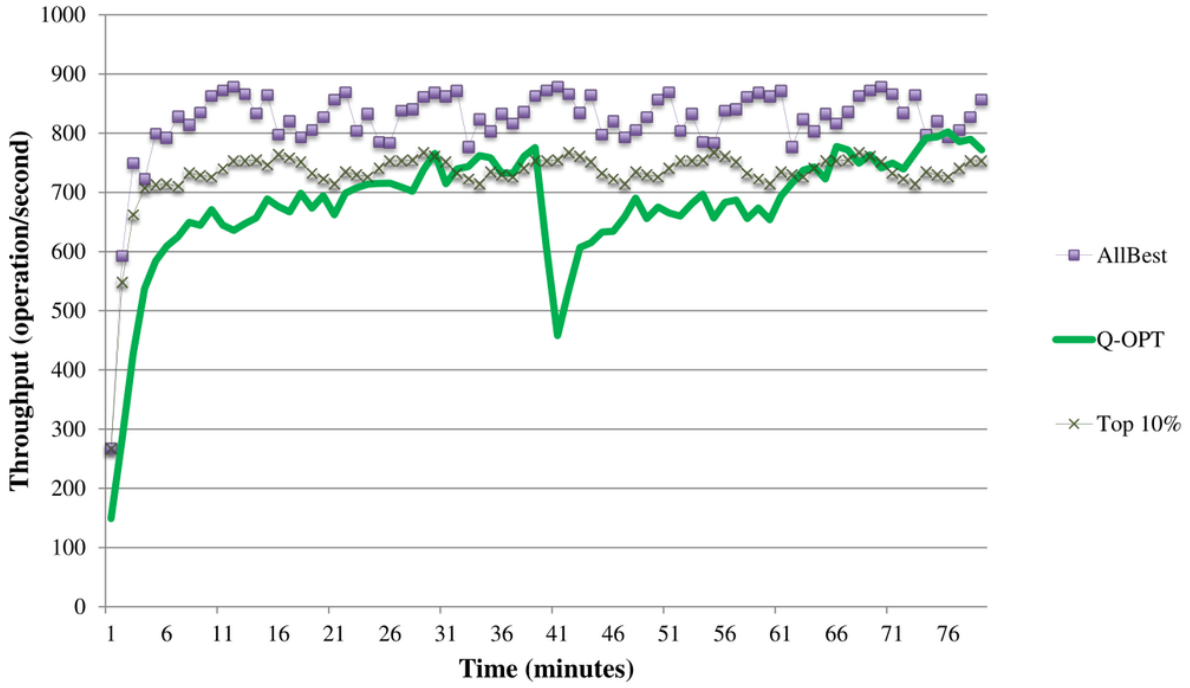

Figure 4.10: Q-OPT's performance in comparison to AllBest and Top10% configurations.

After the swapping (minute 40), Q-OPT starts again the fine-grain optimization phase and continues behaving as described for the first 40 minutes.

Q-OPT behaves as expected. During the first 20 minutes the throughput grows as the fine-grain optimization rounds advance, getting close to the *Top10%* baseline right before optimizing the tail. Once the tail is optimized (after minute 20), the throughput keeps growing even beyond the *Top10%* line and getting closer to the *AllBest* configuration. The *Top10%* configuration does not optimize the tail; therefore, it is expected that Q-OPT outperforms it, at least slightly. As expected, the plot shows that the performance of Q-OPT matches closely that of the optimal configurations. These results confirm the accuracy of Q-OPT's Oracle and highlight that the overheads introduced by the supports for adaptivity are very reduced. After swapping the workloads, Q-OPT experiences a noticeable decrement in the performance since it has to start the optimization phase from zero.

Furthermore, the figures show that none of the basic static configurations is capable of achieving high throughput in comparison to the baselines and Q-OPT. In the worst case, the *R1W5* configuration is more than 2x slower than Q-OPT during stable periods. Even for the best basic static configuration (*R5W1*), Q-OPT still achieves around 45% higher throughput.

4.6 Elimination of Monitoring Overhead with Random Sampling

The optimization protocol we introduce in this work is round based. As already explained in section 3.2, it gathers statistics of object accesses in the current round to find the top-k elements so that those top-k objects can be further tracked in the next round. This initial gathering of statistics tends to incur an overhead at the initial rounds of optimization due to the skew nature of the access distributions. This happens only in initial rounds of optimization because once the head of the skew distribution, which represents the most popular objects, is optimized the overhead would no longer be significant since we do not track already optimized objects.

In order to eliminate this monitoring overhead, random sampling is a simple approach which is quite effective for scenarios with skew access distributions because a larger portion of statistics is dominated by accesses to several popular objects and intuitively a random sample is enough to represent the presence of such accesses with high probability. To this end, when we gather statistics, we only keep metadata for a random set of object accesses rather than keeping them for all the object accesses. To evaluate the impact of using random sampling we performed some experiments to see how the throughput of Q-OPT behaves with different percentages of random sampling.

Figure 4.11 depicts scenarios where Sampling-100% (i.e,. this gather statistics for all object accesses) and Sampling-0% (i.e,. no statistics are gathered at all) as baselines, and random sampling of 10% and 40% depicted by Sampling-10% and Sampling-40% respectively. The plot shows that during initial optimization rounds random sampling of 10% rapidly gets to a higher throughput state defeating all the other cases. But at the 5-th minute (5-th round of optimization) random sampling of 40% gets the lead. This is simply because at this point most of the popular objects have already been optimized, hence 10% sampling no longer able to suggest as many hot objects as it suggested in previous rounds, but in contrast, 40% sampling still be able to see hotter objects which makes it come to lead. After about 8 optimization rounds it seems that both 10% and 40% random sampling will not yield more throughput since they cannot identify non-optimized object access with these sampling percentages. On the contrary, 100% sampling of object accesses seems to gain more throughput even with latter optimization rounds and ultimately will surpass throughput of both 10% and 40% sampling because of two reasons. i) As it gathers statistics for all object accesses it tends to find more hot objects compared to the lower sampling scenarios. ii) Initial optimization rounds have already optimized most of the hot objects so that the monitoring overhead has intuitively reduced for 100% sampling.

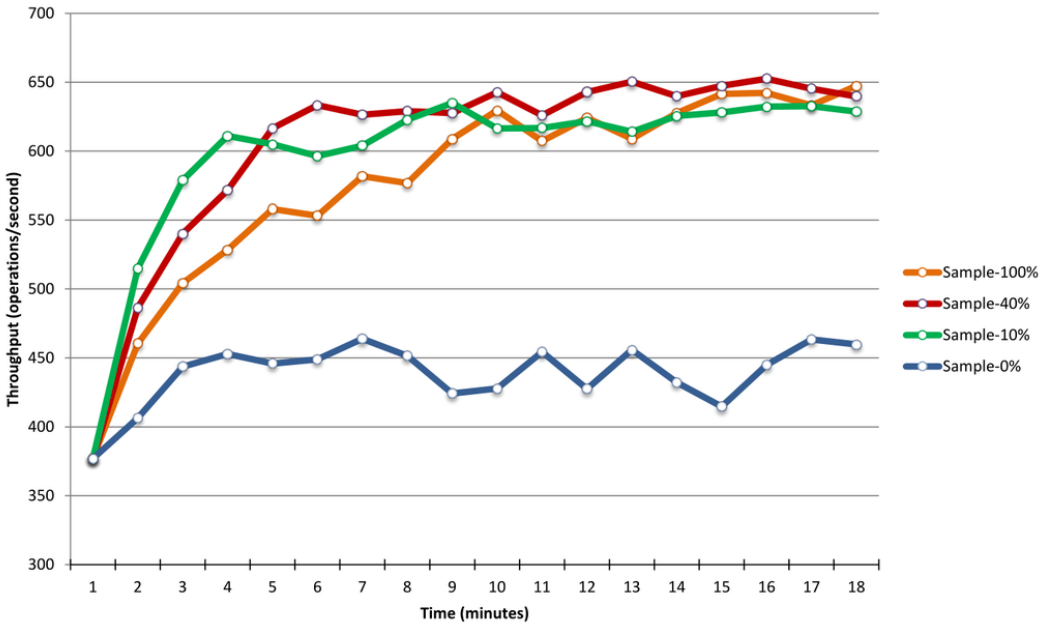

Figure 4.11: Q-OPT's performance with random sampling of statistics.

Based on these results it is beneficial to adapt random sampling in order to eliminate the monitoring overhead at the initial stage of optimization. Furthermore, in order to thrive the system to get its best, it is important to increase the random sampling percentage when it has already optimized sufficient amount of hot objects. Alternatively, instead of changing the random sampling percentage, the time window we track the object accesses (i.e,. here it is 1 minute throughout the experiment) can be increased adaptively when moving further with the optimization rounds, since it will increase the probability of a non-optimized object to be traced by random sampling.

Summary

In this chapter we introduced the experimental evaluation performed on Q-Opt and its results. Initially we presented the behaviour of the throughput with different workload characteristics using the popular workloads specified in YCSB. Then with 170 workloads we showed that there is no liner relationship between write operation percentage and the optimal write quorum configuration which made us to decide to look for a black box machine learning related solution. Next, we showed how robust the Oracle (Machine Learning module) is by plotting the miscalssification rate with varying set of featured used to train the Oracle. It showed that Oracle with only 40% training set produced misclassifications less than 10% and it caused only slightly higher than 5% throughput loss. Hence, with 90% training set

the throughput loss due to misclassification becomes negligibly small. Then, we elaborated the performance of Q-Opt with different static configurations serving as baseline. And it confirmed that Q-Opt can adapt quickly and react to the workload changes automatically and achieve near optimal performance. Furthermore, the overhead evaluation has proved that there is a significant performance gain even with bulk reconfiguration at the worst case scenario (uniform access distribution) when introduced batching and asynchronous updates to the rewriting module. Ultimately, the random sampling technique discussed in order to eliminate the monitoring overhead in the initial optimization rounds helped Q-OPT to adapt even more quickly to the workload changes in the cloud storage system.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduces some directions in terms of future work.

5 Conclusions

Our work tackled the problem of automating the tuning of read/ write quorum configuration in distributed storage systems, a problem that is particularly relevant given the emergence of the software defined storage paradigm. The proposed solution, which we called Q-OPT, leverages on ML techniques to automate the identification of the optimal quorum configurations given the current application's workload, and on a reconfiguration mechanism that allows non-blocking processing of requests even during quorum reconfigurations. Q-OPT's optimization phase first focuses on optimizing the most accessed objects in a fine-grain manner. Then, it ends by assigning the same quorum for all the objects in the tail of the access distribution based on their aggregated profile. We integrated Q-OPT in a popular, open-source software defined storage and conducted an extensive experimental evaluation, which highlighted both the accuracy of its ML-based predictive model and the efficiency of its quorum reconfiguration algorithm.

As future work we would like enhance this system to optimize based on tenants usage. For instance, selecting hot-objects system wide would be not fair for all the tenants using the system because all hot objects may belong to a certain tenant or few of the tenants. This will make poor user experience for the rest of the users and because a cloud storage is shared by vast amount of tenants finding out a proper way to improve fairness in usage would be worthy to research.

References

Arnold, J. (2014). *Openstack Swift: Using, Administering, and Developing for Swift Object Storage*. O'Reilly Media.

Birman, K. and R. V. Renesse (1994). *Reliable Distributed Computing with the ISIS Toolkit*.

Brewer, E. (2012). Cap twelve years later: How the "rules" have changed. *Computer 45*(2), 23–29.

Budhiraja, N., K. Marzullo, F. Schneider, and S. Toueg (1993). *The primary-backup approach*, pp. 199–216. ACM Press/Addison-Wesley.

Chandra, T. and S. Toueg (1996, March). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*(2), 225–267.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber (2008, June). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*(2), 4:1–4:26.

Chihoub, H.-E., S. Ibrahim, G. Antoniu, and M. Perez (2012, Sept). Harmony: Towards automated self-adaptive consistency in cloud storage. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pp. 293–301.

Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, pp. 143–154. ACM.

Couceiro, M., P. Ruivo, P. Romano, and L. Rodrigues (2013). Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, Washington, DC, USA, pp. 1–12. IEEE Computer Society.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, New York, NY, USA, pp. 205–220. ACM.

Dolev, D., I. Keidar, and E. Y. Lotem (1997). Dynamic voting for consistent primary components. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, New York, NY, USA, pp. 63–71. ACM.

Drago, I., M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras (2012). Inside dropbox: Understanding personal cloud storage services. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, New York, NY, USA, pp. 481–494. ACM.

Garcia-Molina, H. and D. Barbara (1985, October). How to assign votes in a distributed system. *Journal of the ACM 32*(4), 841–860.

Ghemawat, S., H. Gobioff, and S.-T. Leung (2003). The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, New York, NY, USA, pp. 29–43. ACM.

Gifford, D. K. (1979). Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, New York, NY, USA, pp. 150–162. ACM.

Guerraoui, R. (2000). Indulgent algorithms (preliminary version). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, New York, NY, USA, pp. 289–297. ACM.

Guerraoui, R. and L. Rodrigues (2006). *Introduction to Reliable Distributed Programming*. Springer.

Hazelcast (2008). Open source in-memory data grid. Accessed on 2015-01-03. `http://hazelcast.org`.

Herlihy, M. (1987, June). Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems 12*(2), 170–194.

Hodge, V. and J. Austin (2004, October). A survey of outlier detection methodologies. *Artificial Intelligence Review 22*(2), 85–126.

IMEX-Research (2015). The promise and challenges of cloud storage. Accessed on 2015-01-02. `http://tiny.cc/sdpftx`.

Infinispan (2009). Distributed in-memory key/value data grid and cache. Accessed on 2015-01-03. `http://infinispan.org`.

Jiménez-Peris, R., M. Patiño Martínez, G. Alonso, and B. Kemme (2003, September). Are quorums an alternative for data replication? *ACM Transactions on Database Systems 28*(3), 257–294.

Kalman, R. et al. (1960). A new approach to linear filtering and prediction problems. *Journal of basic Engineering 82*(1), 35–45.

Kraska, T., M. Hentschel, G. Alonso, and D. Kossmann (2009, August). Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the VLDB Endowment 2*(1), 253–264.

Lakshman, A. and P. Malik (2010, April). Cassandra: A decentralized structured storage system. *Special Interest Group on Operating Systems (SIGOPS) Operating Systems Review 44*(2), 35–40.

Lamport, L. (1978, July). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*(7), 558–565.

Liskov, B. (1991). Practical uses of synchronized clocks in distributed systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '91, New York, NY, USA, pp. 1–9. ACM.

Malkhi, D. and M. Reiter (1997). Byzantine quorum systems. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, New York, NY, USA, pp. 569–578. ACM.

Metwally, A., D. Agrawal, and A. El Abbadi (2005). Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, Berlin, Heidelberg, pp. 398–412. Springer-Verlag.

Mitchell, T. (1997). *Machine learning*. McGraw Hill series in computer science. McGraw-Hill.

Morris, J. H., M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith (1986, March). Andrew: A distributed personal computing environment. *Communications of the ACM 29*(3), 184–201.

Naor, M. and A. Wool (1998, April). The load, capacity, and availability of quorum systems. *SIAM Journal on Computing 27*(2), 423–447.

Openstack-Swift (2009). Highly available, distributed, eventually consistent object/blob store. Accessed on 2015-01-03. `http://docs.openstack.org/developer/swift`.

Page, E. S. (1954). Continuous inspection schemes. *Biometrika 41*(1/2), pp. 100–115.

Paiva, J., P. Ruivo, P. Romano, and L. Rodrigues (2013). Autoplacer: Scalable self-tuning data placement in distributed key-value stores. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA, pp. 119–131. USENIX.

Peleg, D. and A. Wool (1995, December). The availability of quorum systems. *Information and Computation 123*(2), 210–223.

Quinlan, J. (2012). C5.0/see5.0. `http://www.rulequest.com/see5-info.html`.

Quinlan, J. R. (1975). The id3 algorithm Accessed on 2015-06-14. `http://www.cise.ufl.edu/~ddd/cap6635/Fall-97/Short-papers/2.htm`.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon (1985). Design and implementation or the sun network filesystem.

Schneider, F. B. (1990, December). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*(4), 299–319.

Shvachko, K., H. Kuang, S. Radia, and R. Chansler (2010, May). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10.

Tai, A. and J. Meyer (1996, Feb). Performability management in distributed database systems: an adaptive concurrency control protocol. In *Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1996. MASCOTS '96.*, pp. 212–216.

Ullman, J. D. (1990). *Classical Database Systems*. Computer Science Press.

Wada, H., A. Fekete, L. Zhao, K. Lee, and A. Liu (2011). Data consistency properties and the tradeoffs in commercial cloud storages: the consumers' perspective.

Wang, X., S. Yang, S. Wang, X. Niu, and J. Xu (2010, Nov). An application-based adaptive replica consistency for cloud storage. In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pp. 13–17.

Wiesmann, M., F. Pedone, A. Schiper, B. Kemme, and G. Alonso (2000). Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems, 2000.*, pp. 464–474.

Wolfson, O., S. Jajodia, and Y. Huang (1997, June). An adaptive data replication algorithm. *ACM Transactions on Database Systems 22*(2), 255–314.