ELASTICITY, AVAILABILITY AND CONSISTENCY IN CLOUD-BASED AGGREGATE DATA STORES

by

Neil Burke

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2015

© Copyright by Neil Burke, 2015

Table of Contents

List of Tables . . . . **v**

List of Figures . . . . **vi**

Abstract . . . . **x**

List of Abbreviations Used . . . . **xii**

Acknowledgements . . . . **xiii**

Chapter 1 Introduction **1**

1.1 Structure of the Thesis 8

Chapter 2 Background **9**

2.1 Distributed and Cloud Computing 9
2.1.1 Cloud Providers 10
2.1.2 Amazon Web Services 10
2.1.3 Configuration Management 11
2.1.4 Distributed Coordination in the Cloud 15

2.2 OLTP and OLAP Data Stores 16
2.2.1 OLTP 16
2.2.2 OLAP 17

2.3 Distributed Consistency 18
2.3.1 Quorum-based Replication 18
2.3.2 Types of Consistency 20

2.4 PBS for Key-value Stores 22

Chapter 3 *vOLAP* **26**

3.1 Related Work 28

3.2 vOLAP Architecture 29
3.2.1 Architecture Overview 30
3.2.2 System Image 31
3.2.3 Load Balancing 32

3.3 Elastic *vOLAP* on a Managed Cloud 34
3.3.1 SaltStack-managed *vOLAP* Deployment 34

3.3.2 Amazon EC2 Deployment 36
3.3.3 OpenStack Deployment 37
3.3.4 Automated Scaling 38

3.4 Experimental Evaluation 38
3.4.1 Real-Time Load Balancing 39
3.4.2 Horizontal Scale-Up Performance 39
3.4.3 Insert and Query Performance 41
3.4.4 Coverage Impact 41

Chapter 4 Aggregate Quorum-based Replication 46

4.1 Aggregate Quorums 46
4.1.1 Motivation 47
4.1.2 Aggregate Model 47
4.1.3 Aggregate Quorum-based Model 50
4.1.4 Quorum Reads and Writes 50
4.1.5 Combination of Partial Aggregations 53

4.2 Re-engineering *vOLAP* to Support Quorum-based Replication 59
4.2.1 Subset Replication 60
4.2.2 Replica Management 61
4.2.3 Reading and Writing 62
4.2.4 Split Coordination 63

4.3 Evaluation 64
4.3.1 Infrastructure Overhead 65
4.3.2 Replication Factor 66
4.3.3 Read and Write Quorums 67

Chapter 5 PBS for Aggregate Data Stores (A-PBS) 71

5.1 Metrics of Aggregate Staleness 71
5.1.1 Foundations 71
5.1.2 (t, k)-staleness 73
5.1.3 Data Stream Sizes 74
5.1.4 Staleness and Error 76
5.1.5 Probabilistic Staleness 78

5.2 Evaluating A-PBS for Generic Aggregate Systems 80
5.2.1 An Enhanced Aggregate Model for A-PBS Simulation 80
5.2.2 Simulation 81

5.3 Evaluating A-PBS for Quorum-based Aggregate Systems 85
5.3.1 Parameters 85
5.3.2 Simulation 88

5.4 Evaluation 90
5.4.1 Configuration 90
5.4.2 Bounded Staleness 93
5.4.3 Latency 96
5.4.4 Bounded Error 97
5.4.5 Data Stream Size 100

Chapter 6 Conclusion 106

6.1 Summary 106

6.2 Future Work 106

Bibliography 109

List of Tables

3.1 System parameters 31

4.1 Timestamps (greater is more recent) and point data used to create two partial aggregations from two buckets in the same bucket set. 55

4.2 Point data after 4 insertions on a system with $N = 3$ replicas and $W = 2$. 59

5.1 The list of system parameters in the model 81

5.2 The set of system parameters in the new quorum-based model . 88

5.3 The set of default system and simulation parameters used for the experiments in Section 5.4 93

List of Figures

2.1 An example of a read operation using quorum rules. C corresponds to the client, while B_1, B_2 and B_3 refer to the nodes containing the replicated data relevant to the query. Solid lines correspond to read requests, and dotted lines correspond to responses to the client. 20

2.2 The WARS model for Dynamo-style data stores, from [22]. . . 25

3.1 Dimension hierarchy for TPC-DS [12] benchmark data 26

3.2 System overview 31

3.3 *vOLAP* deployment using SaltStack 36

3.4 Load balancing data size per worker as database size N and number of workers p increases. 40

3.5 Query and insert performance with increasing system size. Database size N and number of workers $p = N/50Mil$ ($4 \leq p \leq 20$) both increasing. Low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%. 43

3.6 Performance for various workload mixes and query coverages. TPC-DS; $N = 1$ billion; $p = 20$; $j = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%. 44

3.7 Effect of coverage on query performance. 45

4.1 An insert operation under our generic real-time OLAP model . 49

4.2 A query operation under our generic real-time OLAP model . 49

4.3 Diagram illustrating the node structure of the model. Clients C_1 to C_b each communicate with one of j indices, which communicate with as many as m buckets. 50

4.4 Diagram illustrating the node structure of our aggregate model, modified for quorum-based replication. Clients C_1 to C_b each communicate with one of j indices, which communicate with as many as m groups of buckets, each group containing N replicas. Dotted boxes correspond to groups of structures which replicate the same structure. 51

4.5 An example quorum-based insert operation in the aggregate model with $W = 2$ 53

4.6 An example quorum-based query operation in the aggregate model with $R = 1$ 54

4.7 Communication steps required for a coordinated split across three workers containing the same subset. 65

4.8 Latency and throughput performance for various workload mixes and query coverages prior to the implementation of quorum replication. TPC-DS data; $N = 10$ million; $p = 6$; $j = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%. 67

4.9 Latency and throughput performance for various workload mixes, query coverages, and replication factors. TPC-DS data; $N =$ 10 million; $p = 6$; $j = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%. . 69

4.10 Latency performance for various workload mixes, query coverages, and values of W and R. TPC-DS data; $N = 10$ million; $p = 6$; $j = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%. 70

5.1 A DATA(5, f, D) insert stream. The white circles represent the points in time in which inserts in the stream are sent from the client. All inserts are spaced $i = 1/f$ seconds apart. 73

5.2 An insert stream and query for (t, k)-staleness. The white circles and black diamond represent the respective initiation time of the inserts and query. 74

5.3 A query that has $(t, k{=}1)$-staleness. The upper bar represents the time of initiation of a query or insert, while the bottom bar represents the time at which the corresponding insert is readable, or the time at which the query begins. The last two inserts in the stream and the query are reordered, so more than $k = 1$ inserts are missed. 75

5.4 A query that does not have $(t, k{=}1)$-staleness. The upper bar represents the time of initiation of a query or insert, while the bottom bar represents the time at which the corresponding insert is readable, or the time at which the query begins. Since $k = 1$, the reordering of the last insert in the stream and the query does not impact (t, k)-staleness. 75

5.5 A simulation of a single trial of an insert stream and a query. The last insert, represented as a black circle, is synchronized after the query, represented as the black vertical line, so this simulated query is considered to have $(t, k{=}0)$-staleness. 83

5.6 A simulation of a single insert with $N = 3$ and $W = 2$. After sampling various distributions, the key times marked by asterisks are generated for later computation during query simulation. . 91

5.7 A simulation of a single query on a system with $m = 1$, $N = 3$, $R = 1$. Key times are marked with asterisks. After sampling various distributions, the time at which each bucket receives the simulated query request is recorded, as well as the specific buckets that make up the first R responses. 91

5.8 Plot of the cumulative distribution functions for the latency distributions used in Section 5.4. Since each latency distribution is the same, all lines overlap each other. 94

5.9 Probability of bounded $(t, k{=}0)$-staleness with varying quorum configurations across increasing values of t. 95

5.10 Probability of bounded $(t, k{=}1)$-staleness with varying quorum configurations across increasing values of t. 96

5.11 Probability of bounded $(t, k{=}2)$-staleness with varying quorum configurations across increasing values of t. 97

5.12 Probability of bounded $(t, k{=}0)$-staleness for 25%, 50%, 75% and 100% coverage. 98

5.13 Expected number of missed inserts in a query with 50% coverage for varying quorum configurations. 99

5.14 Average latency of an insert operation for varying quorum configurations 100

5.15 Average latency of a query operation for varying quorum configurations 101

5.16 Plot of the cumulative distribution functions for the measure distributions used for bounded (t, ϵ)-staleness experiments. . . 102

5.17 Probability of bounded $(t, \epsilon{=}0.00000)$-staleness with varying aggregation functions and measure distributions for `[N=3, W=1, R=1]` across increasing values of t. 102

5.18 Probability of bounded (t, ϵ=0.00010)-staleness with varying aggregation functions and measure distributions for [N=3, W=1, R=1] across increasing values of t. 103

5.19 Probability of bounded (t, ϵ=0.00025)-staleness with varying aggregation functions and measure distributions for [N=3, W=1, R=1] across increasing values of t. 103

5.20 Probability of bounded (t, ϵ=0.00050)-staleness with varying aggregation functions and measure distributions for [N=3, W=1, R=1] across increasing values of t. 104

5.21 Probability of bounded (t, ϵ=0.00100)-staleness with varying aggregation functions and measure distributions for [N=3, W=1, R=1] across increasing values of t. 104

5.22 Expected relative error for [N=3, W=1, R=1] and varying aggregation functions and measure distributions across increasing values of t. 105

5.23 Probability of bounded (t, ϵ)-staleness with $t = 0ms$, $\epsilon = 0.00010$ and varying aggregation functions and measure distributions for [N=3, W=1, R=1] across increasingly larger insert streams (log scale). The dotted vertical line represents the point at which the stream is said to be sufficient large (for our 99th percentile calculation of ϕ). 105

Abstract

Quorum-based replication is popular in many distributed online transaction processing data stores, especially key-value stores like Dynamo or Cassandra. By committing each write operation to N different nodes in a distributed system, a data store can achieve a level of fault tolerance. Since writes invariably arrive at different nodes at different times, it is useful to know the way in which the lack of *consistency* between nodes will affect the result of read operations. Read and write quorums, a mechanism used to ensure operational consistency between a subset of nodes in the system, can be used to reduce the system's level of inconsistency at the cost of latency. While many distributed key-value data stores have very poor worst case data consistency, they often work well in practice. In 2012, Probabilistically Bounded Staleness (PBS), a method of modelling consistency of key-value systems, was introduced to better analyze the performance of real systems and to help model the trade-off between consistency and latency.

This thesis has three central contributions. First is an extension of *vOLAP*, a real-time distributed online analytical processing (OLAP) system, to a dynamic cloud environment, in order to support an elastic model of automated resource management. On an 11-node AWS cloud, *vOLAP* is capable of ingesting 140,000 inserts per second and answering 40,000 complex range queries per second.

Next, we present a model of quorum-based replication adapted for OLAP-style data stores. The challenges encountered and experimental results from implementing the model within *vOLAP* are also described. Not surprisingly, quorum-based replication comes at a cost. A *vOLAP* system with $N = 3$ executes 80,000 inserts per second and 20,000 queries per second. While this is a significant drop in performance from the $N = 1$ case, this version of *vOLAP* is essentially doing 3 times the work at approximately 2 times the cost in performance, achieving a high level of fault tolerance.

Finally, Aggregate Probabilistically Bounded Staleness (A-PBS) is introduced, which adapts ideas originally developed in PBS for the simple key-value setting to

a wide class of eventually consistent aggregate data stores. Associated with A-PBS is a simulation framework for exploring the level of consistency of OLAP quorum systems. A Monte Carlo simulation is used to analyze the impact of varying key system parameters on staleness. The A-PBS model illuminated some surprising results about consistency in an aggregate data store under typical query types such as *count*, *sum*, *max*, and *mean*. For example, relative error rates were found to be typically extremely small even when performing large aggregate queries.

List of Abbreviations Used

A-PBS	Aggregate Probabilistically Bounded Staleness
AMI	Amazon Machine Image
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
MBR	Minimum Bounding Rectangle
MDS	Minimum Describing Set
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
PBS	Probabilistically Bounded Staleness
PDC-tree	Parallel DC-tree
S3	Simple Storage Service
vOLAP	VelocityOLAP

Acknowledgements

I would like to sincerely thank my supervisor, Dr. Andrew Rau-Chaplin, for the insight and guidance given throughout the completion of this thesis. I would also like to thank Dr. Frank Dehne and David Robillard, for their constructive weekly discussions. I also express my thanks to the readers of my thesis committee: Dr. Qigang Gao and Dr. Norbert Zeh. Finally, I thank my parents, for their unending support.

Chapter 1

Introduction

Recently, with the advent of inexpensive cloud computing resources, scalable distributed data stores have surged in popularity [58, 34, 57, 9, 81, 29]. Such systems focus on horizontal scalability, and take advantage of cheap, pay by the hour, compute nodes provisioned through the cloud [28]. In doing so, these systems are able to distribute query and insert load across many "shared nothing" compute nodes, improving latency and throughput performance. Three important problems within the context of cloud-based distributed data stores are dynamic hardware scaling (i.e. elasticity), availability, and consistency. In this thesis, we explore how techniques and models used to address these problems in the context of scalable key-value data stores can be applied to scalable Online Analytical Processing (OLAP) or aggregate data stores.

A type of distributed Online Transaction Processing (OLTP) store especially relevant to this thesis is the key-value store. Popularized in [34], key-value stores contain abstract data (or *values*) addressable by specific *keys*. Key-value stores typically use a hash function to convert keys to numeric location values. A client can add an item to a key-value store by sending a key-value pair to the system. The hash value of the key is computed to determine where the value should be stored. Consequently, clients can read an item from the store by sending a key to the system. Hashing the key results in the location of the associated value, which can then be accessed to return the value of the associated value of the key back to the client. This model lends itself well to distributed environments; a set of nodes in a system can be assigned to contain a specific range of hashed keys and their associated values. Read and write operations only require a single node to be accessed, so the system's throughput capacity benefits greatly from additional nodes.

For many key-value data stores, availability is a critically important quality [34]. A system is highly available if reads and writes are processed without error while

some nodes in the system are experiencing hardware, software or network errors (for example, a node suddenly breaks down or goes offline). Key-value stores typically opt for quorum-based replication to improve a system's availability [57, 34]. Under quorum-based replication, key-value systems store N redundant replicas of each chunk of key-value pairs. Each replica replicates their respective chunk of key-value pairs on separate nodes in the system. Consequently, each read and write operation is sent to all N replicas. In doing so, if a single node in the system fails, $N - 1$ nodes replicating the same data remain accessible. Increasing N increases the availability of a system. Replicas also can be used to further spread load across the system.

However, introducing redundant replication to a system introduces the problem of *consistency*. Since networks are unpredictable, each insert operation will arrive at the N different replicas at different times. This leads to the data stored on replicas being inconsistent from one another until each insert arrives on each of the N replicas. The direct consequence of inconsistent replicas is the possibility that any read can return an outdated result. One widely popular method of approaching this problem is to implement a quorum consensus algorithm [51, 43, 44]. With a quorum consensus, the variables W, the *write quorum* and R, the *read quorum* are used to ensure each operation has been applied to a subset of replicas. For example, under a write operation, all N replicas are sent the write. After W replica responses indicated the write was committed, a response is sent back indicating the write was successful. Note however that in the background, writes for the remaining $N - W$ replicas are in progress. Under a read operation, all N replicas are queried and, like quorum-based writes, R replica responses are waited for. Once the R responses have returned, the most up to date result is selected. By selecting values of W and R such that $W + R > N$, at least one of R reads is guaranteed to overlap with one of W of the most recent writes, guaranteeing consistency [43].

However, setting W and R such that $W + R > N$ comes at a cost. For an operation to complete successfully, a reply from at least W or R replicas is required; if more than $N - W$ or $N - R$ replicas do not respond, the operation fails. When W and R are large, the system's availability suffers. In addition, large values of R and W also affect read and write latency respectively, as more read and write replies are required from replicas. As a result, many key-value stores opt to use smaller values

of W and R [34, 57, 22], sacrificing consistency for latency and availability. In 2012, Probabilistically Bounded Staleness (PBS) [22], a method of modelling consistency of key-value systems when $W + R \leq N$, was introduced to better analyze the performance of real systems and to help model the trade-off between consistency and latency.

OLAP systems attempt to quickly present generally summarized, broadly scoped aggregations of information [30]. OLAP systems are typically used to support the "decision makers" of a company (i.e. executives, analysts, managers), to compute complex summarizations of hierarchical data across very large databases. This is in contrast to OLTP data stores, which are primarily focused on efficiently writing and reading individual transactions. In general, OLAP data stores can be viewed on an abstract level as a large, multi-dimensional data cube [77, 46].

Each dimension of a data cube corresponds to an entity or metric that is deemed important for making OLAP supported decisions. For example, time, location and product could be used for the dimensions of a 3-dimensional data cube representing sales at a store. A hierarchy is associated for each dimension, which associates each possible value in a dimension to a point on the hierarchy. For example, a hierarchy for a time dimension may be year, month, day and hour. Inserts write numerical *measure* values to specific hierarchical points within the cube. Using the dimension hierarchies, queries retrieve measure values within a specified area of the cube, and combine the measure values with a provided binary associative aggregation function (for example *sum*) to yield a single value, which is then returned as the query result.

Often, data cubes in OLAP systems are constructed statically, in that they compute aggregations for a wide range of queries in advance [50]. In doing so, such systems are able to quickly answer complex queries, but cannot support real-time insertion of new points. Recently, using the same affordable cloud technology that spurred growth of key-value systems, real-time OLAP systems, which use dynamically changing data cubes to support real-time inserts and queries, have been presented [55, 81]. As real-time OLAP systems grow into maturity, we believe the lessons learned from distributed OLTP stores, specifically with regards to elasticity, availability and consistency can be applied to real-time OLAP systems.

In this thesis, we explore elasticity, availability and consistency for distributed cloud-based real-time OLAP systems. Chapter 3 starts with a description of *VelocityOLAP*

(*vOLAP*) [55], a real-time OLAP system, built collaboratively by the author and other students and researchers. In *vOLAP*, nodes within a distributed system communicate with each other in order to answer OLAP queries while simultaneously supporting ingestion of new data in real time. *Worker* nodes in *vOLAP* store portions of the data cube, called subsets, and can aggregate arbitrary ranges within subsets to answer queries. *Server* nodes record which portions of the data cube belong to which subsets, which the server can use to direct insert or query operations to the appropriate worker(s). *Client* nodes send insert or query requests to server nodes. Finally, a single *manager* node is used to load-balance the system and ensure the even distribution of subsets across the worker nodes.

The initial version of *vOLAP* was designed for a static hardware platform, and used a rudimentary deployment process. Using a list of IP addresses of each node in the distributed system, a simple program was used to initiate one of the *vOLAP* binaries on each node in the system. For running simple tests on a cluster of fixed nodes, this method was serviceable, however, deploying *vOLAP* on cloud infrastructure requires a more robust approach. This deployment method had no facilities to automatically access and provision cloud resources; all instances and storage must be manually provisioned from the cloud provider's web client. Because *vOLAP* has no way of automatically provisioning or deallocating cloud resources, it has no way to elastically adjust for load. Finally, from a pragmatic perspective, this type of deployment on the cloud was too time consuming and unable to support sophisticated benchmarking.

Using SaltStack [10], a distributed configuration management system, *vOLAP*'s deployment architecture was redesigned and implemented to support automatic provisioning of cloud resources and easier management of existing nodes within the system. Under the new deployment infrastructure, a node running the SaltStack *salt-master* binary can provision new nodes from either Amazon EC2 [2] or OpenStack cloud providers by invoking a script. Nodes provisioned in this way launch running the *salt-minion* program, which allows the master node to coordinate and manage the newly provisioned node. For example, *vOLAP* binaries can be copied over to all minion nodes with a single SaltStack command initiated from the master, or the *vOLAP* worker binary can be initiated on all minions designated as worker nodes with a SaltStack command. Most importantly, SaltStack integration allows *vOLAP*

to provision new nodes elastically as it sees fit.

Experimental results of *vOLAP* running on a SaltStack managed Amazon EC2 cloud are presented. The `c3.8xlarge` Amazon EC2 instance type was used for server nodes, `c3.4xlarge` for workers, and `c3.2xlarge` for clients and the manager. On a system with 2 clients, 2 servers, 20 workers storing 1 billion points of TPC-DS data [12], *vOLAP* is capable of answering approximately 13,000 complex aggregation queries per second, or ingesting approximately 52,000 inserts per second. Average query latency is 0.3 seconds, while average insert latency is 0.05 seconds. The elastic scalability of *vOLAP* is also presented. Starting with 4 worker nodes, *vOLAP* added 4 additional worker nodes for every 200 million points added to the system, up to 1 billion points. *vOLAP* was shown to scale well in an elastic cloud environment, experiencing little change in latency and throughput as more workers and data points were added.

One problem with the initial design of *vOLAP* was its low level of availability. Specifically, in the event of hardware or network failure of a data-storing worker node, any queries or inserts that depend on that specific worker node will fail. In Chapter 4, we rectify this by introducing a method of quorum-based replication for OLAP-style data stores. First, quorum replication is described in an abstract aggregate data store model. In this model, similar to *vOLAP*, *buckets* store points of a data cube, *indices* route operations to the appropriate bucket(s), and *clients* send query and insert operations. Each bucket is redundantly replicated N times. Quorum inserts in this model behave similar to key-value store writes: inserts sent from the client arrive at an index, which are then routed to N buckets. A response is sent from the index to the client after W replies are received. Likewise, queries are sent from a client, arrive at an index, and are then routed to N bucket replicas for each unique bucket relevant to the query. R replies are required for each set of bucket replicas.

One interesting problem that arises from quorum-based replication for aggregate data stores is the process of selecting which bucket aggregation from a set of replicas is the best, or most consistent. Recall that key-value stores simply use timestamps to pick the replica response with the most recent write, however this does not apply for aggregations, as time is not indicative of which aggregation is closest to the true aggregation. We propose a method of selecting aggregation results from buckets

replicating the same data using only the aggregation function and each replica's aggregation value. This method guarantees that the best aggregation result is chosen for specific aggregation functions, while still being efficient to implement in practice. We describe methods of bucket aggregation selection for six aggregation functions commonly found in OLAP systems.

With the abstract model in place, it is applied to *vOLAP* in order to improve the system's availability. Because of load balancing and hardware heterogeneity considerations, we map subsets to buckets in the aggregate model; each subset in *vOLAP* is replicated N times. A new structure is introduced to the servers, which keeps track of which subsets are replicated on which workers. Insert and query operations are modified to support quorums for the server, but remain unchanged for clients and workers. The introduction of replicas required some design changes to the load balancer. The *split operation*, previously used to split one subset into two, had to be redesigned in order to synchronize the split across each subset replica in order to ensure correctness. A *replicate* was added to the load balancer. If the manager observes that a subset is under replicated (that is, has fewer than N replicas), then the replicate operation is triggered, which coordinates a subset from one worker to be copied to another.

Performance of the new highly available, fault tolerant *vOLAP* system on Amazon EC2 is presented. On a system storing 10 million TPC-DS [12] points, consisting of six `c4.xlarge` worker nodes, two `c4.2xlarge` server nodes, two `c4.xlarge` client nodes and one `c4.xlarge` manager node, *vOLAP* with `[N=1, W=1, R=1]` saw no difference in latency and throughput when compared with the previous version of *vOLAP* on the same system. Going from `[N=1, W=1, R=1]` to `[N=3, W=1, R=1]` resulted in an approximate 50% decrease in query and insert throughput, as workers are responsible for handling three times the operations. Latencies for queries increased approximately by a factor of 2. Insert latency saw the largest increase, going from 0.025 seconds to 0.05 seconds. The value of the read and write quorums had no noticeable impact on latency, as each node in the system was connected by a high-speed local network.

The addition of quorum-based replication to *vOLAP* addresses the problem of availability, but introduces the problem of subset data consistency; insert operations will inevitably arrive at different subset replicas at different times, and queries therefore

may report incorrect aggregations. Large values of W and R can be used to achieve high, or even perfect consistency. However, this comes at a great cost to system availability. Therefore, it is important to know the cost inconsistent data has on the output of aggregate queries.

Finally in Chapter 5 of this thesis, we describe *Aggregate Probabilistically Bounded Staleness (A-PBS)*, a means of examining consistency within aggregate data stores. Adapted from PBS [22], which examines consistency in key-value stores, A-PBS builds on top of the previous aggregate models described in this thesis in order to support consistency analysis of generic aggregate systems, as well as quorum-based aggregate systems.

A-PBS is composed of two components: First, definitions for describing an aggregate system's data stream, and the state of consistency of individual aggregate queries are presented. Unlike PBS, which views consistency only from the perspective of the number of missed writes, A-PBS uses both the number of missed inserts as well as the relative numerical error of the query when exploring consistency. (t, k)-*staleness* describes queries delayed by t units of time which have missed more than k inserts. (t, ϵ)-*staleness* describes queries delayed by t units of time which have a relative numerical error greater than ϵ. These definitions are then applied to describe a system's probability of yielding consistent queries: *bounded* (t, k)-*staleness* and *bounded* (t, ϵ)-*staleness*.

A-PBS's second component is a Monte Carlo simulation, which, when given a list of specific system parameters, can be used to estimate the probability of consistency of an aggregate data store. This simulation allows users to explore the trade-offs between consistency and latency in aggregate data stores. The simulation emulates the process of ingesting a number of inserts from a data stream. A simulated aggregate query follows the ingestion of the insert stream, and the number of missed points by the query, as well as the result of the aggregation function is used to determine if the simulated query is acceptably consistent. By repeating this process a number of times, the probability of a query being acceptably consistent can be estimated. The specific process of simulation is presented for aggregate data stores with and without quorum-based replication.

Finally, A-PBS is evaluated on a toy aggregate data store with quorum-based

replication. As expected, for quorum configurations with a replication factor of at most 3, a system with quorum configuration `[N=3, W=1, R=1]` yields the worst consistency. Surprisingly, it was observed that setting $R = 2$ has a much larger positive impact on consistency than setting $W = 2$, due to a subtle difference in the way aggregate read quorums work. For arbitrarily large data streams, a query with `[N=3, W=1, R=1]` that covers 50% of points in the stream has approximately a 50% probability of being perfectly consistent. However, the same query is only expected to miss one insert on average. Consequently, the relative error of queries is very small, and improves as more points are added. The *max* aggregation function was found to have almost 100% probability of having no error. The *sum* aggregation function was found to yield greater relative errors, on average, than the *mean* aggregation function.

1.1 Structure of the Thesis

The thesis begins with Chapter 2, wherein background for essential relevant concepts and technologies is presented. Chapter 3 covers *vOLAP*, a distributed aggregate data store built in collaboration with the author of this thesis and other researchers. Within the chapter, Section 3.3 highlights the independent work done to simplify deployment of and introduce elasticity to *vOLAP* in cloud environments using cloud management software. In Chapter 4, a method of quorum-based fault tolerance for distributed aggregate systems is described and evaluated. Chapter 4 presents Aggregate Probabilistically Bounded Staleness (A-PBS), a tool for examining the impact stale or inconsistent data has on numerical accuracy of aggregate data stores, followed by A-PBS analysis of *vOLAP*. Finally, Chapter 6 provides a conclusion of the topics discussed in the thesis, and outlines possible future work.

Chapter 2

Background

2.1 Distributed and Cloud Computing

Cloud computing, generally speaking, refers to the use of provisioned or rented computer hardware accessible only over a network. Users who rent computer hardware or pay for services from a cloud provider pay only for the amount of time they use the hardware or service for. For distributed applications, this is very useful, as it allows new resources to be introduced to the system when demand or load increases. Similarly, resources can be deallocated when they are no longer necessary in order to reduce operating cost. This type of elasticity is a contributing factor to the widespread popularity of cloud computing.

According to [64], a cloud computing provider has five essential responsibilities. First, a customer must be able to unilaterally provision resources from the provider without the requirement of any human interaction between the customer and service provider, allowing resources to be allocated in an automated way. This ties in with a second characteristic, rapid elasticity, which allows for rapid scaling of resources at any time of day, with seemingly unlimited resources (from the perspective of the customer) available for provisioning. Another characteristic requires broad network access, which states that resources must have the capability to be accessed by a variety of devices (e.g., smartphone, workstation). From the provider's perspective, resources must be pooled using virtualization for dynamic allocation and deallocation to and from different customers. This is typically implemented through the use of a hypervisor or virtual-machine monitor [40], with one of the more popular being the Xen hypervisor [24, 67]. Finally, cloud systems must allow the user some way to measure the usage of provisioned resources over time.

There exist three prevailing models of cloud computing [64]. Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS provides users with unrestricted access to virtual machines and other resources. It is

the responsibility of the user to manage the operating system, software packages and other details relating to the system's core infrastructure. PaaS attempts to simplify IaaS by providing customers a platform of preconfigured hardware and software to run their applications on. Using PaaS, the infrastructure is largely managed and configured by the cloud provider. In SaaS, customers access the cloud through a client that communicates with a cloud application. SaaS users are not responsible for the configuration of infrastructure or the software that runs on said infrastructure. Instead, SaaS users are simply clients of a cloud application, for example, users of an email service.

2.1.1 Cloud Providers

In the context of IaaS, the cloud provider's responsibility is to allow the developer of an application to provision compute, storage and network resources in a timely manner, allowing for an elastic increase of resources when user demand requires it. It is also the cloud provider's responsibility to ensure a level of security between resources. Since cloud compute resources are often virtualized, two clients of the same cloud provider may often end up using virtualized resources on the same server rack. Indeed, there has been much concern in recent literature regarding the security of cloud resources [54, 32, 63, 68].

2.1.2 Amazon Web Services

Amazon Web Services (AWS) is currently one of the most popular cloud providers, offering a variety of IaaS and PaaS services to developers. In the context of this paper, we are specifically interested in Amazon Elastic Compute Cloud (EC2) [2], Amazon's flagship IaaS offering. Like many other IaaS providers, Amazon EC2 allows customers to provision individual compute nodes, or *instances*, with varying hardware specifications. Several varieties of instance types are available and grouped into different classes depending on their hardware. For example, the compute optimized class of instance, referred to as C4, provides the highest grade of CPUs per dollar, while the memory optimized R3 class of instance are instead configured to have large amounts of memory. The user is charged on an hourly basis for each instance provisioned, where the cost of each instance is dependant on the class and hardware

of the node.

When a user provisions an Amazon EC2 instance, they must select an Amazon Machine Image (AMI) to use with the instance. An AMI is a snapshot of an instance's file system. When an instance is provisioned with an AMI, the instance launches with the same file system specified by the AMI. This is useful for automating the infrastructure configuration process for newly provisioned nodes, as one needs to only configure the operating system and software packages once and create a new AMI from that image. Amazon also provides AMIs of installations for a variety of Linux distributions. AMIs are stored in Amazon's Simple Storage System (S3) and persist after the original instance is deallocated.

EC2 also provides virtualized hard drives to the user in the form of Elastic Block Store (EBS) volumes. Users may provision EBS volumes much like EC2 instances and pay on an hourly basis per gigabyte used. Volumes come as magnetic or solid state drives and can be attached to at most one instance at a time, at which point they can be mounted and read from or written to. Like AMIs, the contents of an EBS volume can also be snapshotted and stored on S3.

2.1.3 Configuration Management

Managing the configurations of a large number of instances within a cloud environment can prove difficult. When one accounts for heterogeneity between instance hardware and function, as is often encountered in cloud environments, ensuring that each instance has the required software installed, the necessary programs running and the operating system sufficiently configured is not only difficult, but highly time-consuming. Configuration management software like SaltStack [10], Puppet [7] and Chef [1], attempt to automate the process of configuring and deploying multi-node environments. Typically, management configuration software takes a list of rules from the user, and automatically enforces them amongst a group of nodes. Configuration management software can also be used to automatically manage cloud resources, like compute instances or storage. The remainder of this section covers the SaltStack configuration management software in depth.

A distributed system running SaltStack is composed of at least one node running *salt-minion* daemon, and at least one node running the *salt-master* daemon. Using

a TCP connection between each minion and master, the master node is able to orchestrate minion nodes to run specific commands and to conform to specific system rules or states, which greatly simplifies deployment of distributed systems.

SaltStack commands may be initiated in two ways: through a set of command line programs, and through a Python API. In both cases, the format is essentially the same: a string representing the target nodes to execute the command on, the operation to execute on the target nodes, and any additional parameters to the function. For example, a command that uses SaltStack through the command line to run the POSIX standard `date` program on all minion nodes is as follows:

```
salt '*' cmd.run 'date'
```

And in Python:

```
import salt.client
local = salt.client.LocalClient()
local.cmd('*', 'cmd.run', ['date'])
```

The first word in the command, `salt`, is simply the command line program that communicates with the salt-master daemon to orchestrate the following command on the relevant minion nodes. The `'*'` specifies which minion nodes, by name, to run on. The name of each minion can be set through a SaltStack configuration file, but in this case, since the string contains only the wild card asterisk character, all minion nodes will be sent this command. Alternatively, a minion node's name can be used as the target specifier string to run a function only on that specific minion. The following word, `cmd.run`, tells SaltStack to execute the `run` function from the `cmd` package on the relevant minions. `run` is a useful built-in SaltStack function which runs the command specified in the following argument string, which in this case is simply `date`. The resulting output is simply the output of running the `date` program on each relevant minion:

```
worker_003:
    Wed Sep  9 18:05:13 UTC 2015
server_001:
```

```
    Wed Sep  9 18:05:11 UTC 2015
worker_001:
    Wed Sep  9 18:05:12 UTC 2015
worker_002:
    Wed Sep  9 18:05:12 UTC 2015
client_001:
    Wed Sep  9 18:05:12 UTC 2015
manager_001:
    Wed Sep  9 18:05:12 UTC 2015
```

SaltStack also provides utilities to ensure a correct system configuration across each minion node. In the form of `.sls` files, users can create YAML files which contain assertions of the current system state. For example, one might assert that specific files are present, that specific packages have been installed, or that specific services are running. These `.sls` files may be run on a series of minion nodes through the command line or Python interface. Any minions that fail an assertion will automatically attempt to rectify the issue. In the event a failed assertion cannot be rectified, SaltStack aborts the process, and logs the error. Running the following `.sls` file from [11] will ensure that the `vim` text editor program is installed, and that the same `vimrc` configuration file stored on the salt-master is also present on the minion at `/etc/vimrc` as well.

```
vim:
  pkg.installed: []

/etc/vimrc:
  file.managed:
    - source: salt://vimrc
    - mode: 644
    - user: root
    - group: root
```

`salt-cloud` is a core component of SaltStack which attempts to simplify the provisioning of resources on different cloud services like Amazon EC2 and OpenStack [6]. While both OpenStack and Amazon EC2 have command line utilities and Python

clients (novaclient [8] and boto [4] respectively) for creating and managing resources on their cloud services, `salt-cloud` implements a common language and wrapper around both, allowing for the same commands and scripts to work regardless of the cloud provider. After the user creates a provider configuration file containing the relevant cloud provider keys and location, users define node types in a YAML *profiles* file. For example, the following describes a node type called `worker`, to be provisioned by the `openstack` provider, of size `c8-30gb-380` (the naming convention for a node with 8 cores on this specific provider) using the host drive snapshot `my-snapshot`.

```
worker:
  provider: openstack
  size: c8-30gb-380
  image: my-snapshot
```

Any node type specified in the profiles file can then be provisioned using the `salt-cloud` command line program as follows:

```
salt-cloud -p worker worker_001
```

Where `-p worker` specifies to `salt-cloud` that the node type named `worker` is being used as our profile, and `worker_001` will be the name of our newly provisioned node. The same node can also be deallocated with the command:

```
salt-cloud -d worker_001
```

As a convenience, nodes can be allocated and released in groups using a *map* file. Map files specify groups of nodes by name and node type. For example, the following map file specifies three nodes under the node type `worker`, and two more under the node type `other`.

```
worker:
  - worker_001
  - worker_002
  - worker_002
```

```
other:
  - other_001
  - other_002
```

Referencing this file in a call to `salt-cloud` will create three nodes of type `worker` named `worker_001`, `worker_002`, `worker_003`, and two nodes of type `other` named `other_001` and `other_002`. The command used to allocate all nodes in a map file is:

```
salt-cloud -m map_file -P
```

Where the `-P` instructs `salt-cloud` to allocate each instance in parallel. To deallocate each node in the map file, `-P` can instead be replaced by a `-d`.

2.1.4 Distributed Coordination in the Cloud

For many cloud applications, there is often a need for a distributed synchronization or coordination mechanism between nodes in the system. For example, if a single structure is shared across multiple nodes in the system, it is important that the state of the structure remains the same across each node whenever the structure is modified locally on a single node. Such synchronization can be achieved with the use of a coordination management system.

ZooKeeper [52] is an open-source coordination management system designed to coordinate nodes and data within a distributed environment. Through a C or Java library, clients can communicate to individual ZooKeeper servers to read, write or create *znodes*, which contain in-memory data referenced in a hierarchical data tree similar to that found in a typical Linux file system. ZooKeeper supports FIFO execution of write requests, and provides strong ordering for write requests across all clients, allowing for the implementation of system synchronization primitives. In addition, ZooKeeper servers support replication across multiple compute nodes, allowing for high availability and fault tolerance.

2.2 OLTP and OLAP Data Stores

2.2.1 OLTP

Online Transaction Processing (OLTP) systems are a common type of data store. OLTP systems are primarily focused on storing and serving detailed transactional data [50]. For example, one might use an OLTP system to store a record of sales within an organization. Read operations within an OLTP are designed to quickly retrieve a small number of individual records. OLTP systems are therefore best used for answering simple queries which do not need to read from many records [30].

Recently, with the advent of inexpensive cloud computing resources, scalable OLTP systems have surged in popularity [58]. Such systems focus on horizontal scalability, and take advantage of cheap, pay by the hour, compute nodes provisioned through the cloud [28, 34, 57]. In doing so, these systems are able to distribute query and insert load across several "shared nothing" compute nodes, improving latency and throughput performance of the data store, while maintaining high availability.

A type of distributed OLTP store especially relevant to this thesis is the key-value store. Popularized in [34], key-value stores contain abstract data (or *values*) addressable by specific *keys*. Key-value stores typically use a hash function to convert keys to numeric location values [57, 34]. A client can add an item to a key-value store by sending a key-value pair to the system. The hash value of the key is computed to determine where the value should be stored. Consequently, clients can read an item from the store by sending a key to the system. Hashing the key results in the location of the associated value, which can then be accessed to return the associated value of the key back to the client. This model lends itself well to distributed environments; a set of nodes in a system can be assigned to contain a specific range of hashed keys and their associated values. Read and write operations only require a single node to be accessed, so the system's throughput capacity benefits greatly from additional nodes.

For many key-value stores, high availability is a large priority. A system that is highly available continues to operate in the event of failure of individual nodes within the system. Key-value stores often use quorum-based replication of data to ensure availability within the system. Section 2.3.1 discusses the topic of quorum-based replication, and how it can be used to increase availability, in more detail.

2.2.2 OLAP

Online analytical processing (OLAP) data stores, unlike OLTP stores, attempt to quickly present generally summarized, broadly scoped information to decision makers (for example, managers and executives) in order to allow them to make informed decisions [30]. A typical OLAP query often involves operating on several thousands or millions of individual points in the system.

In general, OLAP data stores can be viewed on an abstract level as a large, multi-dimensional data cube [77, 46]. A data cube is composed of several dimensions, where each dimension corresponds to a specific property or metric that is deemed important for making OLAP supported decisions [50]. For example, time, location and product could be three dimensions for a data cube representing sales at a store. A hierarchy is associated with each dimension, which allows users to easily query specific segments of the data cube. Insert operations in an OLAP system write numerical *measure* values to specific hierarchical points within the cube. Using the dimension hierarchies, query operations retrieve measure values within a specified area of the cube, and combine the measure values with a provided binary associative aggregation function (for example *sum*, *mean* or *max*) to yield a single value.

Using queries, users can summarize or eliminate specific hierarchical levels or dimensions to generate a summarized view of a specific part of the cube, called a cuboid. Each cuboid contains an aggregation of the measure values in the associated multi-dimensional space. An entire data cube can be summarized with a set of cuboids.

Clients often interact with an OLAP system through a set of data cube operators. The set of data cube operators most commonly seen in OLAP literature are roll-up, drill-down, slice and dice [76, 18, 47, 50]. Rolling-up a data cube essentially consolidates hierarchical levels of a dimension into a single parent level. As an example, consider a data cube where one dimension represents time. This particular dimension is broken up into 365 cuboids, representing revenue for each day in a specific year. An analyst may not want that level of detail, and decides to roll up the time dimension of the cube to the next level in the hierarchy, the month. A set of cuboids is generated where there are now only 12 cuboids in the time dimension instead of 365.

Drill-down operates in a somewhat opposite manner to roll-up. Instead of summarizing data, drill-down splits one or more cuboids into several cuboids. While roll-up

steps up a level of the dimension hierarchy (say, day to month), drill-down steps down a level in the hierarchy (month to day). Continuing from the previous example, if the analyst decides they would now like to look at daily revenue once again, they could perform a drill-down operation on the time dimension, generating a new set of cuboids with 365 cuboids for each day in the year.

Slicing allows a user to remove specific elements of the hierarchy within a dimension from the cube. A dice operation is simply multiple slice operations across multiple dimensions. An analyst wanting to determine the revenue during the first quarter of the year may do so by slicing all months other than January, February and March.

Often, data cubes in OLAP systems are constructed statically, in that they compute aggregations for a wide range of queries in advance [50]. In doing so, such systems are able to quickly answer complex queries, but cannot support real-time insertion of new points. Recently, using the same affordable cloud technology that spurred growth of key-value systems, real-time OLAP systems, which use dynamically changing data cubes to support real-time inserts and queries, have been presented [55, 81]. In Chapter 3, we discuss a real-time distributed OLAP system.

2.3 Distributed Consistency

2.3.1 Quorum-based Replication

To solve the problem of availability, and to minimize the time a data store is unavailable, data store designers often turn to replication [72, 27, 48]. Replication is the act of storing data on multiple independent nodes in such a way that if one node becomes inaccessible, there exist several other functioning nodes that contain the same data. In doing so, in the event a node in the system crashes, the system can use other replicated nodes as a failover. Replication can also serve to further scale-out or load-balance a system, as read traffic can be split across replicas. Replication is also important for data stores which have users in significantly different locations in the world, as reading or writing to a node across the world is likely to have unacceptable levels of latency. Having replicas spread out across the world can address this problem as well.

While introducing replication to a system addresses many issues important for data accessibility, it also introduces new problems. Now that a system has, say N

replicas of its data distributed on different nodes, the data store's existing algorithms for reading and writing need to be modified. Keeping all replicas up to date is critical to ensure reads are accurate. As write operations now must write to multiple replicas, each replica is guaranteed to have some period of time where its contents differ from the other replicas. This desynchronization of data complicates read operations, as now measures must be taken to ensure the result of the read is up to date.

One widely popular method of approaching these problems is to implement a quorum consensus algorithm [51, 43, 44]. Under this scheme, two new variables are introduced to the system, W, the *write quorum* and R, the *read quorum*. Under a write operation, all N replicas must be sent the write. After W replica responses indicate the write was committed at W replicas, the write operation from the client's perspective is completed. Note however that in the background, writes for the remaining $N - W$ replicas are in progress. Greater values of W increase the chance of each client reading its own writes, but does not improve the read accuracy of writes sent from different clients. Under a read operation, all N replicas are queried and, like quorum-based writes, R replica responses are waited for. Once the R responses have returned, the most up to date response is selected, using some means of comparison (in key-value stores, this is usually a simple timestamp representing the time of last write).

Figure 2.1 presents a graphical representation of a read in a quorum system. The client C begins by sending a read request to each of the N replicas, represented here as B_1 through B_3. Once the client receives $R = 2$ replica replies, represented as the dotted lines, the client selects the most up to date response, completing the read operation. The remaining $N - R$, or 1 remaining response is received at a later time and ignored by the client. Figure 2.1 could also be used as an example of a write operation in a quorum system. The write is sent to all N replicas and, after the client receives $W = 2$ responses, the write is considered complete.

However, implementing these quorum rules is not enough to guarantee up to date reads. Values of N, W and R must be configured in such a way to ensure that at least one of the R read responses contains the most up to date version of the data being queried [43, 51]. For example, consider the case where `[N=3, W=1, R=1]`. Under these parameters, all writes complete once the data has been committed to one replica, while all reads complete when the client receives its first response from a replica. In

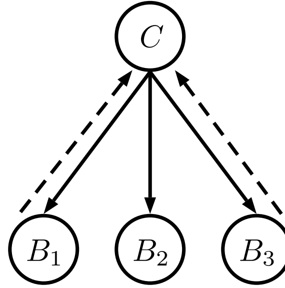


Figure 2.1: An example of a read operation using quorum rules. C corresponds to the client, while B_1, B_2 and B_3 refer to the nodes containing the replicated data relevant to the query. Solid lines correspond to read requests, and dotted lines correspond to responses to the client.

other words, a write followed by a read on the same point will only be up to date if the first replica to reply to the query was the first replica to reply to the write. In this case, reads are not guaranteed to be up to date.

A well known property of quorum consensus systems is that reads are guaranteed to always be up to date and consistent if $W + R > N$ [43]. Under this condition, for any read/write pair, there is always at least one more read or write reply than the number of replicas. It then follows that there must be at least one replica that has replied both to the write as well as the read. Therefore, if W, R and N are configured such that $W + R > N$, each read is guaranteed to read the latest write. However, it is also well known that large W, R and N values can negatively impact several performance characteristics of a data store [78], specifically availability and latency.

2.3.2 Types of Consistency

In distributed highly available data stores, data consistency is a common problem. As data stores seeking to improve their availability turn to storing copies of existing data on different nodes to improve system resiliency and minimize downtime, they introduce the possibility of said copies desynchronizing. Since writes to the system now need to be written to multiple copies instead of one, a list of factors (for example, varying network latencies across nodes, varying node workloads) result in each write committing at a different time at each copy of the data. This is best summed up with the CAP theorem [45], which states that no distributed data store can ever simultaneously provide strong *c*onsistency, *a*vailability and *p*artition-tolerance guarantees. Consistency

requires each query or read operation to return the correct result. Availability requires that every operation eventually completes. Partition-tolerance requires that the system continues to function within an unreliable network (e.g. some nodes may not be able to communicate with others). In practice, systems must choose between poor availability or poor consistency.

In general, we refer to a system as consistent if all reads reflect the latest write. However, this definition is somewhat strict. [73] compiles several popular definitions of weaker forms of consistency which are applicable mostly to distributed data stores. Here we present a selection of the definitions described in [73] and other popular literature.

Strong consistency guarantees the system offers the best possible consistency at all times. Specifically, under strong consistency, all reads are guaranteed to use the values of the latest writes to the data store. Since this guarantees perfect consistency, the CAP theorem implies that any system that offers strong consistency will have poor availability, partitioning, or both. Indeed, Cassandra [57], a popular key-value store, under the $W + R > N$ quorum rules guarantees strong consistency, but offers poorer availability when compared with less restrictive quorum rules, as reads and writes begin to fail when $N - R$ or $N - W$ replicas have died.

Read my writes guarantees strong consistency, but only for the writes initiated on the same client as the reader.

The *monotonic reads* guarantee requires that, on each client, once a point has been read by a query, any subsequent reads of the same point must read the same or more recent write as the previous read. Essentially, consistency cannot degrade over time.

Eventual consistency, an especially important type of consistency described in [78], guarantees that all writes sent to the system are eventually readable. This is the most lenient guarantee of all, but is typically the least sensitive to partitioning and can provide high availability. Eventual consistency also often offers excellent read and write latency.

A quorum system that obeys *k-regular consistency semantics* [19] guarantees that, provided the read and write quorums do not overlap, the result from one of the last k completed writes is returned. If there is overlap between the read and write quorums, either the result from the last k completed writes is returned, or the result from a

more recent write which has not yet completed on W replicas is returned.

A system that obeys *bounded staleness* has the view that a bounded amount of staleness (or, inconsistency) in a system is acceptable. k-regular consistency semantics is a type of bounded staleness that allows for a bounded amount of staleness with respect to the versioning or ordering of writes.

A final consistency guarantee covered in this thesis, described in [21, 22, 23], is probabilistically bounded staleness (PBS). PBS is similar to bounded staleness, in that they both provide guarantees based on version bounds. However, while bounded staleness is strict and guarantees every query obeys its bound, PBS requires each query to meet its bound with a specific probability. PBS is covered in depth in the following section.

2.4 PBS for Key-value Stores

In recent years, there has been increased interest in providing probabilistic consistency guarantees for quorum systems, with several papers published on the subject [62, 22, 21, 23, 61, 17]. In [22], the authors examine eventual consistency from the perspective of quorum-based key-value stores and present a probabilistic metric of staleness for partial quorums ($W + R \leq N$). First a closed form method for determining the probability of no intersection between randomly selected read and write quorums, p_s, is given:

$$p_s = \left(\frac{\binom{N-W}{R}}{\binom{N}{R}} \right) \tag{2.1}$$

In this equation, the numerator represents the number of different ways R replicas can be selected from the set of replicas that do not contain the most recent write $(N - W)$, and the denominator represents the number of ways R replicas can be selected from the total number of replicas N. In other words, the number of quorum selections that yield no intersection between reads and writes, and the number of possible quorum selections in total.

With this in mind, PBS k-staleness is introduced whose definition is included verbatim from [22] below:

Definition 1 (PBS k-staleness) *A quorum system obeys PBS k-staleness consistency if, with probability $1 - p_{sk}$, at least one value in any read quorum has been committed within k versions of the latest committed version when the read begins.*

In other words, a system with PBS k-staleness has, given a stream of k writes to the same key, probability $1 - p_{sk}$ of reading the kth most recent write or newer.

Following this, a closed form for determining the value of p_{sk} is derived. Since time is not modelled in this view of staleness, each write is assumed to be written on only W randomly chosen replicas; eventual consistency is not represented in this specific model. Because of this, p_{sk} becomes the probability of the read quorum not intersecting with any of the write quorums, which can be determined using equation 2.1, for k versions:

$$p_{sk} = \left(\frac{\binom{N-W}{R}}{\binom{N}{R}} \right)^k \tag{2.2}$$

Additionally, a second type of bounded staleness is presented in the paper. PBS t-visibility is a time-based metric and therefore reflects how long it takes for writes to become eventually consistent. The definition as seen in [22] is presented verbatim:

Definition 2 (PBS t-visibility) *A quorum system obeys PBS t-visibility consistency if, with probability $1 - p_{st}$, any read quorum started at least t units of time after a write commits returns at least one value that is at least as recent as that write.*

In PBS t-visibility, t encodes a lower bound on the range of time between the committal of the latest write quorum (when W write replies have been received) and the time of initiation of the read quorum. Because t-visibility is strongly influenced by the read and write latencies of a system, and not the quorum configuration as k-staleness is, it is difficult to construct a closed form analysis of t-visibility.

To address this, a method for examining t-visibility in the context of distributed key-value stores is discussed. In such stores, inconsistency is largely caused by message reordering [34]. Therefore, in order to model inconsistency in a key-value store, message reordering must also be modelled. Using the WARS model, writes and reads may be modelled using system-specific distributions for replica write latencies `W`, write replies or acknowledgements `A`, replica reads `R`, and replica read responses `S`.

For example, a write followed by a read, using WARS, would be modelled as follows. A coordinator begins by sending a write request to N replicas. The distribution `W` is sampled N times to determine the time taken for each replica to receive the write request and to complete the write locally. `A` is sampled for each replica, and summed with each replica's sampled value of `W` to obtain the time at which the coordinator has received the quorum write reply from each replica. Of these, the Wth smallest time is the time at which the coordinator has received W replies from each replica, satisfying the quorum rules for a write. Reads behave similarly, except the distribution `R` is used to determine the time taken for a read request to arrive and complete locally on a replica, and `S` is the time taken for a read reply to arrive at the coordinator.

As is illustrated in Figure 2.2, t-visibility can be evaluated using WARS by modelling a single write, waiting t units of time after W write acknowledgements, and modelling a read. If, of the R replicas which responded earliest, at least one completed their read `R` after the write `W` completed, then this specific read is said to contain the most recent write.

This sets the stage for a relatively simple method of determining the value of a system's value of p_{st} using event-based simulation. Simply construct a Monte Carlo simulation, where each trial is a WARS style write, followed by a delay of t units of time, followed by a read. As mentioned above and illustrated in Figure 2.2, this can be easily computed by sampling from `W`, `A`, `R` and `S` and checking if the write and read has been reordered in all of the read replies. If enough trials are simulated, the value of p_{st} is estimated by the ratio of trials which were not consistent over the total number of trials.

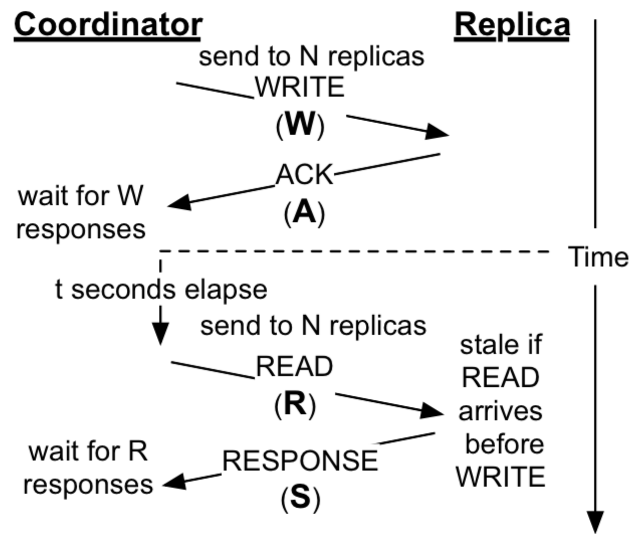


Figure 2.2: The WARS model for Dynamo-style data stores, from [22].

Chapter 3

vOLAP

In this chapter, we describe *VelocityOLAP* (*vOLAP*) [55], a real-time, scalable OLAP system that enables up-to-date OLAP querying for high velocity data with multiple dimension hierarchies and relies on cost-efficient horizontal scaling using commodity hardware (standard cloud instances). A distinguishing feature of *vOLAP* is that it exploits the dimension hierarchies to improve performance. *vOLAP* has been tested on TPC-DS [12] test data, which include the dimension hierarchies shown in Figure 3.1. Using 20 worker instances on the Amazon EC2 cloud for a database size of 1 billion items, results show that *vOLAP* is able to ingest new data items at a rate of over 400,000 items per second, and process streams of interspersed inserts and OLAP queries in real-time at approximately 50,000 inserts and 15,000 OLAP queries per second. These experiments include a wide range of queries ranging from small queries, to average size queries that need to aggregate several hundred million data items, up to queries that need to aggregate nearly the entire database.

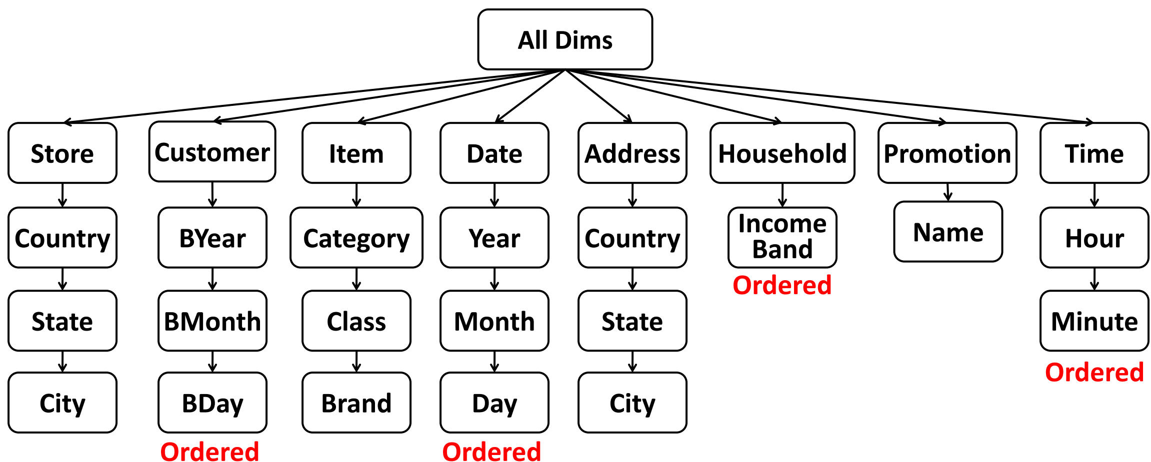


Figure 3.1: Dimension hierarchy for TPC-DS [12] benchmark data

An important feature of *vOLAP* is that it supports horizontal scaling on commodity hardware (standard cloud instances). For example, experiments use only standard

Amazon EC2 `c3.4xlarge` worker instances, and scale up the number of such nodes to match the database size and query workload. This is more cost-efficient than other high-performance OLAP systems such as SAP HANA, which rely on vertical scaling, i.e. the use of a small number of special-purpose large compute nodes, or even special-purpose hardware such as a IBM Netezza data warehouse appliance [15]. In Section 3.3, we discuss *vOLAP*'s cloud deployment infrastructure, and how it enables additional nodes to be added to *vOLAP* elastically.

vOLAP is a fully distributed, cloud-based system that uses as a building block the multi-threaded PDC-tree [36], a spatial tree designed for handling OLAP-style insert and query operations in parallel. Data is partitioned into subsets stored in PDC-trees on *worker* nodes of the cloud environment. Each PDC-tree supports multi-threaded insert and query processing, with minimum locking. As is increasingly typical for current high-performance OLAP systems, *vOLAP* is an in-memory system and supports ingestion of new data but no deletion. Multiple server nodes handle the incoming streams of new data inserts and OLAP queries, and route them to the appropriate workers. ZooKeeper is used for managing global state information. A *manager* background process monitors the load status of the system and provides instructions to worker nodes for global real-time load balancing. This load balancing is fully automatic and adjusts dynamically to the data distribution, which can change significantly over time due to the high velocity of incoming new data. Unlike other distributed OLAP systems, *vOLAP* does not use a special partitioning dimension that needs to be manually configured.

Both server and worker nodes in *vOLAP* can be added or removed as necessary to adapt to the current workload. Unlike other systems, no single node acts as a performance bottleneck or failure point for the entire system. Besides this, *vOLAP* introduces novel index and worker data structures, a fully decentralized design with support for multiple servers and improved load balancing algorithms for a fully decentralized environment.

The remainder of this chapter is organized as follows: Section 3.1 discusses recent related work. Section 3.2 describes *vOLAP*'s design, including network architecture, synchronization, data representation, and load balancing. The discussion and contributions within this section are the result of group work with several other students

and researchers. The author's independent contribution to *vOLAP* is described in Section 3.3, wherein we describe *vOLAP*'s underlying cloud infrastructure which allows for elastic provisioning of resources and simple deployment of *vOLAP* to cloud environments. Section 3.4 presents experimental results that demonstrate the performance of a prototype implementation on an Amazon EC2 cloud.

3.1 Related Work

Many published systems store and query large data sets in cloud environments. Hadoop [5] and its file system HDFS are popular examples, with applications typically built on MapReduce [33]. However, these systems are not designed for *real-time* operation. Instead, they are based on batch processing or "quasi real-time" operation [25, 53, 69, 70]. The situation is similar for Hive [74], HadoopDB [16], BigTable [29], BigQuery [3], and Dremel [65]. To overcome the batch processing in Hadoop based systems, Storm [13] introduced a distributed computing model that processes in-flight Twitter data. However, Storm assumes small, compact Twitter style data packets that can quickly migrate between different computing resources. This is not the case for large data warehouses. Several more recent cloud-based OLAP systems [26, 31, 49, 60] are also based on MapReduce and do not support full real-time operation. For peer-to-peer networks, related work includes distributed methods for querying concept hierarchies [39, 20, 38, 71]. However, none of these methods provide *real-time* OLAP functionality. There are various publications on distributed B-trees for cloud platforms [80]; however, these only support 1-dimensional indices which are insufficient for OLAP queries. There have been efforts to build distributed multi-dimensional indices on cloud platforms based on R-trees or related multi-dimensional tree structures [79, 82, 56]. However, these methods do not support dimension hierarchies, which are essential for OLAP queries. The systems closest to *vOLAP* are Druid [81], SAP HANA [42], IBM Netezza data warehouse appliance [15] and CR-OLAP [35]. The remainder of this section will discuss these in more detail.

Druid [81] is an open-source, distributed, scalable, OLAP store designed for real-time exploratory queries on large quantities of transactional events. Druid is specialized to operate on data items that have timestamps, such as network event logs. In particular, it partitions data based on these timestamps and queries are expected

to apply to a particular range of time. This is not applicable to general OLAP, where all dimensions have equal importance.

SAP HANA [42] is a real-time in-memory database system that also supports OLAP queries. SAP HANA relies mainly on *vertical* scaling. A basic HANA installation uses a single, special-purpose, very large multi-core compute node. A limited scale-out version for multiple compute nodes is available, using a distributed file system that provides a single shared data view to all compute nodes. Horizontal scalability is restricted, however, because the system has a single master node for maintaining the shared data view, and this single master node becomes a bottleneck as system size increases.

The IBM Netezza data warehouse appliance [15] relies on special-purpose FPGA boards that provide a hardware implementation of OLAP functionality.

The precursor to *vOLAP*, *CR-OLAP* [35, 55], is similar to HANA in that it is also a centralized system with a single master node. As in HANA, this becomes a bottleneck in larger systems. CR-OLAP stores data in memory on worker nodes. Similar to *vOLAP*, CR-OLAP uses the PDC tree [36] as a building block, but as one large tree, where the top few levels are stored on the master node and the induced subtrees are stored on worker nodes. Unlike *vOLAP*, this design does not allow for a distributed index. Similar to HANA, the single master node in CR-OLAP becomes a bottleneck for larger systems and restricts horizontal scalability. In *vOLAP*, the single master node is replaced by a distributed system of global and local system images. *vOLAP* introduces novel index and worker data structures, a fully decentralized design with support for multiple servers and improved load balancing algorithms for a fully decentralized environment.

3.2 vOLAP Architecture

Consider a d-dimensional data warehouse D with N_{items} data items and d dimension hierarchies such as those shown in Figure 3.1 on page 26. *vOLAP* processes multiple input streams of interspersed insert and OLAP query operations on D. Each OLAP query specifies, for each dimension, either a single value, range of values at any level of the respective dimension hierarchy, or a symbol "*", which matches the entire range for that dimension. The OLAP query result is the aggregate of the specified

items in D. The coverage of an OLAP query is the percentage of D that needs to be aggregated.

3.2.1 Architecture Overview

vOLAP is a cloud-based architecture consisting of j servers $S_1, \ldots, S_j$ for handling client requests (inserts and queries); p workers $W_1, \ldots, W_p$ for storing data and performing operations requested by servers; a ZooKeeper cluster for maintaining global system state [52]; and a manager background process for analyzing global state and initiating load balancing operations. Workers and servers are multi-core machines which execute up to k parallel threads. As is typical for current high-performance databases, all data is kept in main memory. With increasing database size and/or changing network topology, *vOLAP* reorganizes the data to make the best use of currently available resources. *vOLAP* is elastic in that more workers and servers can be added if necessary. Figure 3.2 illustrates the distributed architecture, and a summary of the system parameters is given in Table 3.1.

Workers are used for storing data and processing OLAP operations. The global data set D is partitioned into data subsets $D_1, \ldots, D_m$. Each subset D_i has a bounding box B_i, which is a spatial region containing D_i, represented by either a MBR (one box) or MDS (multiple boxes) [41]. Bounding boxes may overlap. Each worker typically stores multiple subsets. The number of initial subsets and maximum size of each subset are configurable parameters. Smaller subsets are easier to load-balance, but are slow to answer queries, as each subset must be queried individually. Conversely, large subsets are difficult to load-balance, but can offer faster query times.

Servers receive OLAP operations from clients, determine the subsets relevant to each operation, and forward the operations to the worker(s) responsible for those subsets. Once the workers respond, the server reports the relevant results to the originating client.

Servers, workers, and the manager background process communicate using ZeroMQ [14]. It provides socket-based APIs for inter-process and inter-thread communication. In the prototype implementation, every working thread has its own local socket to receive messages and a network socket to send messages without locking. A separate network socket receives incoming requests, which are dynamically load-balanced

between the local thread sockets to ensure a high degree of parallelism.

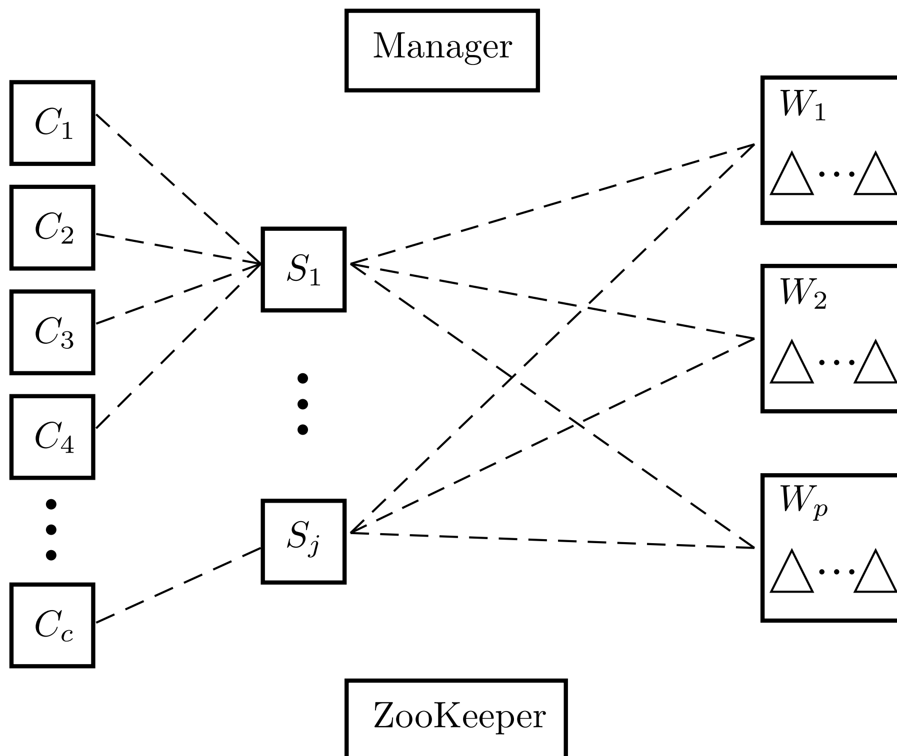


Figure 3.2: System overview

Parameter	Description
N_{items}	Total number of data items
m	Number of data subsets
p	Number of workers
j	Number of servers

Table 3.1: System parameters

3.2.2 System Image

The system image represents the global system state, and is stored in ZooKeeper [52], a fault tolerant distributed coordination service. The image contains the global information required by servers and the manager, including: lists of the current workers and servers, configuration parameters, and for each subset D_i, its size $|D_i|$, bounding box B_i, and the address of the worker where it is located.

Each server S_k maintains a local image I_k which is used to perform inserts and queries. The local image serves as a cache to prevent ZooKeeper from becoming a bottleneck. If the local image is changed by an insert operation, the server updates the global image in ZooKeeper. This update rate is configurable; in our experiments, servers update ZooKeeper every 3 seconds if necessary. Since changes take time to reach other servers, the local image of each server may be outdated. Servers monitor changes via ZooKeeper's watch facility, so servers are notified of a change when it occurs, without wasteful polling. Workers update subset statistics in ZooKeeper periodically as well. This information is used by the manager to plan load balancing operations.

Given a query, server S_k computes from I_k the address of the target worker(s) that contain data covered by the query. Given an insert, server S_k uses I_k to determine the subset D_i that the data should be inserted into and the corresponding worker address. The size $|D_i|$ is updated accordingly, as is the bounding box B_i if it has changed.

The performance of the local image is crucial to performance. In particular, it must quickly find the bounding boxes relevant to a given query, and choose where to place inserts such that overlap between bounding boxes is minimized.

3.2.3 Load Balancing

Effective load balancing is crucial for the scalability of distributed systems. When the workload of *vOLAP* is unevenly partitioned among its resources, some nodes may go underutilized while the remainder struggles to pick up the slack. This has a negative impact on throughput, response time, and stability. This tends to get further compounded as *vOLAP* scales up in size. However, the load balancing operations themselves can also incur significant costs due to the overhead of moving potentially large subsets of data over the network. High-performance requires a load balancing scheme which offers a good trade-off between load balancing overhead and effectiveness.

In *vOLAP*, the manager initiates load balancing operations. The manager periodically analyzes the system state stored in ZooKeeper and decides on suitable load balancing operations. It then initiates these operations, coordinating the necessary actions between workers and servers. For example, the manager may identify a worker that is overloaded and about to run out of memory. Based on this, it sends messages

to workers instructing them to perform splits and/or migrations via the data structure operations SplitQuery, Split, SerializeSubset and DeserializeSubset discussed below. Note that the manager is a background process that only initiates operations which are then executed by the workers and servers. The manager process is therefore not a bottleneck for query performance, and it can reside anywhere in the system.

A key to achieving high-performance in *vOLAP* is the design of migration protocols that allow the system to move subsets transparently from one worker to another while the system continues to service both inserts and OLAP query requests. In addition to being integral to achieving high-performance, dynamic load balancing also permits *vOLAP* to add, remove or replace workers dynamically in order to maintain system performance in the face of changing OLAP loads and data sizes.

In order to support load balancing, the subset data structures provide the following operations:

- An operation SplitQuery(D_i, B_i) returning a hyperplane h that partitions D_i into two subsets D_i^1 and D_i^2 with bounding boxes B_i^1 and B_i^2, respectively, such that D_i^1 and D_i^2 are of approximately equal size. Note that this operation does not perform an actual split.

- An operation Split(D_i, B_i, h) returning a tuple $(D_i^1, B_i^1, D_i^2, B_i^2)$, where D_i is partitioned into D_i^1 and D_i^2 with bounding boxes B_i^1 and B_i^2, respectively, such that D_i^1 and D_i^2 are spatially separated by the hyperplane h. Note that the split operation does not interrupt the query stream and maintains strong serialization of operations on D_i.

- An operation SerializeSubset(D_i), which returns a flat binary blob b containing the data in D_i (suitable for network transmission), and a corresponding operation DeserializeSubset(b) which builds the data structure from such a blob.

The SplitQuery, Split, SerializeSubset, and DeserializeSubset operations are used to support load balancing and data migration. A subset D_i stored on a worker W_s (the source worker) can be migrated to another worker W_d (the destination worker) if, for example, W_s is running out of memory or W_d is a new worker allocated for spreading the load. D_i can also be split if the load balancer requires smaller subsets

for migration. Each worker W_k stores a *mapping table* T_j to handle operations while a split is in progress. If a subset D_i is split into subsets D_i^1 and D_i^2, then T_j stores an entry with key D_i and value pointing to the two data structures for D_i^1 and D_i^2. To ensure consistency during splits and migrations, an insertion queue is created for the relevant subset. During the operation, inserts for the subset are inserted into the insertion queue to prevent the subset from growing continuously while it is being serialized. The insertion queue uses the same data structure as subsets, and is queried as well for any incoming queries that touch the subset. When the operation is finished, the insertion queue is drained into the subset and destroyed.

When a manager observes that the number of points within a subset has exceeded a configurable "soft limit" value, it will initiate a split operation on said subset to yield two smaller subsets below the soft limit. Similarly, when a manager observes that a migration of a specific subset from one worker to another can reduce the difference in the number of total points stored on the two workers by a configurable "difference threshold", the manager considers the subset and workers as a migration candidate. Once the manager considers all other migration candidates, the manager initiates a migration operation using the candidate that yields the greatest reduction in the difference of the number points between two workers.

3.3 Elastic *vOLAP* on a Managed Cloud

In this section, we describe how *vOLAP* is deployed on cloud infrastructure and how elastic provisioning of resources is supported. While the work described in this section is primarily of an engineering nature, it is fundamental to support elastic provisioning, as well as quorum-based replication described in Chapter 4.

3.3.1 SaltStack-managed *vOLAP* Deployment

Earlier versions of *vOLAP*'s deployment architecture were based on `mpirun`, a small Linux program, typically used on clusters of nodes with static IPs, to launch processes from a single node onto a cluster of nodes. While `mpirun` works well in clusters, where resources are not elastic and IP addresses are more or less static, `mpirun` is ill-suited for use within a cloud environment. For example, using `mpirun`, the process to provision and launch *vOLAP* on a typical IaaS platform is as follows:

- Manually provision the desired number of nodes from the cloud provider's web client.
- Manually provision storage devices using snapshotted TPC-DS data.
- Manually attach each storage device to the relevant client nodes using the cloud provider's web client.
- Log into each client and mount the storage devices.
- Log into the master node and copy the IP of all other nodes into a host file.
- Run a script that uses `mpirun` and the host file to launch the appropriate *vOLAP* binary on each node.

Under this scheme, deploying *vOLAP* is a very involved process, and requires constant manual interaction with the cloud provider's web client. Since each instance is not associated with a name or string identifier, determining what type of role (worker, server, manager or client) each individual node is playing is confusing, and is determined solely by the order of the node's IP in the host file. Additionally, in order to scale the system up or down without installing additional software, nodes must either be provisioned or removed from the web client; *vOLAP* has no facilities available to it to manage elastic resources.

To simplify and automate *vOLAP*'s deployment process, SaltStack can be used to provision and manage nodes within the system. After provisioning a single salt-master instance, all other instances can be provisioned and configured automatically via `salt-cloud`. Once all nodes in the system have been provisioned, SaltStack can selectively run the appropriate *vOLAP* binaries based on the assigned name of the node (worker, server, client, etc.). The same process used to provision nodes can also be used to support automatic provisioning of additional resources, allowing *vOLAP* to scale up or down according to load.

Figure 3.3 illustrates the process of deploying *vOLAP* on the cloud using SaltStack. First (1), the salt-master node is provisioned and is accessed via `ssh` from a workstation. (2), the salt-master can then provision and configure all *vOLAP* nodes in a single step. Finally (3), each *vOLAP* node is instructed by the salt-master to run the appropriate binary associated with the node's name, completing the deployment process.

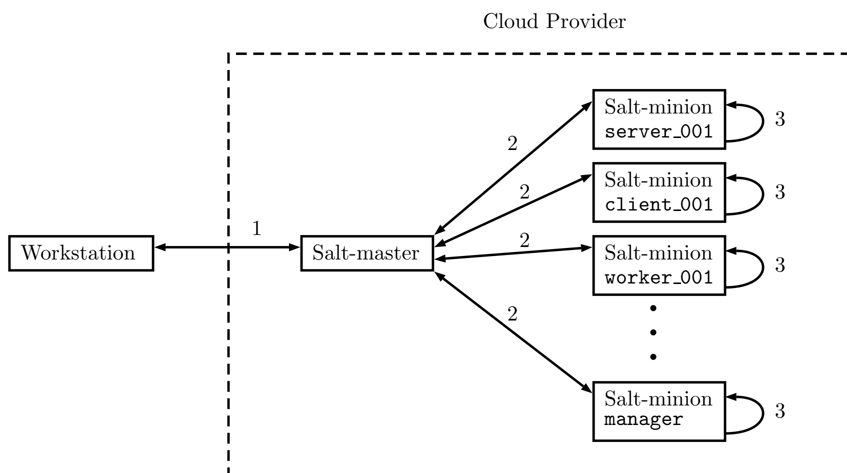


Figure 3.3: *vOLAP* deployment using SaltStack

The following Sections 3.3.2 and 3.3.3 describe the process of deploying *vOLAP* on Amazon EC2 and OpenStack in more detail. Section 3.3.4 describes how SaltStack is used to support elastic resource provisioning within *vOLAP*.

3.3.2 Amazon EC2 Deployment

To improve *vOLAP*'s cloud deployment process and to enable *vOLAP* to leverage the elastic resources available, SaltStack was used to manage all of *vOLAP*'s interactions with the cloud provider. First, SaltStack was installed and configured for salt-master and salt-minion nodes and snapshotted to two salt-master and salt-minion AMIs. The salt-minion AMI is relatively small, and contains only the libraries needed for *vOLAP* to run and the salt-minion software. The salt-master AMI is configured to contain a providers file, containing the access keys for SaltStack to interact with the cloud provider, and a profiles file. In the profiles file, the instance type and AMI are specified for worker, server, client and manager nodes. For each node type, we use the same salt-minion AMI. For client nodes, we specify in the profiles file that client nodes must launch with an EBS drive using snapshotted TPC-DS data. With this, all "client" type nodes launched by SaltStack will automatically provision and mount their own EBS drives containing TPC-DS data. Client nodes not provisioned through SaltStack or outside of the cloud can still communicate with servers as normal.

With both salt-master and salt-minion AMIs snapshotted, a salt-master node may be provisioned using the cloud provider's web client. The salt-master node is responsible

for communicating with the cloud provider for elastic resource management, as well as coordinating jobs among salt-minions. Once a salt-master node is provisioned, the `salt-cloud` program can be run with a map file containing a list of named resources to provision nodes for *vOLAP*. Once `salt-cloud` completes, *vOLAP* binaries can be copied over from the salt-master to each minion by using a simple state file, similar to the one shown in Section 2.1.3. Once the *vOLAP* binaries have been copied to each minion, *vOLAP* can be started by running a string of SaltStack commands, contained within a Python script.

The script itself uses the SaltStack Python API to selectively issue jobs to minions by using their instance name specified by the map file. For example, if the naming convention for nodes in the map file is the node's "role" followed by a unique integer (e.g. worker_1, worker_2, server_3, client_4), a Python script on the salt-master can easily issue a job to all nodes that are workers, servers, etc. using a selective `cmd.run` command. For example, in order to launch the worker binary on all worker nodes, the Python equivalent of `salt 'worker_*' cmd.run './worker_binary'` can be used.

To issue multiple jobs to multiple minions asynchronously, the `run_job` function is used. This allows for jobs to be easily issued while others are still in progress. However, in doing this, output from `stdout` and `stderr` are no longer published to the synchronous Python script. To remedy this, all asynchronously issued jobs use a Python wrapper script which acts as a communication layer between the salt-minions and the salt-master. The wrapper script launches the binary specified in its arguments, and watches for any output from the invoked program. Whenever any output is received, the wrapper script sends a notification to the salt-master using the SaltStack Python API. The script on the salt-master picks up on any notifications and prints them, along with the name of the node, to standard output. The wrapper script is also used to notify the salt-master whenever a program terminates. This is useful for testing, as scripts can be instructed to wait until certain jobs have exited without error before proceeding, or to abort if any jobs fail.

3.3.3 OpenStack Deployment

Since `salt-cloud` has built-in support for OpenStack, setting up the SaltStack environment within OpenStack is largely the same as on Amazon EC2. Initial snapshots

for salt-master and salt-minion nodes are generated through OpenStack's web client as AMIs would be on EC2, and volume snapshots for TPC-DS data are taken in the same way EBS snapshots would be. After initial set-up, provisioning and interacting with nodes on OpenStack through SaltStack is almost indistinguishable from Amazon EC2.

3.3.4 Automated Scaling

Using SaltStack, scaling *vOLAP* up or down is relatively simple. As mentioned in the previous section, the `salt-cloud` program can be used to provision or deallocate nodes by name or map file. For automatically scaling up *vOLAP*, a Python script is used which creates a map file containing a number of new workers and calls `salt-cloud` with the path to the map file. This initiates the provisioning process. Once the nodes are provisioned, the *vOLAP* binaries are copied over using a state file, and `run_job` is used to start the worker binary on each of the newly provisioned nodes. Scaling down simply requires another call to `salt-cloud` on the previous map file, with the deletion flag `-d`.

Once the new nodes are provisioned and running the worker binary, the existing load balancing mechanism within *vOLAP* takes place. The new workers register themselves to ZooKeeper, alerting the manager and servers that additional workers have been added to the system. This causes the manager to detect a load imbalance between workers (as the new workers are empty), resulting in the manager scheduling migration and split operations to rebalance the system.

3.4 Experimental Evaluation

Using the SaltStack environment on Amazon EC2, the performance of *vOLAP* was evaluated with respect to the system parameters shown in Table 3.1, the workload mix (percentage of inserts in the operation stream; e.g. workload mix 25% is 25% inserts and 75% OLAP queries), and query coverage (percentage of the database that needs to be aggregated for an OLAP query). The TPC-DS decision support data set from the Transaction Processing Council was used, with $d = 8$ hierarchical dimensions as shown in Figure 3.1. Each experiment used Amazon EC2's `c3.8xlarge` instances for servers, `c3.4xlarge` instances for workers, and `c3.2xlarge` instances for clients and

the manager. All instances were running Amazon Linux with Linux 3.14.35, ZeroMQ 4.0.5 and ZooKeeper 3.4.6.

3.4.1 Real-Time Load Balancing

The real-time load balancer in *vOLAP* is important for the elasticity of the system. As workers are added to *vOLAP*, the load balancer automatically moves data items to the new workers to balance the workload. Figure 3.4 shows the performance of the real-time load balancing method during a horizontal scale-up experiment. In this experiment, load phases are interleaved with insert and query benchmarking phases. At the start of each load phase, two additional workers are provisioned via SaltStack and added to the system to account for the increase in database size. The red region shows the minimum and maximum number of data elements stored on a worker. When new workers are introduced, they are empty, causing the minimum to go to zero. The effects of the load balancer are clearly visible as the gap between minimum and maximum worker size is reduced by moving data to the newly introduced workers. The number of migration operations for this process are shown as a dotted purple line associated with the right y-axis. Once balance is achieved, loading proceeds, increasing the minimum and maximum size per worker as new elements are inserted. Note that *vOLAP* can load balance at any time, including while data ingestion and queries are being performed. This experiment was designed in phases to ensure a stable benchmarking environment for evaluating scale-up performance, but it is not necessary to have discrete phases as shown here. In general, load balancing is performed concurrently with the inserts and OLAP queries. The load balancer continually monitors the system, and performs splits or migrations as necessary to maintain balance.

3.4.2 Horizontal Scale-Up Performance

Figure 3.5 shows the insert and OLAP query performance for various workloads as the system size increases. This data is from the same experiment as shown in Figure 3.4, where two new empty workers are added at each scale-up step. This is to demonstrate the elastic capabilities of *vOLAP* in a cloud environment. For each system size with p workers and $N \approx p \times 50$ million data elements, we tested *vOLAP* with inserts as well

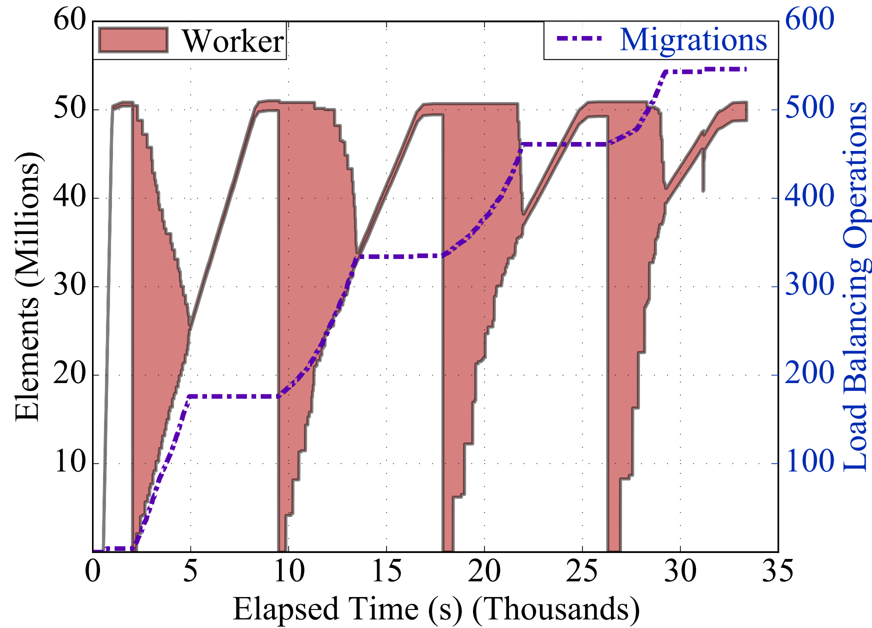


Figure 3.4: Load balancing data size per worker as database size N and number of workers p increases.

as queries with low (below 33%), medium (between 33% and 66%), and high (above 66%) coverage. The throughput and corresponding latency are shown in Figure 3.5.

Figure 3.5 shows that *vOLAP* scales well in an elastic cloud environment. As the database size increases, and processing resources (workers) are added at a fixed ratio, *vOLAP* maintains its performance over the entire range of database sizes. The curves in Figure 3.5, show that insert latency and throughput are unaffected by database size, as each insert requires communication from only a single server and worker. As expected, query latency and throughput slowly degrade as the database size increases, since each query must communicate an increasing number of workers. Despite this, *vOLAP* with a database size of 1 billion points was able to answer approximately 15,000 complex range queries per second and write 50,000 inserts per second.

In addition, *vOLAP* also supports bulk ingestion which allows data to be loaded at a much higher rate than point insertion. When many records are available to be bulk inserted at once, our experiments have shown *vOLAP* to be capable of ingesting data at over 400,000 items per second.

3.4.3 Insert and Query Performance

Figure 3.6 shows the throughput and latency for inserts and queries at a fixed database size (the largest size shown in Figure 3.5). The performance of *vOLAP* was measured for various workload mixes (percentage of insert operations) and query coverages (percentage of the database aggregated by an OLAP query).

Workload mix has a significant impact on throughput because each insert may trigger re-balancing operations on a data structure that is concurrently being used to answer queries. Coverage has a significant impact on performance because it globally impacts the number of different trees and workers that must be searched in answering an OLAP query, and locally impacts the number of nodes in each tree that must be searched.

In the experiments in Figure 3.6, insertion was approximately three times faster than queries, with a predictable linear relationship between workload mix and overall performance. This also demonstrates that inserts do not significantly impact concurrent query performance.

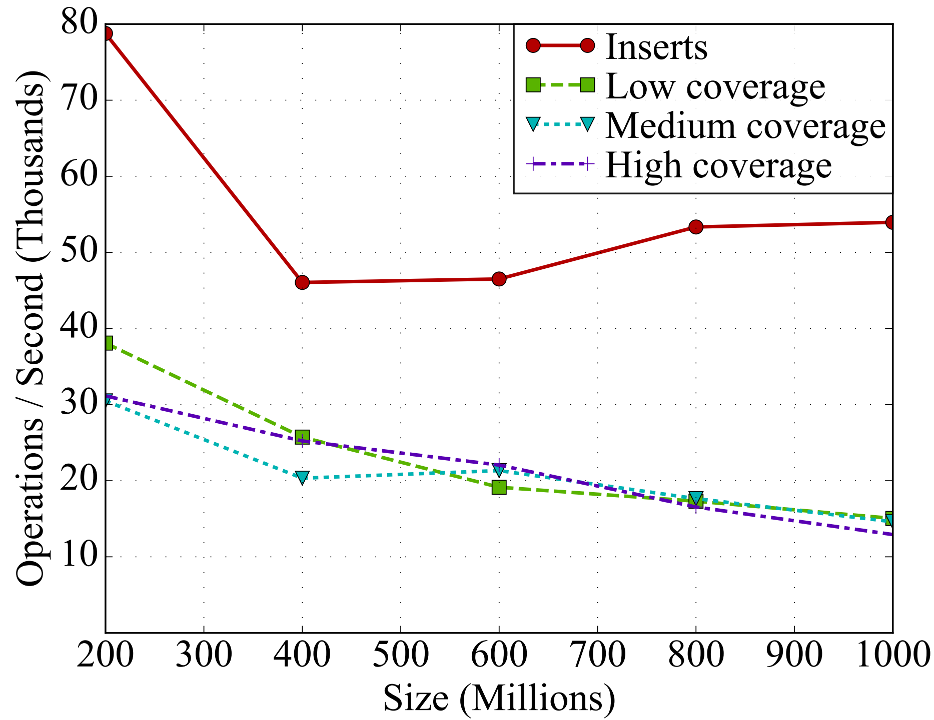
3.4.4 Coverage Impact

A more detailed analysis of the impact of query coverage on performance is shown in Figure 3.7. Both the impact on individual query time and the number of subsets searched are shown as a heat map, indicating how many queries showed which performance.

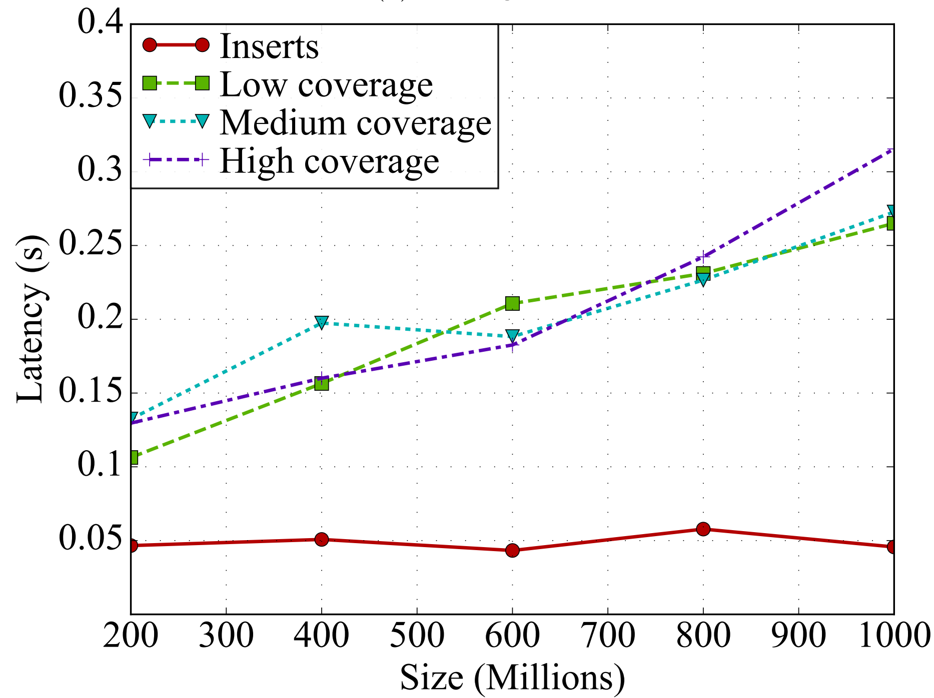
As shown in Figure 3.7(a), the majority of queries are executed very quickly, with a few outliers at low coverage. With high coverage, it is highly likely that aggregates will be found at higher levels in the tree, making deeper traversal unnecessary. However, with low coverage, it may be necessary to walk to the leaf level several times to find individual values, if none of the higher level directory nodes completely cover the query region.

As shown in Figure 3.7(b), the relationship between coverage and number of subsets searched is approximately linear, where increasing coverage requires an increasing number of subsets to be searched. There are some outlier points at around 50% coverage where many more subsets must be searched, however. This is due to queries that do not partition on the same boundaries as subset partitions, requiring a larger

number of subsets to be queried.

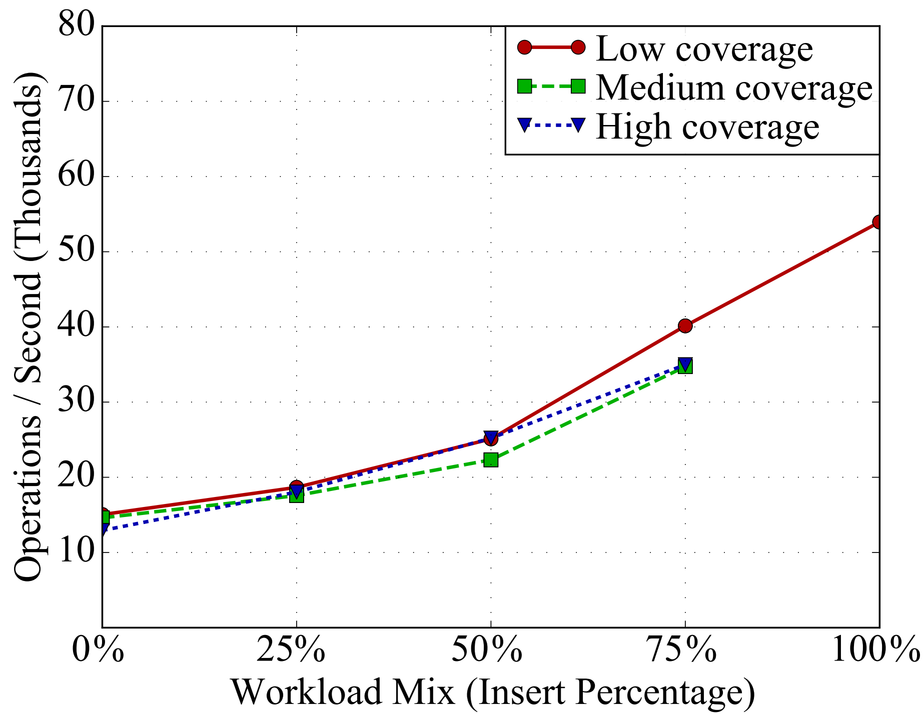


(a) Throughput

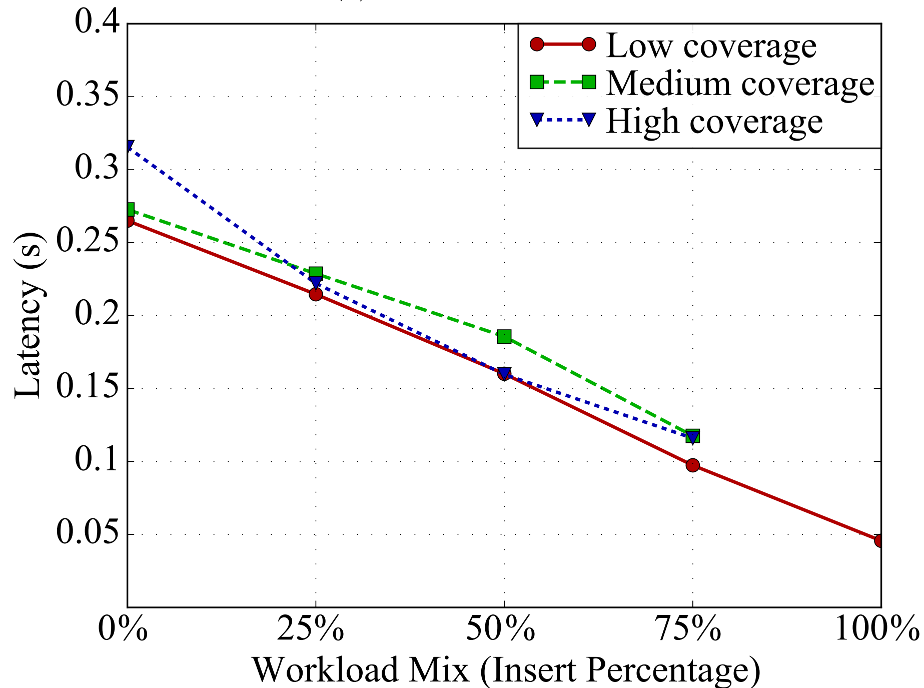


(b) Latency

Figure 3.5: Query and insert performance with increasing system size. Database size N and number of workers $p = N/50Mil$ ($4 \leq p \leq 20$) both increasing. Low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.

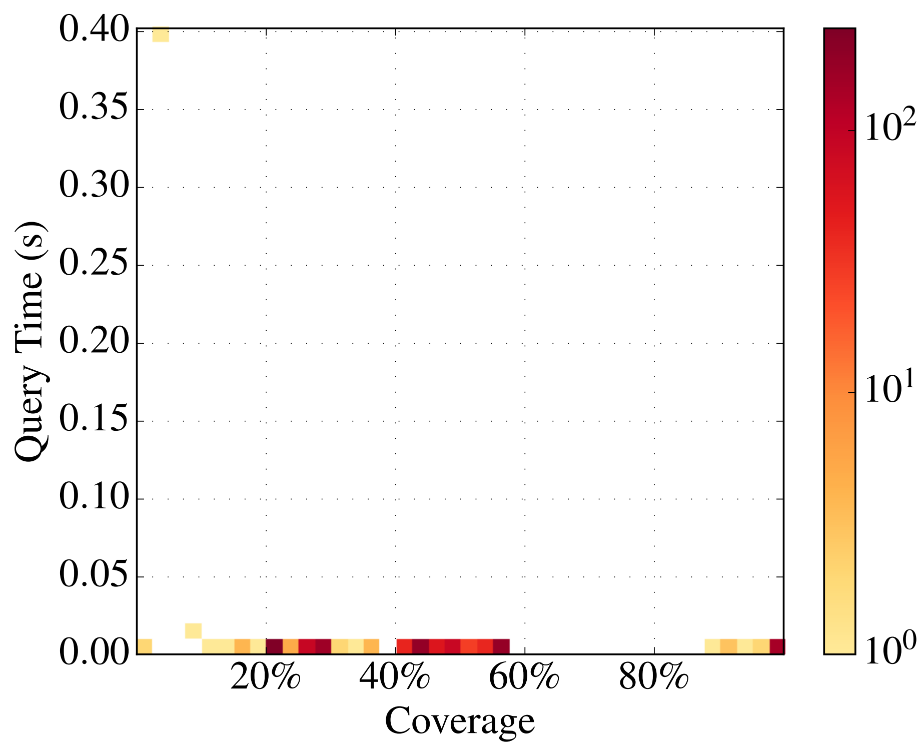


(a) Query throughput

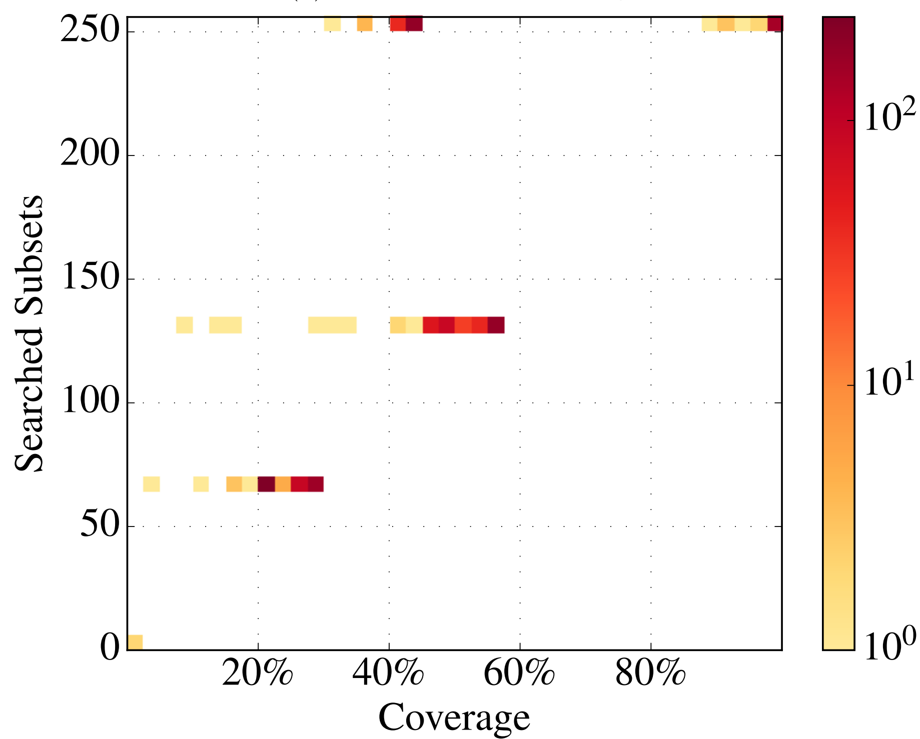


(b) Query latency

Figure 3.6: Performance for various workload mixes and query coverages. TPC-DS; $N = 1$ billion; $p = 20$; $j = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.



(a) Query time vs. coverage.



(b) Searched subsets vs. coverage.

Figure 3.7: Effect of coverage on query performance.

Chapter 4

Aggregate Quorum-based Replication

In this chapter, we propose a model and algorithms for applying quorum-based replication to distributed aggregate data stores. Recall that an OLAP system, or aggregate data store, contains a set of d-dimensional points with associated measure values, and answers queries which apply an aggregate function to subsets of the contained points. We begin by constructing a simple, generic model for a distributed aggregate store, and modify it to support replication and quorum inserts and queries. Quorum-based queries are especially difficult, due in part to problems in combining partial aggregations from replicas.

Later in the chapter, the model and algorithms are applied to add quorum-based replication to *vOLAP*, wherein additional internal structures, as well as revised and new manager operations, are required.

Finally, the impact quorum-based replication has on throughput and latency for *vOLAP* is experimentally measured on Amazon EC2 [2]. Given an 11-node distributed data store in which data is redundantly replicated N times, throughput and latency both experienced an approximately linear impact with respect to N, with higher coverage queries noticing the largest impact. For example, if N is set to 2, throughput for the most demanding queries is halved, while latency is doubled. The quorum consensus values W and R were found to have little impact on the latency of the system, due to the high-performance network environment used for the experiments.

4.1 Aggregate Quorums

In this section, we present a generic model of a distributed aggregate system and modify it to support quorum-based replication. In doing so, we hope to describe a set of algorithms which can be used to easily apply the benefits quorum-based replication provides for key-value stores to real-time distributed OLAP systems. We begin in Section 4.1.1 with a brief recap of quorum-based replication and how it typically is

implemented in key-value stores. We then define in Section 4.1.2 a simple model of a distributed aggregate system and, in Section 4.1.3, extend the model to support quorum-based reads and writes.

4.1.1 Motivation

As mentioned in Section 2.3.1, quorum-based replication is, in essence, a way of introducing redundancy in a system [72, 27, 48]. Under quorum-based replication [51, 43, 44], typically all data in a data store is stored redundantly on multiple nodes. Each node that stores a copy or replica of a set of data is said to "replicate" that data. By having multiple different nodes replicate the same data, the data store's availability improves. For example, without redundant copies of data on separate nodes, the loss of a single node in the system immediately results in the data stored on that node being inaccessible. If instead all data in the system was replicated by a factor of at least 2, other copies of the data would be available on other nodes in the event of the death of a single node. Replication can also serve to improve horizontal scaling [78], as query load can be spread across replicas, or to minimize latency for geographically distributed applications, allowing clients to access replicas that are geographically closest.

Because replicated data will inevitably introduce some amount of inconsistency or desynchronization to the system, quorum consensus rules are used to synchronize reads and writes [51, 43]. Under a quorum consensus, if each set of data is replicated N times, then each write operation is sent to all N replicas, and at least W write replies must be received by the client before the insert is considered committed. Similarly, for reads, a read request must be sent to all N replicas, and R responses must be received by the client before the query is considered committed. W and R can be set to low numbers to tune the system for low latency at the cost of consistency, or to higher numbers to tune the system for greater consistency at the cost of read or write latency.

4.1.2 Aggregate Model

We begin by defining a simple model for a distributed aggregate data store without any type of quorum-based replication. Refer to Figures 4.1, 4.2, 4.3 for graphical

illustrations. We list the following properties as the basis of the model:

- d-dimensional point data and their associated measure values are stored in a structure called a *bucket*.
- Location data used to determine which buckets store which points are stored in a structure called an *index*.
- Insertions and queries are produced by *clients*, and are sent to an index to route operations to the relevant buckets.

In this model, any node can take the role of 0 or 1 clients or indices, and any number of buckets. We use b to refer to the number of clients in the model, j to refer to the number of indices, and m to refer to the number of buckets. Buckets store the data points, and are capable of aggregating an arbitrary set of points. Buckets do the majority of the heavy lifting within the system in that they are responsible for both data storage and aggregation. Indices store location metadata used to determine which buckets store which points. Indices act as routers for query and insert requests; when a client wishes to insert or query the system, an index must be queried to determine which buckets the operation must be forwarded to. Responses from buckets are then propagated to the index, and back to the client. In the event a query covers multiple buckets, the index is responsible for aggregating each aggregation returned by each bucket. This is an important step, since, in this case, each bucket is only capable of aggregating a portion of the query, and the index must aggregate each aggregation in order to have a single, intelligible result to return to the client. We henceforth refer to aggregate query results from buckets as *partial aggregations* and aggregate query results from indices as *complete aggregations*.

Under this model, insertions are straightforward and operate as displayed in Figure 4.1. A client C_1 begins by forwarding an insert request to the index I_1. The insert request must consist of both a d-dimensional point and an associated measure value. For example, a point could be the time, day and location of a sale, while the measure could be the amount paid in that sale. Once the index receives the point and measure, a bucket (in this case, B_2) is selected using the index's stored metadata and the point data of the insert. The insert operation is then passed to the bucket, where the measure is stored at the location specified by the point. A "successful insert"

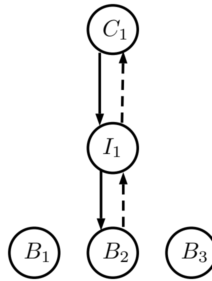


Figure 4.1: An insert operation under our generic real-time OLAP model

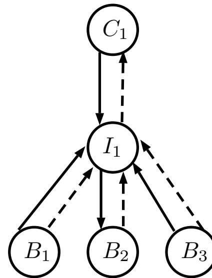


Figure 4.2: A query operation under our generic real-time OLAP model

response is then bubbled up to the index from the bucket, and then to the client, completing the insert operation.

Queries behave somewhat differently, as illustrated in Figure 4.2. A query consists of a specific portion of the d-dimensional space, which we will refer to as a box, and a binary associative aggregation function. The box defines the range of points to be aggregated. The query process begins with a client C_1 sending a query to an index I_1. Once the index I_1 receives the box and the aggregation function, it uses its metadata and the box to determine which buckets store points that lie in the box. In the case of Figure 4.2, the buckets B_1, B_2 and B_3 all store points contained by the box, and as such buckets B_1, B_2 and B_3 are all sent the query from the index I_1. Once a bucket receives a query from an index, it uses the aggregation function to aggregate every point in the box stored by the bucket, and sends the partial aggregation to the index. Once the index receives a partial aggregation result from every bucket it queried, it aggregates these partial aggregations and returns the complete aggregation as a response to the client, completing the query operation.

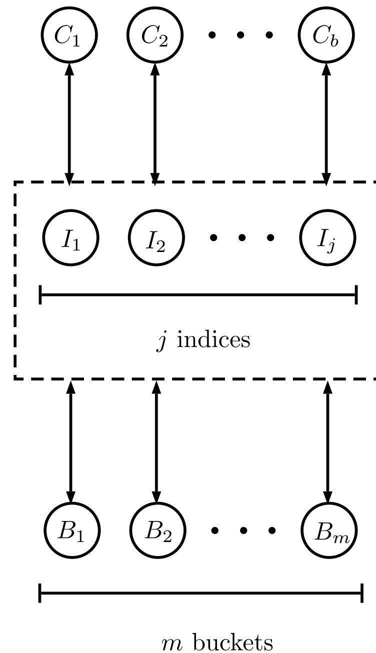


Figure 4.3: Diagram illustrating the node structure of the model. Clients C_1 to C_b each communicate with one of j indices, which communicate with as many as m buckets.

4.1.3 Aggregate Quorum-based Model

In order to apply the idea of quorum-based replication to distributed aggregate systems, we first introduce redundant or replicated buckets to the model in Section 4.1.2. We now say an aggregate quorum system has m bucket sets, rather than m individual buckets, where a bucket set is a group of buckets which all replicate the exact same group of points. We say that each bucket set is composed of N replicas, and that there are $N * m$ buckets in the system. Bucket set member B_u^x is the xth replica of the bucket set u. Note that bucket sets will not be stored on single nodes. Rather, they will be distributed across nodes to achieve redundant storage and fault tolerance. Figure 4.4 presents a graphical representation of the bucket sets in an aggregate quorum-based model.

4.1.4 Quorum Reads and Writes

Now that there are redundant buckets in the system, the way indices route inserts and queries to buckets must be modified. First, the location metadata stored in indices

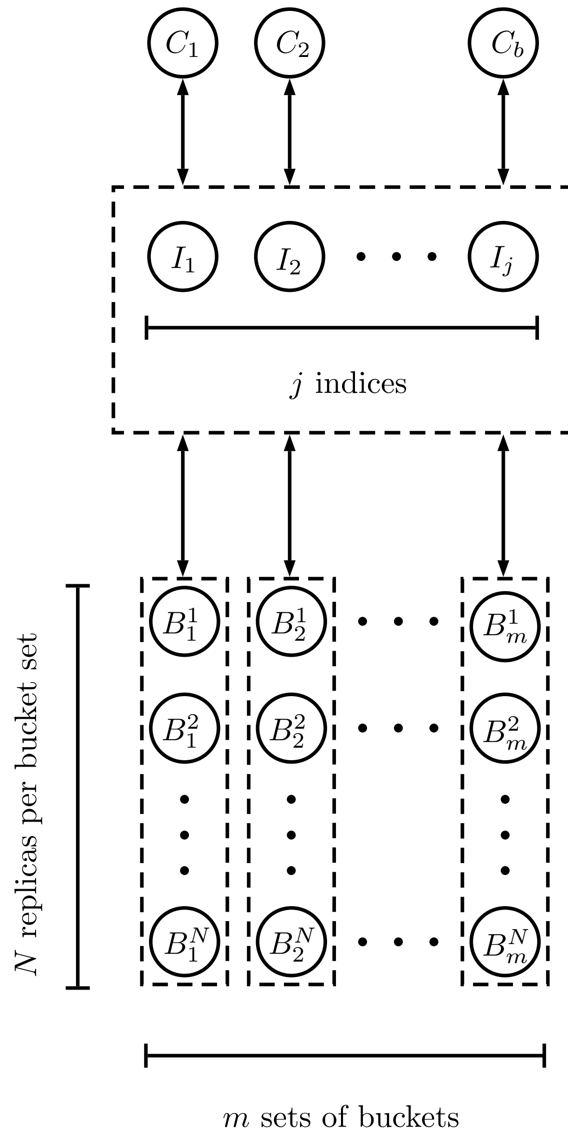


Figure 4.4: Diagram illustrating the node structure of our aggregate model, modified for quorum-based replication. Clients C_1 to C_b each communicate with one of j indices, which communicate with as many as m groups of buckets, each group containing N replicas. Dotted boxes correspond to groups of structures which replicate the same structure.

must now contain mappings of boxes or points to bucket sets, rather than individual buckets. In aggregate systems, since writes only deal with a single point in the system, much like key-value stores, a quorum-based insert operation in an aggregate quorum model is conceptually identical to that of a key-value store. Therefore, we describe our algorithm for quorum-based insertion in an aggregate data store as follows. A client sends to an index: point data specifying the location of the insert, measure data specifying the value of the insert, and an integer W specifying how many buckets must write the insert before the insert is considered committed. Using its location metadata, the index routes the insert to N buckets, all of which replicate data ranges containing the point specified by the client. Each bucket responds back to the index when completed, and the index responds back to the client after it receives exactly W bucket responses.

Quorum-based queries are somewhat more complicated. Similar to queries in our earlier model, the client begins by sending a query containing a box and an aggregation function. Unlike the earlier model, the client additionally sends an integer R, specifying the number of query replies to wait for from each relevant bucket set before returning the complete aggregation to the client. Once the query is sent to an index, the index uses its location metadata to determine which buckets contain points relevant to the query. The index then routes the query request to all relevant buckets, which use the aggregation function and box to compute partial aggregations. Once a bucket has computed its partial aggregation, it sends the result back to the index. After the index receives at least R partial aggregations from each relevant bucket set, the index selects the "best" partial aggregation from each bucket set. Once the best partial aggregation from each bucket set has been computed, the index aggregates the partial aggregations and sends the complete aggregation to the client. In Section 4.1.5, we describe the meaning of "best" in this context.

Figure 4.5 illustrates a quorum-based insert in our aggregate model. The client sends a single insert message containing a point and measure data to the client, and the client eventually sends back a message to the client specifying when the write was completed. Once the index receives the write request, it uses its metadata and the point data provided by the insert to determine which bucket set should store the point. Then, the index must determine which buckets belong to which bucket sets,

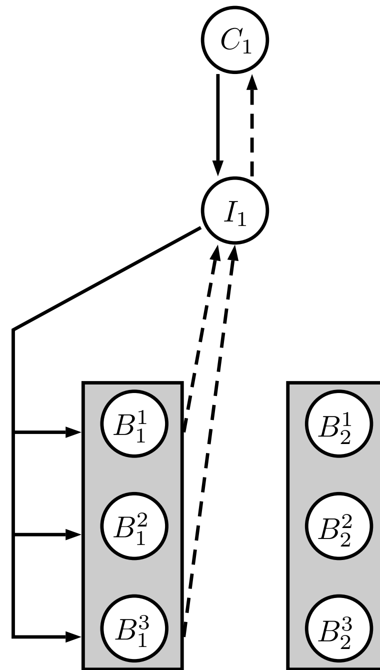


Figure 4.5: An example quorum-based insert operation in the aggregate model with $W = 2$

and route the insert's point and measure data to all buckets in the set. Each bucket in the set receives the insert, handles the operation as it normally would, and sends a reply to the index indicating the insert was committed. Once the index receives W replies from W buckets (in this case, $W = 2$), a message is sent from the client indicating the insert has been written to at least W buckets. Similarly, Figure 4.6 illustrates the quorum-based query process.

4.1.5 Combination of Partial Aggregations

In a quorum consensus, waiting for W replies after a write is used as a way to force the client to wait until the write has been written to W replicas before considering the insert readable on the data store. W's impact on consistency is straightforward: increasing W results in greater insert latencies and, consequently, greater confidence that the insert has been written to more replicas in the system.

However, waiting for R read responses is more complex. By waiting for R read responses, we have R different partial aggregations per bucket set with likely varying

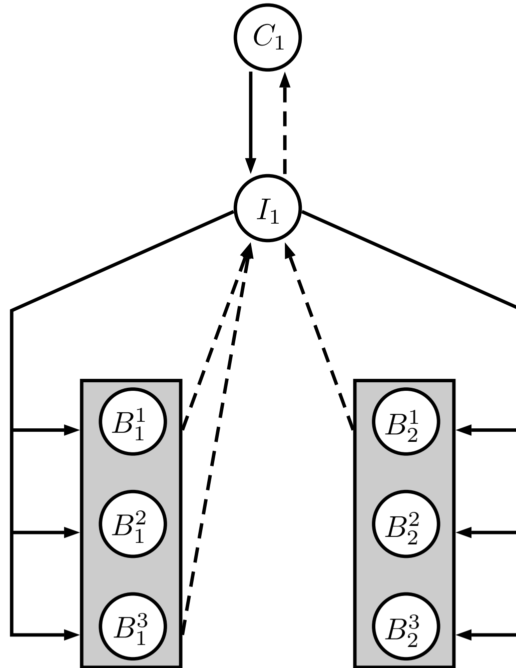


Figure 4.6: An example quorum-based query operation in the aggregate model with $R = 1$

degrees of consistency. This introduces the problem of deciding which partial aggregations are the most correct, and should be used in the construction of the complete aggregation. In key-value stores, this is typically resolved by including a timestamp of the last write to each query result [34]. In an aggregate data store, for queries which cover only a single point, the most recent result, which we will also consider to be the most correct, is simply the result with the most recent timestamp. However, the problem of selecting the most recent partial aggregation is more complicated. For example, if each point in an aggregate system had a timestamp, what would the timestamp of a partial aggregation look like? There does not appear to be a reasonable way of combining large sets of timestamps that provide a meaningful view of the correctness of a partial aggregation.

Consider, for example, a query on two buckets belonging to the same bucket set returning two different partial aggregations obtained by running the *max* aggregation function on the data specified in Table 4.1, where timestamps are represented in the left columns as seconds since system boot, and point data measure values are represented on the right. Looking only at the timestamps for each table, it would seem likely that

Table 4.1(a) would aggregate to yield a more correct *max* value than Table 4.1(b), as Table 4.1(a) contains more recent data. However, when looking at the values of the two subtables, Table 4.1(a) yields a *max* value of 5, while Table 4.1(b) yields a *max* value of 10. Aggregating the points in Table 4.1(b) results in the more correct partial aggregation, despite the timestamps for the points being exceptionally old relative to the other table. This example illustrates the challenges in using timestamps in an aggregate data store, and also suggests an alternative approach. As we will show in the following, a better way of deciding which partial aggregation is more correct must be based in comparing the two partial aggregations' numerical results; timestamps are not at all indicative of correctness in aggregate queries.

Timestamp	Value
1	2
3	4
4	3
8	5
13	1
20	1

(a) Timestamps and point data of each point used for one partial aggregation

Timestamp	Value
3	4
0	10

(b) Timestamps and point data of each point used for a second partial aggregation

Table 4.1: Timestamps (greater is more recent) and point data used to create two partial aggregations from two buckets in the same bucket set.

Since timestamps are unsuitable for determining the relative correctness of two partial aggregations, we instead turn to the value of the partial aggregation as the means of deciding the better of two partial aggregations. Because the aggregation function used to calculate the partial aggregation is important in the selection of the more correct result, we discuss the process of selection for each of the following six aggregation functions commonly used in OLAP systems [59, 50].

For a set of measure values M:

$$count(M) = \sum_{i=1}^{|M|} 1 \tag{4.1}$$

$$sum(M) = \sum_{i=1}^{|M|} M_i \tag{4.2}$$

$$max(M) = \max_{m \in M} m \tag{4.3}$$

$$min(M) = \min_{m \in M} m \tag{4.4}$$

$$mean(M) = \frac{sum(M)}{count(M)} \tag{4.5}$$

$$stdev(M) = \sqrt{\frac{\sum_{i=1}^{|M|}(M_i - mean(M))^2}{count(M)}} \tag{4.6}$$

We begin by exploring partial aggregation selection for the simplest case, the *count* aggregation function. If we assume points may only be inserted to the system, and any kind of delete operator is unsupported (as is conventional in OLAP), then partial aggregate selection becomes trivial. Since each relevant insert adds only a value of 1 to the result of the aggregation, regardless of measure value, and points are never removed from the system, the true aggregation value of a *count* query never decreases over time. We refer to an aggregation function that follows this property as monotonic, and define the two cases of monotonicity as follows.

Definition 3 (Monotonically increasing aggregation function) *An aggregation function is said to be monotonically increasing if and only if the result of the aggregation function never decreases as more points are added.*

Definition 4 (Monotonically decreasing aggregation function) *An aggregation function is said to be monotonically decreasing if and only if the result of the aggregation function never increases as more points are added.*

A monotonic aggregation function is either a monotonically increasing aggregation function or a monotonically decreasing aggregation function.

The property of monotonicity makes the selection of partial aggregations trivial for a specific set of aggregation functions under certain conditions. Since "stale" queries in our model can only arise from missing points (rather than having additional incorrect points), for monotonically increasing aggregation functions, the partial aggregation with the largest aggregation value is guaranteed to be less than or equal to the value of the true aggregation. In other words, with a monotonically increasing aggregation function, the largest partial result received determines the lower bound of what the true

partial aggregation value is. Clearly then, it follows that for monotonically increasing aggregation functions, the most correct partial aggregation function is always the aggregation with the greatest value.

Of course, as mentioned earlier, the *count* aggregation function is only monotonically increasing if points in the system are never deleted. If deletions are permitted, then each deletion of a single point can be viewed as decreasing the result of all *count* aggregation queries with boxes containing the deleted point by 1. Likewise, inserts can be viewed as increasing the result of all *count* aggregation queries with boxes containing the point by 1. Since, with deletions, an operation can either reduce or increase the value of a *count* query by 1, *count* is then not guaranteed to be monotonically increasing or monotonically decreasing. Non-monotonic aggregation functions make selection of correct partial aggregations especially difficult to do with perfect accuracy, since values of non-monotonic functions can move back to any previous value in history.

The *sum* aggregation function is similar to *count*, where instead of contributing a value of exactly 1, each point contributes the value of its measure data, which, for simplicity, we assume is drawn from a distribution D. If D is known to only yield values greater than or equal to 0, then *sum* is a monotonically increasing function. In this case, selection of the more correct partial aggregation is trivial: simply choose the partial aggregation with the largest *sum*. Likewise, if D is known to only yield values less than or equal to 0, *sum* is a monotonically decreasing function, and the most correct partial aggregation can always be selected by choosing the smallest *sum*. If D yields negative as well as positive values, *sum* loses its monotonic property. In this case, determining the most correct partial aggregation is difficult to do with perfect accuracy. Instead we offer a heuristic.

The heuristic for selecting partial aggregations from *sum* will be dependent on the mean of the measure distribution D. If the mean is known to be greater than 0, then given an infinite insert stream, the value of *sum* will approach ∞. In this case, we select the partial aggregation with the largest *sum* value. Similarly, if the mean is less than 0, we select the partial aggregation with the lowest value. If the mean is 0, then we expect the value of *sum* to approach 0 as more points are added to the system. In this case, we select the partial aggregation closest to 0. Since only the sign of the

mean is relevant to the heuristic, and the mean is simply *sum* over *count*, the sign of the mean can be determined by taking the sign of the *sum* partial aggregation. This heuristic method has low computational cost and on average yields the correct result.

The *min* and *max* aggregation functions are the most straightforward when selecting the most correct of multiple partial aggregations. Since both aggregation functions merely return the smallest and largest measure values of the relevant points respectively, *min* is always a monotonically decreasing function, while *max* is always a monotonically increasing function. Because of this, the smallest partial aggregation under *min* is always the most correct, and the largest partial aggregation under *max* is always the most correct.

The *mean* and *stdev* aggregation functions are the most difficult, since both are virtually never monotonic under any circumstances. In this case, we propose using the value of the *count* aggregation on the same space to make a heuristic guess. The partial aggregation with the largest *count* value is the one with the most samples, and is therefore the most likely to be closer to the true mean or standard deviation. Note that, while this method is likely to select the most correct partial aggregation of each of the 6 aggregation functions covered in this paper, it does not guarantee correctness for *max*, *min* or *sum*, of which there are methods for guaranteed correct selection, and is therefore ill-suited as a generic solution for every aggregation function.

In key-value quorum systems, guarantees of consistency can be derived from the $W + R > N$ rule [51]. This is unfortunately not the case in aggregate quorum systems. As mentioned in Section 2.3.1, in key-value quorum systems, if $W + R > N$, then the system is guaranteed perfect consistency on all of its read operations. This is because the overlap between W replicas which have committed the insert, and the R results from R replicas is always at least 1. However, with aggregate quorum rules, read or query operations are the result of several smaller reads of many points. In other words, unlike key-value quorum reads, single reads are affected by multiple writes.

Consider the very simple, but plausible, situation illustrated in Table 4.2. Each bucket contains a subset of the points inserted into the system, $P = \{1, 2, 3, 4\}$ (the measure values of the points are irrelevant in this example). We state that each insert was written with a write quorum value of $W = 2$, and that the replication factor of the system is $N = 3$. In the current system state, each insert has been committed

on only the minimum of 2 replicas each, leaving one replica insert of each insert still in transit. If, at the current system state, we initiate a simple *count* query on all points with $R = 2$, we will receive two of the three partial aggregations: 3 from B^1, 3 from B^2, 2 from B^3. Regardless of the pair received, the final result after selecting partial aggregations based on the largest value will always be 3. Note that our *count* query then is guaranteed to be incorrect, even though $W + R > N$, as the true value of *count* is $|P|$, or 4. Indeed, while the union of the committed points in any pair of two replicas is equal to the true set of committed points P, we lose the ability to combine points when we return only the value of the aggregation function to the index (rather than a list of points) in order to save on bandwidth. This property somewhat diminishes the benefit of greater values of R, as setting $W = N$ still guarantees perfect consistency, while setting $R = N$ does not.

Having described how the idea of quorum-based replication can be applied to an abstract aggregate data store, we now describe in the following section how this model can be applied to *vOLAP* in order to support quorum-replicated point data.

Point data
1
2
3

(a) Bucket B^1

Point data
1
2
4

(b) Bucket B^2

Point data
3
4

(c) Bucket B^3

Table 4.2: Point data after 4 insertions on a system with $N = 3$ replicas and $W = 2$.

4.2 Re-engineering *vOLAP* to Support Quorum-based Replication

In this section, the process of applying quorum-based replication to *vOLAP* is described. We first cover the design issues inherent in deciding whether entire worker nodes or individual subsets should be replicated. In addition, changes to the manager to support replication are discussed, specifically a new "replicate" operation and the need for a coordinated split operation. Changes to the insert and query operations are also discussed.

4.2.1 Subset Replication

Recall that in *vOLAP*, the system's point and measure data is divided into a set of structures called subsets, which are evenly distributed across a pool of worker nodes. The first major design decision to be considered is whether replication should be at the level of whole workers or individual subsets. During the initial design of quorum replication for *vOLAP*, it was determined that replication of entire worker nodes would be suboptimal when compared to replication of individual worker subsets. For example, by requiring replicas to replicate all data on the worker node, a worker node and its replicas may never use any more memory than that of the smallest replica, so as to not cause failure on the smallest replica. Furthermore, the migrate operation becomes vastly more complex, as the migration of a subset must be coordinated and initiated for all replicas on the sending and receiving end. Additionally, when compared to subset replication, worker replication impacts the manager's ability to properly load balance, since the subsets on each replica may experience different loads, but the manager is forced to use the same subset composition for each replica. Subset replication, on the other hand, avoids many of these problems. Individual subsets are typically configured to be relatively small, so hardware with less storage capacity can simply replicate a number of subsets in proportion to its memory. While some manager operations are still impacted by the introduction of replication, migrations remain largely unchanged, as it is no longer required that every replicating subset must migrate if one does. Load balancing is also better under subset-based replication, as the manager works with small grained subsets, rather than entire nodes of data.

In order to implement quorum-based subset replication, a new structure must be added to the *vOLAP* system image, which will eventually propagate down to each server's local image. The subset-worker map is a structure which provides a mapping of a single subset ID to a set of worker locations which replicate that subset. The elements of the subset-worker map are maintained globally via ZooKeeper whenever a new subset or replica is created. Worker locations are removed from the map whenever a worker stops responding to requests from the server or appears to be dead. Subset IDs are removed from the map whenever subsets are deleted, for example, through a split operation. Servers listen for any signals from ZooKeeper that reflect the map has been changed, and update as soon as possible. While this is another element of

data desynchronization in *vOLAP*, changes to this structure beyond the initial system boot up are so infrequent that they have little impact on correctness in practice. This structure allows servers, after getting the relevant subset IDs from the index structure, to route insert and query operations to all workers replicating the relevant subsets.

4.2.2 Replica Management

For the creation of new subset replicas, a new manager operation is presented. The new *replicate* operation can be viewed as a derivation of the load balancer's migrate operation, without the deletion of the subset on the source worker. Essentially, the replicate operation instructs a subset to serialize its contents and send them to the specified worker. The worker that receives the serialized data deserializes it and adds it as a new subset. Once the operation completes, the manager writes the receiving worker's location under the subset's ID in ZooKeeper's subset-worker map. The following is a more detailed view of the operation:

- The manager sends a replicate request with a subset ID and receiving worker location to a source worker.
- The source worker serializes the relevant subset and sends it to the receiving worker.
- The receiving worker deserializes the subset and updates the system image's subset-worker map to reflect the newly replicated subset. The receiving worker can receive writes to this subset as soon as a server pulls the new subset-worker map.
- The receiving worker sends a message to the source worker, notifying the source worker that the subset was received successfully.
- A message is sent to the manager from the source worker signifying the completion of the replicate operation.

Unfortunately, if any insert operations are issued to a subset during the window of time where the source subset has been serialized and the server has not yet updated its subset-worker map, the newly replicated subset will miss those inserts, resulting

in a subset replica that is always missing a small number of points. Under these conditions, *vOLAP* is not an eventually consistent system. To rectify this, a form of anti-entropy [37] is needed; a protocol for workers to communicate with each other to find and resolve any inconsistencies between subsets. This specific type of desynchronization is avoided when replicate operations are initiated in periods of time where there are no inserts, such as initial system boot up. The design and implementation of an anti-entropy mechanism has been left for future work.

4.2.3 Reading and Writing

The implementation of a quorum insert in *vOLAP* is relatively straightforward. Since inserts in *vOLAP* operate on point data, much like key-value stores, there is little different here from the classical quorum write operation. Once the server receives the insert operation from the client, containing point data and measure data as usual, the server simply needs to determine the relevant subset using the index structure, and then use the index in conjunction with the subset-worker map to retrieve the workers which replicate the relevant subset. The server forwards the insert to each relevant worker, who complete the write exactly as they would without quorum replication, and return a message back to the server. The server needs to only keep an integer containing the number of responses received for each insert in progress. As it receives a reply from each relevant worker, the number is incremented. Once the number reaches W, an "insert complete" message is sent to the client. From the perspective of the server, the insert has not truly completed yet. The server waits for responses from the remaining $N - W$ workers before the server completes the insert, in order to ensure that the remaining workers have not died. If a particular worker takes too long to respond to the insert request, specified by a configurable timeout parameter, the server assumes the worker has died and stops waiting for a response. The server also removes the worker from its local subset-worker map, and pushes its new map to ZooKeeper, so as to alert other servers not to send any more requests to the dead worker. In the event that a worker eventually becomes responsive again, it is wiped of all data and treated as if a new node has been added to the system, as the worker may have permanently lost several insert requests and violates the eventually consistent property of the system.

Queries behave in a somewhat similar manner. As with inserts, when a server receives a query request, it first uses the provided box and the index structure to search for all subsets relevant to the query. This is then converted into a list of relevant worker locations through the subset-worker map. The query is then forwarded once to each worker which replicates a relevant subset. During the routing of the queries, the server must keep track of the specific subsets queried, as well as the number of query replies for each subset replica. Once a worker receives a query request, it aggregates its relevant subsets using the box contained in the query request and returns to the server a partial aggregation for each relevant subset it contains. Here as well, the workers are required to reply before a timeout has passed. When the server receives a query reply from a worker, it iterates through each partial aggregation returned, increments the counter corresponding to the number of replies received for each specific subset, and stores a copy of the partial aggregation by subset ID for later combination. Once R replies have been received from each queried subset, which the server simply checks by determining that the subset reply counter for each relevant subset has met or exceeded R, the server combines each partial aggregation for each subset, using the combination rules described in Section 4.1. The combined subset results are then aggregated using the aggregation function provided by the query, and the final complete aggregation is passed to the client.

4.2.4 Split Coordination

The introduction of subset replication introduces some new challenges to the manager's split operation. Previously, since there were no subset replicas, a split operation merely required coordination between a single worker and the manager. When a manager observed a subset on a specific node began to exceed the maximum subset size limit, it would simply issue a split request to that worker with the relevant subset ID. When the worker finished its split, a message was then sent to the manager which notified all servers with the split subset ID as well as the two new subset IDs resulting from the split. With subset replicas, splits must now be coordinated across each subset replica in order to ensure all incoming inserts are received on each subset replica during a split. For example, without split synchronization across replicas, a point may simultaneously belong to the unsplit subset A, as well as a split subset B. Since the

index data structure maps points to only individual subsets, any inserts received while the system is in this “partially split” state will result in only one subset receiving the insert. When the unsplit subset eventually splits, it will permanently be missing any inserts sent within the range of time the subset was partially split.

The coordinated split operation must also ensure that each subset splits on the same hyperplane. Since *vOLAP* is an eventually consistent system, it is possible that any two replicas of the same subset may differ by an unknown margin under certain circumstances. In extreme cases, this could result in subset replicas locally determining differing optimal hyperplanes for a single split operation, resulting in replicas of the same subset containing different ranges of points; a source of irreparable inconsistency. To rectify this, the coordinated split operation needs another additional step: the manager, before initiating any formal split requests to replicas, must first query a single replica for its best separating hyperplane at the current moment in time. The manager must then send that hyperplane to each other replica as the hyperplane it needs to split on.

In summation, the coordinated split operation begins as illustrated in Figure 4.7 with the manager sending a hyperplane query to a single arbitrarily chosen replica (1). The replica makes a decision as to what hyperplane would best split the subset, and sends the hyperplane as a response to the manager (2). A split request containing the hyperplane is then sent to each worker containing a replica of the specific subset (3). Each worker who receives the split request splits the subset using the hyperplane and sends a reply to the manager once it completes the local split (4). The manager, after receiving a split reply from each worker, finalizes the operation by notifying each server (5) and writing a record of the split to ZooKeeper (6). As with inserts and queries, any workers which do not send a reply in an allotted amount of time are considered dead and are removed from the system’s global record.

4.3 Evaluation

To evaluate the impact the aggregate quorum implementation has on *vOLAP*, several experiments were run on Amazon EC2 to determine the latency and throughput of the system under varying quorum configurations and query coverages. Like the evaluation of *vOLAP* prior to the quorum implementation in Section 3.4, the TPC-DS

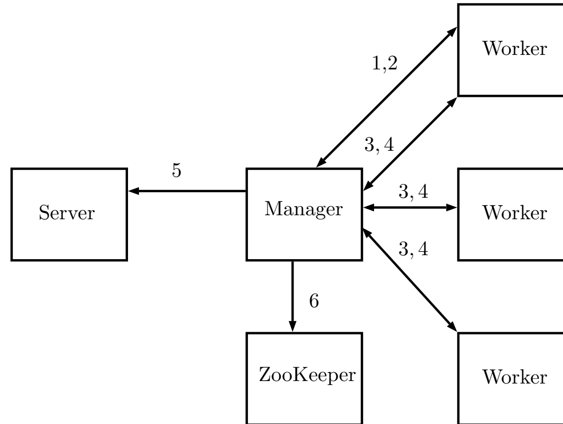


Figure 4.7: Communication steps required for a coordinated split across three workers containing the same subset.

decision support data set from the Transaction Processing Council was used with $d = 8$ hierarchical dimensions. For each experiment, *vOLAP* was pre-loaded with 10 million points from the TPC-DS data set. All nodes in this set of experiments were run on Amazon Linux with Linux 3.14.35, ZeroMQ 4.0.5 and ZooKeeper 3.4.6, and were provisioned into a single low-latency virtual private cloud. Six `c4.xlarge` nodes were used for workers, two `c4.2xlarge` nodes for servers, two `c4.xlarge` nodes for clients, and one `c4.xlarge` node for the manager.

4.3.1 Infrastructure Overhead

In this section, we examine the performance cost of the system infrastructure required to support quorum-based operations (that is, management of the addition of the worker-subset map and changes to the insert and query operations).

Figure 4.9(a) shows the operational throughput of *vOLAP* under varying coverages with quorum configuration `[N=1, W=1, R=1]`. The x-axis represents the proportion of insert operations in the workload. The y-axis represents operations (in thousands) per second given the current workload mix. In this configuration, there is no replication of data and *vOLAP* is functionally the same as it was prior to the quorum implementation. Figure 4.9(b) shows the operational latency under the same quorum configuration. The y-axis corresponds to the average latency of each operation. Both figures are used as a baseline of comparison for other experiments in this section.

Figures 4.8(a) and (b) illustrate the throughput and latency respectively of *vOLAP*

on the same hardware prior to the implementation of the quorum infrastructure. Since the computational costs of these changes by themselves are negligible, little difference between the latency and throughput is observed when compared with results from `[N=1, W=1, R=1]` in Figures 4.9(a) and (b).

4.3.2 Replication Factor

In this section we explore the impact of increasing N, the replication factor. Figure 4.9 illustrates the throughput and latency observed for varying replication factors. Each row of subfigures corresponds to a specific replication factor, and each column corresponds to either throughput on the left, or latency on the right.

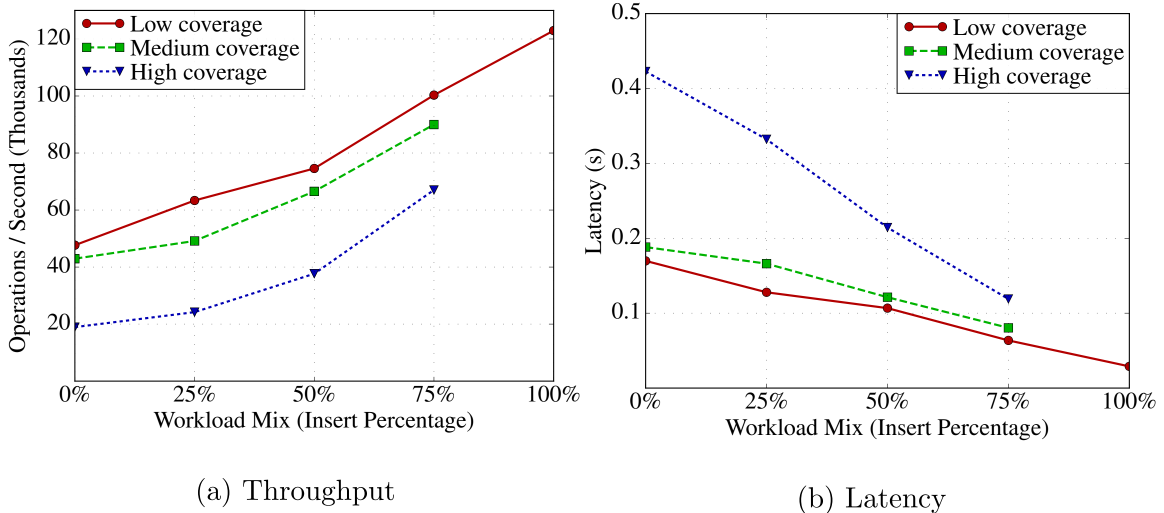
In Figure 4.9(c), *vOLAP*'s throughput for quorum configuration `[N=2, W=1, R=1]` is illustrated. Compared to quorum configuration `[N=1, W=1, R=1]`, throughput for workloads consisting entirely of queries drops by approximately 35% for medium and high coverage queries, as the amount of local computation (subset aggregation) required across all worker nodes doubles, while the amount of client and server computation remains approximately the same. Low coverage queries suffer only a 10% loss in throughput. Since low coverage queries typically only aggregate a small number of subsets, doubling the amount of worker computation required for each low coverage query likely does not add enough load onto the workers to make them the system's bottleneck in this case. Insert operation throughput is approximately 28% slower than without replication. Similar to query throughput, the amount of computation required on the workers double, while the work done on the client and server remain approximately the same.

Replication has a harsher impact on latency than it does on throughput. Figure 4.9(d), shows *vOLAP*'s latency for quorum configuration `[N=2, W=1, R=1]`. Setting a replication factor of 2 to the system increases latency for high, medium and low coverage query workload mixes by approximately 80%, 50% and 20%, respectively. Note that, since W and R are set to 1, the increased latency is solely due to increased load on worker nodes. High and medium coverage queries suffer greater penalties, as their latency is largely bound by worker load. Similarly, latency for purely insert-based workloads appears to be highly sensitive to increased worker load, increasing by 100%.

Figure 4.9(e) illustrates *vOLAP*'s throughput with a quorum configuration of

[N=3, W=1, R=1]. We observe generally the same slow-down of throughput rates in increasing the replication factor from 2 to 3 as was encountered when increasing the replication factor from 1 to 2; high coverage query throughput decreases by 30%, medium coverage throughput decreases by 18%, low coverage throughput decreases by 13% and insert throughput by 24%.

In Figure 4.9(f), *vOLAP*'s latency with quorum configuration [N=3, W=1, R=1] is displayed. Going from a replication factor of 2 to 3 results in further increases to average latency for queries and inserts. Latency for all query coverages decrease at nearly half the rate going from a replication factor of 1 to 2, while the insert latency again increases by approximately 100%.



(a) Throughput

(b) Latency

Figure 4.8: Latency and throughput performance for various workload mixes and query coverages prior to the implementation of quorum replication. TPC-DS data; $N = 10$ million; $p = 6$; $j = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.

4.3.3 Read and Write Quorums

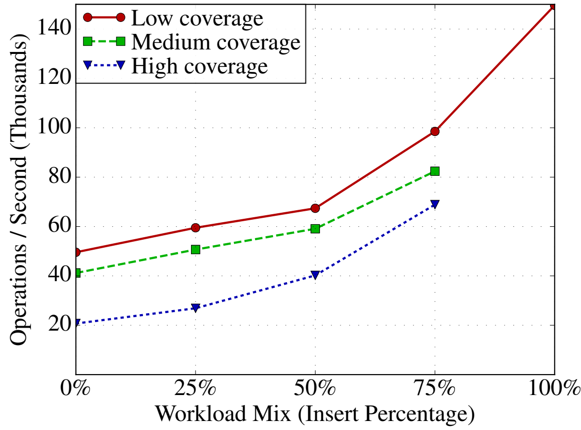
In this section, we explore the impact of varying W and R on a system with a replication factor of $N = 3$. In practice, this is the most commonly encountered replication factor used for quorum-replicated key-value stores [22].

Figure 4.10 illustrates the latency observed for varying values of W and R with a replication factor of $N = 3$. For readability, the latency for quorum configuration [N=3, W=1, R=1] from the previous section is repeated in Figure 4.10(a).

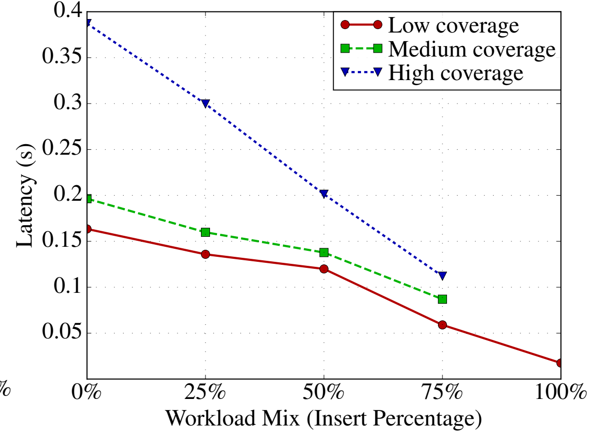
In Figure 4.10(b), we examine the effects an increased write quorum value has on *vOLAP*'s latency. The same latency benchmark from the previous section was rerun with a quorum configuration of `[N=3, W=2, R=1]`. Despite the increased number of worker responses to complete an insert, no significant differences in insert latency were observed when compared with quorum configuration `[N=3, W=1, R=1]` in Figure 4.10(a). This is due to the network configuration used for the set of experiments in this section. Since all nodes in *vOLAP* are provisioned into the same virtual private cloud, they all communicate with each other through a high-speed private network. In addition, all nodes are placed in the same availability zone and are therefore in the same geographical area. Because of this, network communication latencies between nodes are uniform, small and relatively stable. Provided the load between workers is evenly balanced, this results in each redundant write from server to worker completing at approximately the same time, as is observed. Furthermore, at least within the same availability zone, we note that the largest cause of latency in *vOLAP* are queuing delays between clients and servers; any increase in worker latency would be largely overshadowed by client-to-server communication latency.

Figure 4.10(c) examines the latency of *vOLAP* with a greater read quorum value (`[N=3, W=1, R=2]`). Similar to the value of W, increasing R has no distinguishable impact on query latency. This is again due to the low-latency network configuration of the system.

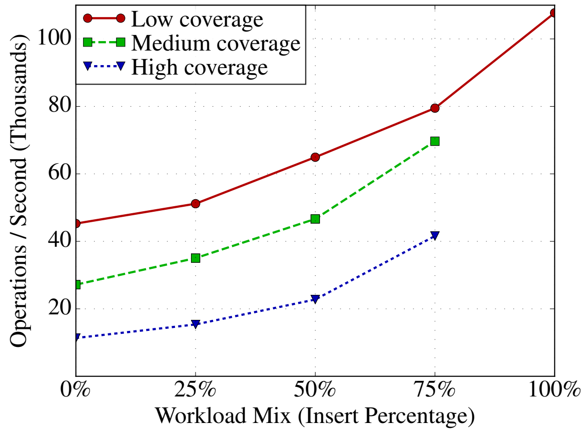
In the case throughput for quorum configurations `[N=3, W=2, R=1]` and `[N=3, W=1, R=2]`, we note that the value of W and R has little impact on the system, as no additional computational load is introduced.



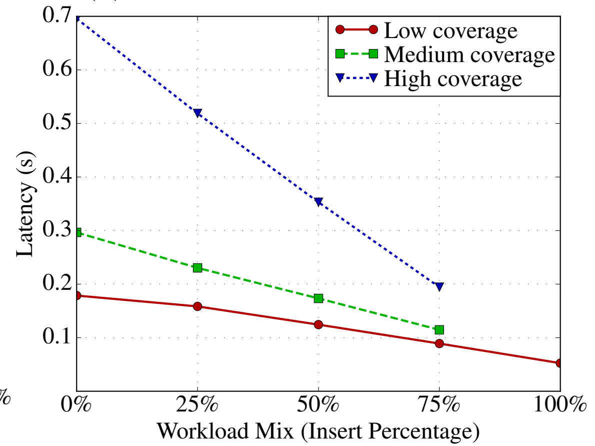
(a) Throughput for `[N=1, W=1, R=1]`



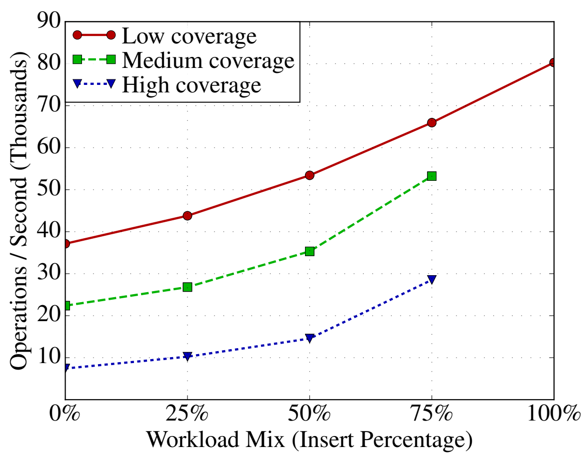
(b) Latency for `[N=1, W=1, R=1]`



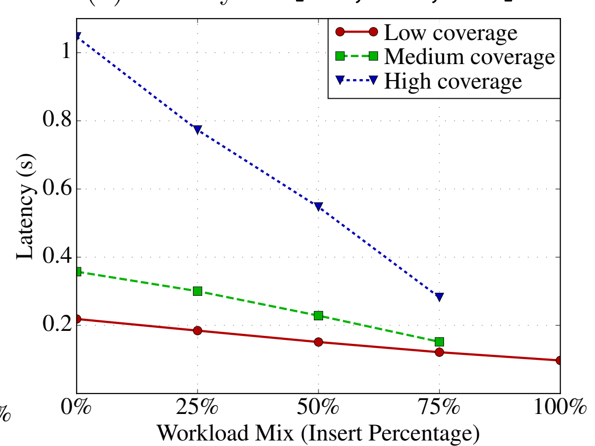
(c) Throughput for `[N=2, W=1, R=1]`



(d) Latency for `[N=2, W=1, R=1]`

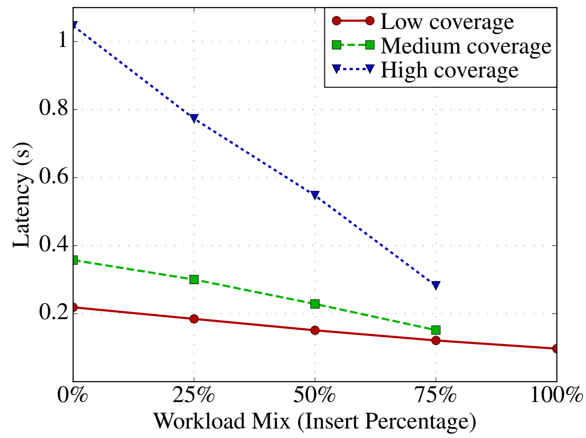


(e) Throughput for `[N=3, W=1, R=1]`

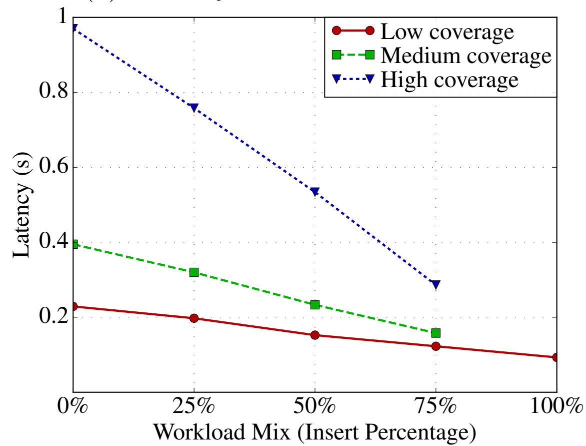


(f) Latency for `[N=3, W=1, R=1]`

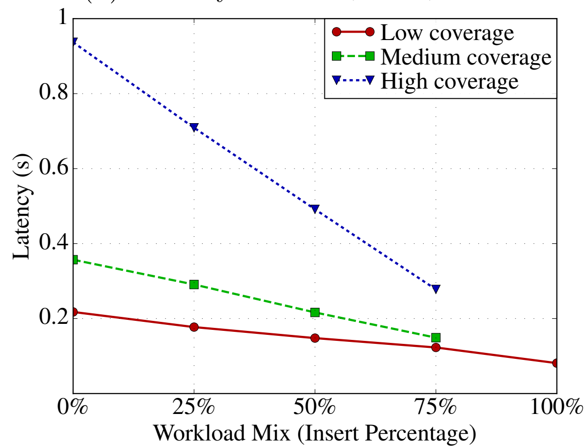
Figure 4.9: Latency and throughput performance for various workload mixes, query coverages, and replication factors. TPC-DS data; $N = 10$ million; $p = 6$; $j = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.



(a) Latency for `[N=3, W=1, R=1]`



(b) Latency for `[N=3, W=2, R=1]`



(c) Latency for `[N=3, W=1, R=2]`

Figure 4.10: Latency performance for various workload mixes, query coverages, and values of W and R. TPC-DS data; $N = 10$ million; $p = 6$; $j = 2$; low coverage: below 33%; medium coverage: between 33% and 66%; high coverage: above 66%.

Chapter 5

PBS for Aggregate Data Stores (A-PBS)

In this chapter, we show how ideas originally developed in Probabilistically Bounded Staleness (PBS) [21, 22, 23], a recent model for exploring the trade-offs between consistency and latency in key-value stores, can be adapted for aggregate data stores. The resulting model, which we call *Aggregate Probabilistically Bounded Staleness (A-PBS)*, is a means of analyzing consistency in aggregate data stores. Like PBS, A-PBS defines metrics examining the consistency of aggregate queries in terms of missed writes. Unlike PBS, since aggregate queries, especially in OLAP, are conventionally numerical by nature [50], additional consistency metrics are introduced that view consistency from the perspective of numerical error.

Both of the distributed aggregate system models presented in Sections 4.1.2 and 4.1.3 are given a set of *system parameters* in this chapter. The models, when combined with a set of system parameters, can be used to accurately represent the performance characteristics with respect to consistency for a wide variety of aggregate data stores. With this, a Monte Carlo simulation (described in this chapter) can be used to determine various staleness metrics of a system.

5.1 Metrics of Aggregate Staleness

5.1.1 Foundations

While queries in key-value stores essentially pull a single value specified by a key from a node, aggregate queries may involve a much larger percentage of the data. Unlike key-value stores, a single query in a typical aggregate OLAP system can operate on anything between a single point in the system to an aggregation of all points across all nodes in the system. This is an important distinction which divides key-value staleness analysis from aggregate OLAP-style staleness analysis. Specifically, a key-value query may only ever cover a single key (or point in OLAP parlance), while an

aggregate query often covers many different points specified by a multi-dimensional bounding box in d-dimensional space. Because of this distinction, staleness analysis in aggregate systems does not enjoy many of the same simplicities key-value staleness analysis does. Instead of needing to know only the write history of a single key in the system, staleness analysis on an aggregate system requires the write history of each point within the multi-dimensional bounding box relevant to the query to be known. Therefore, we first begin with a set of definitions describing the write history of a system, henceforth described as the *data stream*, and the *coverage* of an aggregate query, both fundamental for further discussion of correctness in aggregate stores.

Definition 5 (Input data stream DATA(n, f, D)) *DATA(n, f, D) is a stream of n insert operations, where each insert, with measure value sampled from the distribution D, is sent from a client every* $i = 1/f$ *units of time.*

Definition 6 (Aggregate query Q**)** *An aggregate query* Q *is defined by an aggregate function* A *and a multi-dimensional bounding box specifying the subset of an input stream to be aggregated with* A*. We refer to the coverage of a query* C *as the percentage of insertions of a data stream covered by the bounding box.*

A data stream DATA(n, f, D) describes the number of points (n) and the rate at which they are ingested into the system (f), as well as the distribution of the numerical values of these inserts (D). n corresponds to the size of the stream, f refers to the rate in seconds at which inserts are sent to the system, and D refers to the distribution from which measure values are sampled. A query Q has coverage C, that is, the percentage of insertions in the data stream which are relevant to the query Q, and the aggregation function A. Using these two definitions, we can model a sequence of insertions followed by a single query. For example, if we are interested in the consistency of a system under maximum load, whose maximum insert throughput is $10,000$ inserts per second, with about $1,000,000$ insertions in the database, we need to only model a DATA(1,000,000, 10,000, D) stream and observe the result of a query Q initiated some time after all operations in the data stream have been sent. Figure 5.1 presents a graphical representation of a simple data stream.

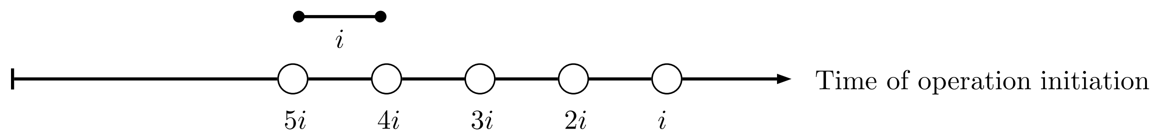


Figure 5.1: A DATA(5, f, D) insert stream. The white circles represent the points in time in which inserts in the stream are sent from the client. All inserts are spaced $i = 1/f$ seconds apart.

5.1.2 (t, k)-staleness

With the definitions from the previous section in mind, we proceed to define our first metric of staleness in an aggregate setting.

Definition 7 ((t, k)-staleness) *Given an insert data stream DATA(n, f, D), a query Q, initiated t units of time after the last insert in the stream has been sent, has (t, k)-staleness if and only if more than k insert operations covered by the query's bounding box were not included in the computation of the value of A.*

Note that Definition 7 makes no comment on the correctness of the result. Indeed, while there are many situations where a stale query could return an incorrect result, there are also situations where a stale query could yield correct results. For example, the *max* aggregation function is likely to not be sensitive to missing a few data points, while *count* will always return the incorrect result if any inserts within the query's bounding box are missing.

Essentially, for a given data stream and query, (t, k)-*staleness* allows us to make a binary "fresh" or "not fresh" judgement of a single instance of that query and stream. If the query happens to miss more than k relevant points in its computation of the aggregate result, the query has (t, k)-staleness. Otherwise, the query does not have (t, k)-staleness.

Similar to the key-value staleness metrics presented in [22], (t, k)-staleness has two important "slack" parameters, t and k. t refers to the units of time between the initiation of the last insert in the stream and the initiation of the query. Since eventually consistent systems always converge towards consistency over time, greater values of t will lead to fewer queries with (t, k)-staleness, while smaller values of t will be more likely to yield queries with (t, k)-staleness. t behaves essentially the same as

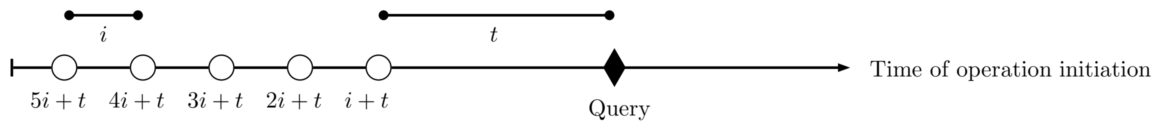


Figure 5.2: An insert stream and query for (t, k)-staleness. The white circles and black diamond represent the respective initiation time of the inserts and query.

it does in key-value PBS. k corresponds to the maximum number of allowable missed inserts.

Figure 5.2 provides a visual representation of the slack parameters in (t, k)-staleness. As time proceeds from left to right, insert operations, represented by white circles, are sent to the system at a rate of $i = 1/f$ units of time. Once the final insert has been initiated, t units of time are waited until the query, represented by the black diamond, is also initiated. Figures 5.3 and 5.4 demonstrate queries with (t, k)-staleness and without (t, k)-staleness, respectively.

We expect to encounter fewer queries with (t, k)-staleness for larger values of k, as we gain tolerance of a larger number of missed inserts. This is somewhat different from the definition of k-staleness for key-value PBS. In key-value stores, often values are overwritten by new values, and reflected by a different version number. k-staleness for key-value PBS specifies how many *versions* from the most recent a query can return while still being fresh. In our data stream model, we allow only insert operations; overwriting measure values of existing points is not permitted.

In contrast to the definition of freshness in key-value stores, aggregate query freshness seems like a much harder quality to consistently ensure. While a typical key-value query's staleness is determined by the write history of a single key in the system, staleness in aggregate systems is instead determined by a large stream of points. When one considers that it is not unusual for a bounding box in an OLAP query to cover millions of points, it is apparent that (t, k)-staleness is a potentially much more difficult quality to achieve than the key-value PBS equivalent.

5.1.3 Data Stream Sizes

From the perspective of (t, k)-staleness, the f parameter from a DATA(n, f, D) data stream also partially determines the effective number of inserts covered by the query. Remember that, in (t, k)-staleness, a DATA(n, f, D) data stream is provided. However,

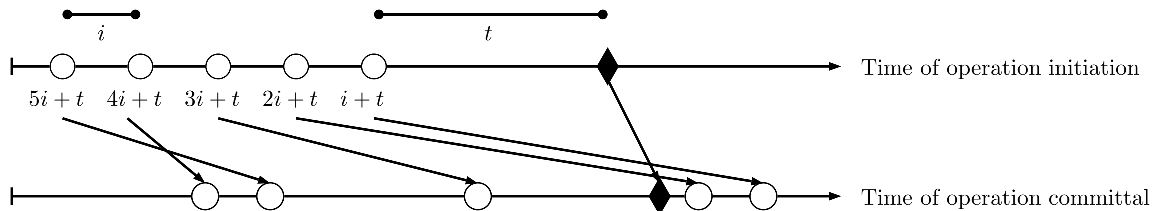


Figure 5.3: A query that has $(t, k{=}1)$-staleness. The upper bar represents the time of initiation of a query or insert, while the bottom bar represents the time at which the corresponding insert is readable, or the time at which the query begins. The last two inserts in the stream and the query are reordered, so more than $k = 1$ inserts are missed.

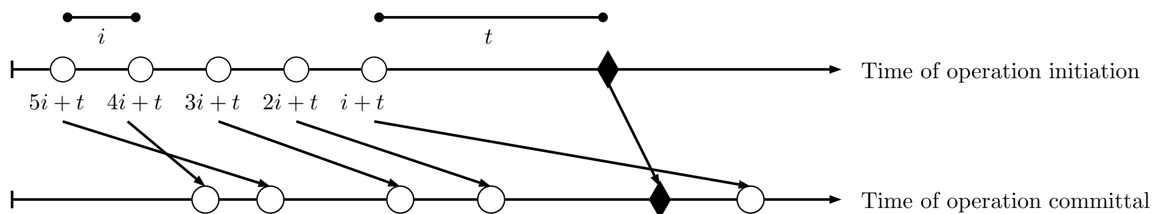


Figure 5.4: A query that does not have $(t, k{=}1)$-staleness. The upper bar represents the time of initiation of a query or insert, while the bottom bar represents the time at which the corresponding insert is readable, or the time at which the query begins. Since $k = 1$, the reordering of the last insert in the stream and the query does not impact (t, k)-staleness.

since each point in the set is initiated $i = 1/f$ units of time before the next point, the initiation time of each insert, relative to the time of query, increases by i for each point. In other words, while the number of inserts in the stream may be n, the difference between the time of query initiation and insert initiation will grow larger for each consecutive insert in the stream. Then, in any eventual consistent system that has an upper bound on how long a single point can remain inconsistent, only the inserts in the stream initiated before that bound are in danger of being missed; all of the other inserts in the set can be safely assumed as readable.

We define a system's upper bound on how long a single insert can remain inconsistent as ϕ. We refer to the range of time between a query initiation and ϕ units of time before the query initiation as the *uncertain past*. That is, the area in which initiated inserts have a chance, however remote, of being missed by the query. The range of time from ϕ to ∞ units of time before the initiation of a query is referred to as the *certain past*; the range in which initiated inserts are guaranteed to be committed and readable in time for the query.

Then, with knowledge of the system variable ϕ and the insertion rate f of a data stream, the maximum number of points in danger of being missed by the query, which we refer to as n_0 is:

$$n_0 = \lfloor \phi \cdot f \rfloor \tag{5.1}$$

Therefore, so long as the number of points in the insert stream n is $n \geq n_0$, increasing n has no impact on (t, k)-staleness.

Then, t essentially serves to decrease the range of the uncertain past, as t ensures that no inserts exist in the range of the past between the query initiation and t units of time prior to the query initiation. With larger values of t, we should expect the likelihood of freshness to increase. The same is true for f; larger gaps between inserts lead to fewer inserts in the uncertain past.

5.1.4 Staleness and Error

In key-value PBS, read operations are deemed fresh or stale solely by the number of writes missed by the read operation. Interestingly, in A-PBS, the number of missed insert operations is not the only means of qualifying query staleness. The relative

error of an aggregation query can also be used as an indicator of a query's staleness.

Before we may begin to reason about aggregate error, we must first describe what the *correct* answer of a query is, as well as what the *observed* value is.

Definition 8 (True aggregate value) *Given a data stream DATA(n, f, D), the true aggregate value of a query Q is the result of applying the function A to all inserts in the data stream covered by the query's bounding box.*

Definition 9 (Observed aggregate value) *Given a data stream DATA(n, f, D), the observed aggregate value of a query Q is the result of applying the function A to all inserts in the data stream covered by the query's bounding box currently readable by the query (that is, excluding any insertions which are in transit, and therefore not written to the data store).*

One immediate problem that arises when comparing the error of aggregation functions is the variation of the measure values associated with each point. It is reasonable to expect that one system may contain measure values that lie closely between, say 0 and 1, while another system may contain mostly measure values that lie in a much larger range, for example between $1,000$ and $10,000$. In this scenario, if error is defined as absolute (that is, simply the difference between the true value and the observed value), and the aggregation function being used is sum, the first system is likely to have a much smaller absolute error than the system with very large measure values, regardless of the state of consistency of the systems, as a missed point in the second system has a far greater impact on the error than even several missed points in the first system. Because of this, absolute error can be a poor indicator of result quality when comparing aggregate results, especially in cases where the stream size or measure distribution is not known.

Since we would like to examine error in a way that is not closely tied to specific distributions of measure values, we present the following definition of aggregate relative error:

Definition 10 (Aggregate relative error) *The aggregate relative error of a query Q with an observed aggregate value of o and a true aggregate value of v is $\frac{|o-v|}{v}$.*

By dividing the absolute error $|o - v|$ by the true aggregation value v, we obtain a relative view of the error. Using relative error, we can compare the numerical results from aggregation functions across multiple measure distributions or different functions.

With this in mind, we describe a method, much like (t, k)-staleness, of classifying the result of a query of being acceptably consistent, with respect to the relative error:

Definition 11 ((t, ϵ)-staleness) *Given an insert data stream DATA(n, f, D), a query Q with aggregation function A, initiated t units of time after the last insert in the stream has been sent, has (t, ϵ)-staleness if and only if the relative error of A is greater than ϵ.*

(t, ϵ)-*staleness* essentially places an upper bound ϵ on the relative error of a query. A query whose relative error is less than or equal to ϵ is said to have an acceptable amount of error, in which case the query is acceptably consistent (similar to k in (t, k)-staleness). Where (t, k)-staleness measures staleness depending on whether or not points are present during the time the aggregation takes place, (t, ϵ)-staleness measures staleness based on the result of the aggregation. Because of this, (t, ϵ)-staleness is dependent on the aggregation function A used by the query, as well as the distribution D of measure values in the data stream.

Another important difference is that points missed by a query only impact (t, ϵ)-staleness if the missed point has an impact on the final aggregation. For example, missing a single point will not likely have an impact on (t, ϵ)-staleness where the aggregation function used is *max*, as that point would have to be the largest point covered by the query in order to impact the result of the aggregation function. Conversely, missing a single point where the aggregation function is *count* is guaranteed to increase the relative error of the query.

5.1.5 Probabilistic Staleness

So far, most of the work presented in this chapter has been based in examining how staleness impacts queries on an individual basis. While this is a useful way to begin to reason about staleness in aggregate systems, it only comments on individual queries within a system, not a system itself. Since we would like to be able to comment on a system's accuracy or staleness on a whole, we now apply both staleness metrics

introduced in the chapter on a systems-based level. This can be done by applying (t, k)-staleness and (t, ϵ)-staleness within a probabilistic framework as follows:

Definition 12 (Bounded (t, k)-staleness) *A system for aggregate queries on an input data stream DATA(n, f, D) has bounded (t, k)-staleness with probability p if and only if, with probability p, an aggregate query Q with coverage C does not have (t, k)-staleness.*

Definition 13 (Bounded (t, ϵ)-staleness) *A system for aggregate queries on an input data stream DATA(n, f, D) has bounded (t, ϵ)-staleness for probability p if and only if, with probability p, an aggregate query Q with coverage C and aggregation function A does not have (t, ϵ)-staleness.*

Using *bounded (t, k)-staleness* and *bounded (t, ϵ)-staleness*, the probability of a system being unacceptably inconsistent (determined by k or ϵ), can be described. For example, consider a toy aggregate data store, where each inserted point has, while in the uncertain past, a 5% chance of being missed by a query. In this example system, the chance of missing a query in the uncertain past remains constant regardless of how far it is from the time of query. Although, in actual systems, we may expect the chance of missing an insert in the uncertain past to decrease as the insert initiation time approaches the certain past. If the system has a ϕ value of 0.1 seconds, what is bounded (t, k)-staleness with $k = 0, t = 10$ms on a sufficiently large data stream with an insert frequency of 1000 items per second?

As the system is relatively simple, this can be evaluated through closed form analysis. Solving Equation 5.1 yields the number of points required for the data stream to be sufficiently large (that is, the value of n_0):

$$\lfloor \frac{0.1}{1/1000} \rfloor = 100 \tag{5.2}$$

Therefore the data stream DATA(100, 1000, D) is sufficiently large to model $t = 10$ms bounded (t, k)-staleness for arbitrarily large data streams, as any additional points in the stream are guaranteed to be readable.

Since 100 points are in the uncertain past, each with a 5% chance of being missed, the probability of missing no more than $k = 0$ insertions is 0.95^{100}, or 0.00592.

Therefore, the p value for bounded (t, k)-staleness with $k = 0, t = 10$ms on this system is 0.00592. While bounded (t, k)-staleness with a probability of less than 0.01 indicates that the system has very poor consistency, this example demonstrates the impact the number of insertions in the uncertain past has on staleness. While each insert in the uncertain past has a deceptively high chance of being observed by the query, the chance of each one being observed is much lower.

5.2 Evaluating A-PBS for Generic Aggregate Systems

In this section, we make modifications to the non-replicated aggregate data store model presented in Section 4.1.2 and describe a Monte Carlo simulation wherein probabilities for bounded staleness can be calculated. We do the same for the quorum-based replication model in Section 4.1.3, in order to calculate bounded staleness probabilities under quorum replication.

5.2.1 An Enhanced Aggregate Model for A-PBS Simulation

In order to apply staleness analysis to our distributed aggregate data store model presented in Section 4.1.2, we must first describe the ways in which the system state can become "dirty", or "desynchronized", of which there are exactly two. Recall, as shown in Figure 4.3, that a distributed aggregate system has a three level structure consisting of clients, indices and buckets. A single insertion may desynchronize either data in the index or bucket, or both, but only the data required for reading that specific insert; an insert A cannot affect the synchronization status of a different insert B. Desynchronizations result in individual inserts not being readable until data is eventually synchronized. While inserting can cause data desynchronization, queries cannot.

Essentially this means that inconsistency may be introduced to this generic model in two ways. An insert, when sent to an index, may invalidate or update data on that specific node. Once an insert is routed to a bucket, said insert may also invalidate part of the data in the bucket as well. To keep the model as generic as possible, we refrain from describing any ways in which data within indices and buckets become desynchronized and instead describe the probability of an insert introducing metadata

desynchronization to an index as δ, and the probability of an insert introducing data desynchronization to a bucket as θ.

Furthermore, since indices and buckets are eventually consistent, we assign Δ and Θ to be the distributions of time in which desynchronization introduced by individual inserts is corrected. It follows then that ϕ is the largest value from either Δ or Θ.

In this model, any query with a box that happens to contain any desynchronized points on the index or buckets will essentially behave as if the relevant points do not exist in the system.

With this, we feel the model is simple enough to be easily applied to general distributed real-time OLAP systems, while still accurately reflecting the presence of staleness in the system. However, the problem of evaluating the probabilities of bounded (t, k)-staleness and bounded (t, ϵ)-staleness within the model remain. It is our belief that, with the number of variables and distributions at work in the model, any closed form analysis of staleness in this model is impractical. In the section that follows, we discuss the application of a Monte Carlo simulation which emulates inserts and queries according the model.

Name	Type	Description
δ	Real	Probability of insert introducing index desync
θ	Real	Probability of insert introducing bucket desync
Δ	Distribution	Distribution of time taken for index desync correction
Θ	Distribution	Distribution of time taken for bucket desync correction

Table 5.1: The list of system parameters in the model

5.2.2 Simulation

In order to view how the model's parameters affect staleness, a method for evaluating the probability value p behind bounded (t, k)-staleness and bounded (t, ϵ)-staleness for varying values of each parameter in Table 5.1 is needed. As mentioned earlier, the relative complexity of the model makes deriving a closed form analysis unappealing. Instead, we opt to approximate the probability of bounded (t, k)-staleness and bounded (t, ϵ)-staleness with a Monte Carlo simulation. Essentially, a Monte Carlo simulation simulates several "trials" of a stochastic system and generates a final result from the results these trials. In our case, each trial will be several simulated inserts, followed

by a simulated query. The result of each trial will be whether or not the query has (t, k)-staleness or (t, ϵ)-staleness. Then, from a set of trial results, probabilities for bounded staleness can be determined.

As just mentioned, each trial is composed of a stream of simulated inserts, followed by a single query. The number and rate of simulated insertions is determined by a data stream DATA(n, f, D). To minimize computation, n_0 is used for all values of $n > n_0$, as setting $n > n_0$ does not effect bounded (t, k)-staleness and only positively effects bounded (t, ϵ)-staleness (relative errors grow smaller).

Each simulated insert operates as follows. First, the time the insert began (relative to the time of query) is stored and computed as $-(c * i + t)$, where c is the number of future inserts in the stream, and i is $1/f$, the time between inserts in the stream. Then, two random real numbers between $[0, 1]$ are generated by a random number generator and compared against δ and θ respectively. If the first number is greater than δ, then the insert is marked as having desynchronized the point at the index. If the second number is greater than θ, then the insert is marked as having desynchronized the point at the bucket. If the insert is marked as only having desynchronized the point at the index, then a "time until synchronized" value is assigned by randomly sampling from Θ and adding it to $-(c * i + t)$, the time the insert began. The same applies if the insert is marked as only having desynchronized the bucket, only instead sampling from Δ. If the insert desynchronized both the index and the bucket, the "time until synchronized" value is assigned the largest value from sampling Θ once, and sampling Δ once and adding it to $-(c * i + t)$.

Once a stream of inserts has been simulated, simulation of the query is trivial. The "time until synchronized" value of each insert is scanned. For simplicity, we assume queries begin immediately after they are initiated, therefore the "time until synchronized" value essentially counts the units of time until the query reads the insert data. Any value greater than 0 means that the insert was synchronized after the query, and that those specific inserts were not observed by the query. Then, the error of the query can be evaluated simply by comparing the aggregation of the measure values with and without the missed points to determine whether or not the query has (t, ϵ)-staleness. Similarly, the total number of missed points can be used to determine whether the query has (t, k)-staleness.

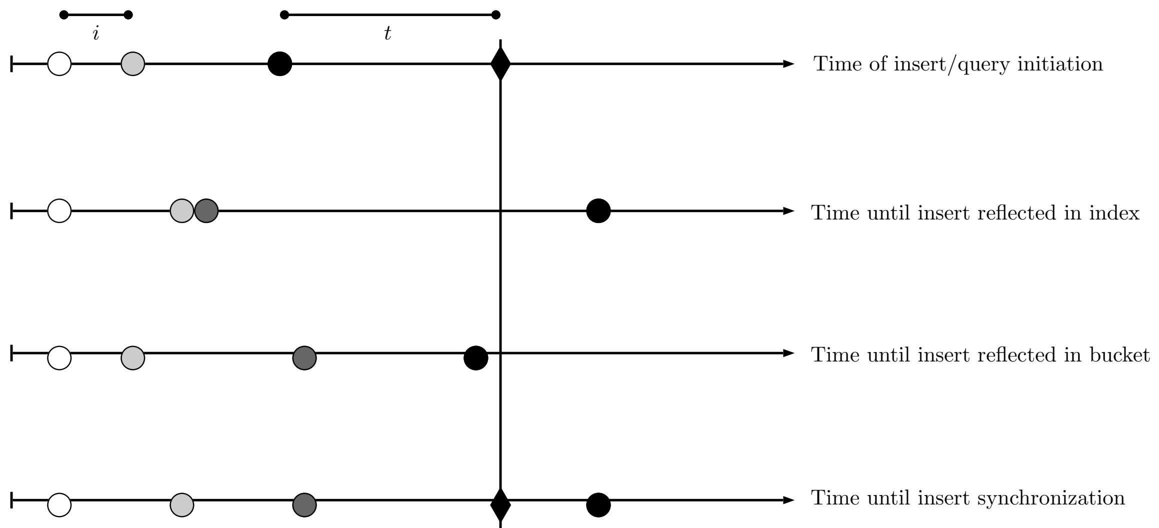


Figure 5.5: A simulation of a single trial of an insert stream and a query. The last insert, represented as a black circle, is synchronized after the query, represented as the black vertical line, so this simulated query is considered to have $(t, k{=}0)$-staleness.

Figure 5.5 presents a graphical view of a single trial. Each color-coded circle on the top line represents the time an insert in the stream was initiated. The circles on the middle two horizontal lines represent the time the insert was synchronized on the index and bucket. The positions of the inserts on these two lines are determined by δ and Δ for the index line, and θ and Θ for the bucket line. The bottom horizontal line represents the time each insert was synchronized on both the index and the bucket. Any inserts to the right of the time of query on the bottom horizontal line are inserts which have synchronized after the query. In this case, the last insert, represented as the dark black circle, was missed by the query. If $k = 0$, this simulated query would be considered to have (t, k)-staleness. Note that, if the horizontal lines representing the time of index and bucket synchronization are removed, the figure essentially becomes Figure 5.3, the figure used earlier for describing (t, k)-staleness.

Algorithm 1 describes how individual trials of an insert stream are simulated for computing bounded (t, k)-staleness. Likewise, Algorithm 2 describes how queries are simulated given the insert simulation output from Algorithm 1. In essence, Algorithm 1 computes a list of the synchronization times of each insert in the uncertain past, while Algorithm 2 simulates the query by iterating through the list and determining if more than k inserts have been missed by the query. The simulated query returns a boolean

value, true if the query does not have (t, k)-staleness, false otherwise.

Adjusting this algorithm for (t, ϵ)-staleness is simple. For each simulated insert, sample from a provided distribution D and record the value for each insert. The simulated query can then compute the true and observed aggregate value by aggregating all points and all readable points, respectively. The relative error can be determined by the true and observed aggregate value, and whether or not a query has (t, ϵ)-staleness can be determined by checking if the relative error is greater than ϵ.

Algorithm 1 A single trial of an insert stream simulation that returns the list of synchronization times of all desynchronized points in the simulated stream.

```
1:  Let φ = max(Θ) + max(Δ)
2:  Let i = 1/f
3:  Let sync_times be an empty array
4:  for c = 0; c < n; c++ do
5:      Let sync_time = −(c ∗ i + t)
6:      Let index_desync be a uniform random number between [0, 1]
7:      Let bucket_desync be a uniform random number between [0, 1]
8:      if index_desync > δ and bucket_desync > θ then
9:          Let sample = max(sample(Δ), sample(Θ))
10:         Let sync_time = sample + sync_time
11:     else
12:         if index_desync > δ then
13:             Let sample = sample(Δ)
14:             Let sync_time = sample + sync_time
15:         end if
16:         if bucket_desync > θ then
17:             Let sample = sample(Θ)
18:             Let sync_time = sample + sync_time
19:         end if
20:     end if
21:     Append sync_time to sync_times
22: end for
23: return sync_times
```

Algorithm 2 A single trial of a simulated query, which computes if a query on the simulated insert stream does not have (t, k)-staleness

1: **Input**: a list of insert sync times, *sync_times*, from Algorithm 1
2: Let $missed = 0$
3: **for** *sync_time* in *sync_times* **do**
4:     **if** $sync_time > 0$ **then**
5:         $missed = missed + 1$
6:     **end if**
7: **end for**
8: **return** $(missed < k)$

Finally, in order to approximate the probabilities for bounded staleness, a number of trials are run (that is, a simulated insert stream followed by a simulated query). The number of trials where the simulated query does not have (t, k)-staleness or (t, ϵ)-staleness are observed. Note that a sufficiently large number of trials must be chosen, as the accuracy of the computed result is in part determined by the number of trials. One must simply divide the number of queries which did not have (t, k)-staleness or (t, ϵ)-staleness by the total number of trials to get the percentage of fresh queries, or, in other words, an approximation of the probability of a query not being bounded (t, k)-staleness or bounded (t, ϵ)-staleness within the specific instance of the model.

5.3 Evaluating A-PBS for Quorum-based Aggregate Systems

In this section, we apply the aggregate quorum theory in the previous chapter to our simulation. In doing so, we create a slightly more complex simulation which will enable the approximation of the probabilities of bounded (t, k)-staleness and bounded (t, ϵ)-staleness for aggregate quorum systems. A summary of the key system parameters required for simulation is given in Table 5.2.

5.3.1 Parameters

Recall the structure of an aggregate quorum-based data store shown in figure 4.4: b clients communicate with j indices, which in turn communicates with m bucket sets. Each bucket set is composed of N individual bucket replicas. All buckets within a

bucket set replicate the same area in d-dimensional space. We use the notation B_u^x to refer to the xth replica of bucket set u. Inserts sent from an index must wait for W replies from the relevant bucket set before signalling to the client the insert has completed. Likewise, queries sent from an index must wait for at least R replies from all relevant bucket sets before sending the result of the aggregation query to the client. We begin by adding the number of bucket sets m and the quorum parameters N, W and R to the simulation's list of system parameters.

Since the desynchronization on the bucket level is now defined by the interaction between quorum rules and the distributions of operation times for inserts and queries, the system probability of inconsistency being introduced to a bucket (θ), and the distribution of time taken for synchronization of data on a bucket (Θ) are no longer needed and are removed from the simulation's system parameters. Instead, the probability of bucket set desynchronization is essentially always 100%, as writes are very likely to commit at different bucket replicas at different times. The time taken for a bucket set to completely synchronize is now dependent on the network latencies of each bucket replica, as well as their distributions of time taken to commit the insert once received. Since our aggregate quorum model does not use quorum-based replication for indices, we keep the probability and distribution parameters for index desynchronization.

To better describe this, we define the following distributions and add them to the simulation's system parameters. $L(A \to B)$ is a distribution which describes the latency of sending a message from a node A to a node B. We make the assumption here that all query and insertion requests and replies have the same impact on network latency, and thus network latencies for any type of message (insert or query) sent within the system can be drawn from $L(A \to B)$, where A is the sending node and B is the receiving node. However, note that latencies may still vary depending on the sending node and the receiving node. For example, the distribution $L(I_1 \to B_1^1)$ may be completely different from the distribution $L(I_1 \to B_1^2)$, especially if B_1^1 and B_1^2 are located at different geographical locations.

$T_w(A)$ is a distribution which describes the time taken for the node A to complete the local computation required for an insert. For indices, this is the time taken to parse the incoming insertion message, plus the time taken to update its location metadata,

plus the time taken to determine the location of the bucket replicas that must be sent the insert. For buckets, this is the time taken to parse the incoming insertion message, commit the insert, and construct a reply that will be sent to the index. $T_w(A)$ does not include any time taken for network-based factors, such as sending or receiving data. Like the latency distributions, the distribution $T_w(A)$ varies on the input node A. Not only does this allow for the distributions to be different for indices and buckets, but it also allows the simulation to model indices and buckets with varying hardware specifications or geographical locations.

Similar to $T_w(A)$, $T_r(A)$ is a distribution which describes the time taken for a node A to complete the local computations required for a query. For indices, this includes the time taken to retrieve the list of all buckets which must be queried and the time taken to construct the message for each bucket. For buckets, this includes the time taken to compute the aggregation for the query and to construct the result message. We make a simplifying assumption here, which is that the box of the query and the aggregation function of the query do not have any impact on the distribution $T_r(A)$.

Since our calculation of ϕ for the previous simulation relied on the now removed Θ distribution, a new method of calculating ϕ for this model is required. In order to do this, we must consider the maximum amount of time an insert can remain uncommitted on at least one bucket from the time of initiation of the insert. If, for the sake of simplicity, we say that inserts sent from clients immediately arrive at indices, then the longest possible bucket insert committal time can be determined as follows. For each index I_u and bucket B_u^x pair in the system, determine the sum of the maximum latency and local write distributions required to write an insert to a bucket. Specifically, that is: $max(T_w(I_u)) + max(L(I_u \rightarrow B_u^x)) + max(T_w(B_u^x))$. If δ is not 0, then index synchronization must also be considered in our calculation of ϕ. We determine the maximum amount of time an index can remain desynchronized as $max(T_w(I_u)) + max(\Delta)$. The max is taken of both the maximum bucket synchronization time and the maximum index synchronization time, for each index bucket pair, to determine the value of ϕ for the system.

Name	Type	Description
j	Integer	Number of indices currently in the system
m	Integer	Number of bucket sets currently in the system
δ	Real	Probability of insert introducing index desync
Δ	Distribution	Distribution of time taken for index desync correction
N	Integer	Number of replicas per bucket set
W	Integer	Number of insert replies required until insert completion
R	Integer	Number of query replies required until query completion
$L(I \to B)$	Distribution	Dist. of time taken to send and queue message from I to B
$L(B \to I)$	Distribution	Dist. of time taken to send and queue message from B to I
$T_w(I)$	Distribution	Dist. of time taken for local insert work on an I
$T_w(B)$	Distribution	Dist. of time taken for local insert work on a B
$T_r(I)$	Distribution	Dist. of time taken for local query work on an I
$T_r(B)$	Distribution	Dist. of time taken for local query work on a B

Table 5.2: The set of system parameters in the new quorum-based model

5.3.2 Simulation

The act of determining the probability of bounded staleness of an aggregate quorum system by simulation is somewhat more involved than the previous model; however, the general approach is the same. A fixed number of trials which simulate an insert data stream followed by a query are executed. Each trial outputs only whether the query in that specific trial has (t, k)-staleness or (t, ϵ)-staleness. Once a fixed number of trials have been run, the proportion of trials which met the criteria over the total number of trials is used to estimate the probability that a query under the specified system parameters will obey the bounded freshness or error specified by the user.

Each trial begins with the computation of the number of inserts in the uncertain past of the stream, which can be trivially computed with Equation 5.1. A structure is also needed to keep track of important data for each simulated insert. For example, it was previously sufficient to simply only store the commit time and measure value of each insert. However, now that each insert writes to multiple buckets, the commit times of each bucket must be stored. For each insert in the uncertain past, the trial begins by computing the start time of the insert as $-(c*i+t)$, as it was in the previous simulation. Unlike the previous simulation, the bucket set number for the current insert is determined by randomly sampling from a uniform distribution and rounding down to the nearest integer. The bucket set number is relevant to the simulation,

as buckets have an impact on the combination of aggregation functions. The index synchronization time for the insert is simulated the same way as in the previous simulation, and is stored in our important insert information structure. From here, another loop is entered, where the committal time of the insert on each bucket in the bucket set is simulated. This is simply done by determining the duration of time until committal, which can be obtained by sampling $L(I \rightarrow B_u^x)$, $T_w(I)$ and $T_w(B_u^x)$ and summing (where u is the bucket set number computed earlier, and x is the number of the current bucket replica). The bucket replica committal time is then determined by adding the duration to the start time. Furthermore, the quorum reply time (the time at which the index has received the write reply from the bucket) is computed by adding a sample from $L(B_u^x \rightarrow I)$ to the bucket replica committal time. This value, along with the bucket replica committal time, bucket set and replica number, u and x, are written to the important insert information structure for each replica. At the end of this entire process, an insert information structure will be computed for each insert in the uncertain past, which is then used for query simulation. This entire process is summed up in Figure 5.6.

Once the insert stream of a trial has been simulated, the results of the stream are then passed on to the query simulation part of the trial. The trial begins by determining the earliest time the query can be initiated while still observing write quorum rules and the value of t. To do this, the time of the Wth earliest quorum reply time for each insert is retrieved, and the maximum value of all of them is taken and recorded as the earliest time all write quorums have met at least their minimum requirements (that is, at least W replies for each insert). This time is then offset by t to get the time of query initiation. This value is added to a random sample of $T_r(I)$, to get the time at which the index has done its local bucket location lookup. Then, for each bucket B_u^x in the system, the time of query arrival at bucket B_u^x is recorded by adding a sample of $L(I \rightarrow B_u^x)$ to the time the index has finished its local work. While the time of query arrival at each bucket is useful for determining freshness (as we also know the time at which each insert has committed on each bucket), further simulation is required to determine which buckets are the first R responders. To do this, for each bucket, the time of query arrival is offset by a random sampling from $T_r(B_u^x)$ and $L(B_u^x \rightarrow I)$. From this, the time R responses have been received from

each bucket set can be observed, and the maximum value is taken to get the time the query has met its read quorum rules for each bucket set. Figure 5.7 covers this whole process on a relatively simple system with $m = 1$, $N = 3$ and $R = 1$.

From here, the query simulation groups all buckets into two sets: those which have responded to the query before or during R responses have been received from each bucket set, and those which have not. Since those that have not responded in time are not included in the result of the query, they may be safely ignored. The buckets which have responded in time will simulate an aggregation operation on all inserts they contain (within the query's coverage), with the exception of points that were committed locally later than the time of query arrival on the current bucket and points which have an index synchronization time later than the time of query initiation. When evaluating (t, ϵ)-staleness, the aggregation function provided should be used, while when evaluating (t, k)-staleness, the *count* aggregation function should be used. Afterwards, the partial aggregation of each bucket is combined by bucket set using the combination rules described earlier in Section 4.1.5. Then, all combined partial aggregations are aggregated into a single value to get the observed aggregation value. This process of aggregation is repeated, except without exclusion of any inserts, regardless of committal time, to compute the true aggregation value, the result of the aggregation we would expect to get under a perfectly consistent system. With the true value and the observed value, whether or not the query has (t, ϵ)-staleness can be determined by evaluating the error between the true and observed value. Likewise, (t, k)-staleness can be determined by comparing the observed and true counts in relation to the value of k.

5.4 Evaluation

In this section, we implement the aggregate quorum simulation presented in this section to create a program for approximating the probabilities of bounded (t, k)-staleness and bounded (t, ϵ)-staleness for a set of varying parameters.

5.4.1 Configuration

We begin with a short description of a set of values which will be used as the default simulation parameters, unless otherwise specified, for the following set of experiments.

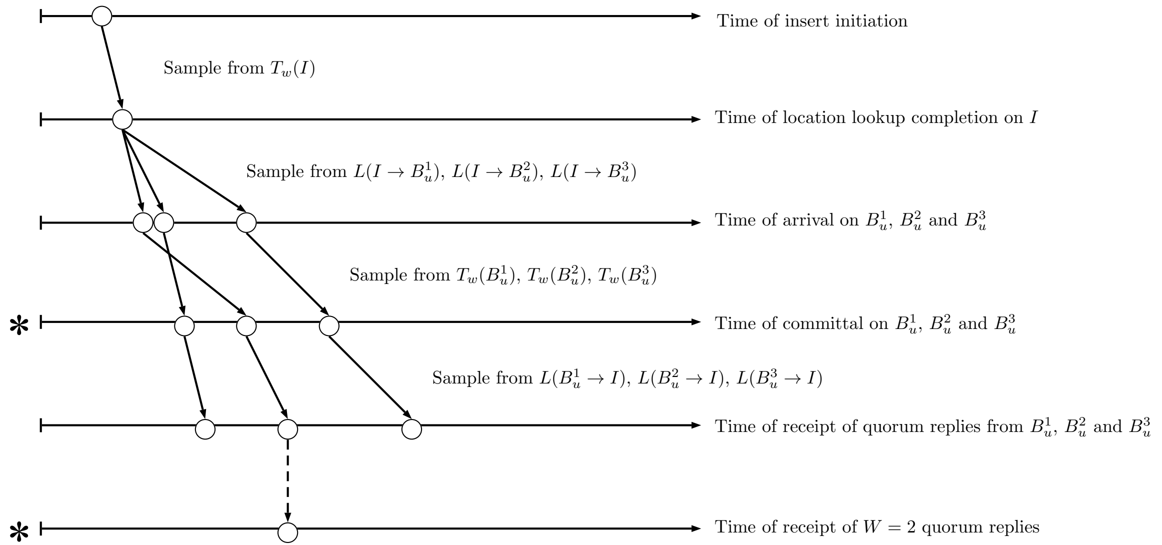


Figure 5.6: A simulation of a single insert with $N = 3$ and $W = 2$. After sampling various distributions, the key times marked by asterisks are generated for later computation during query simulation.

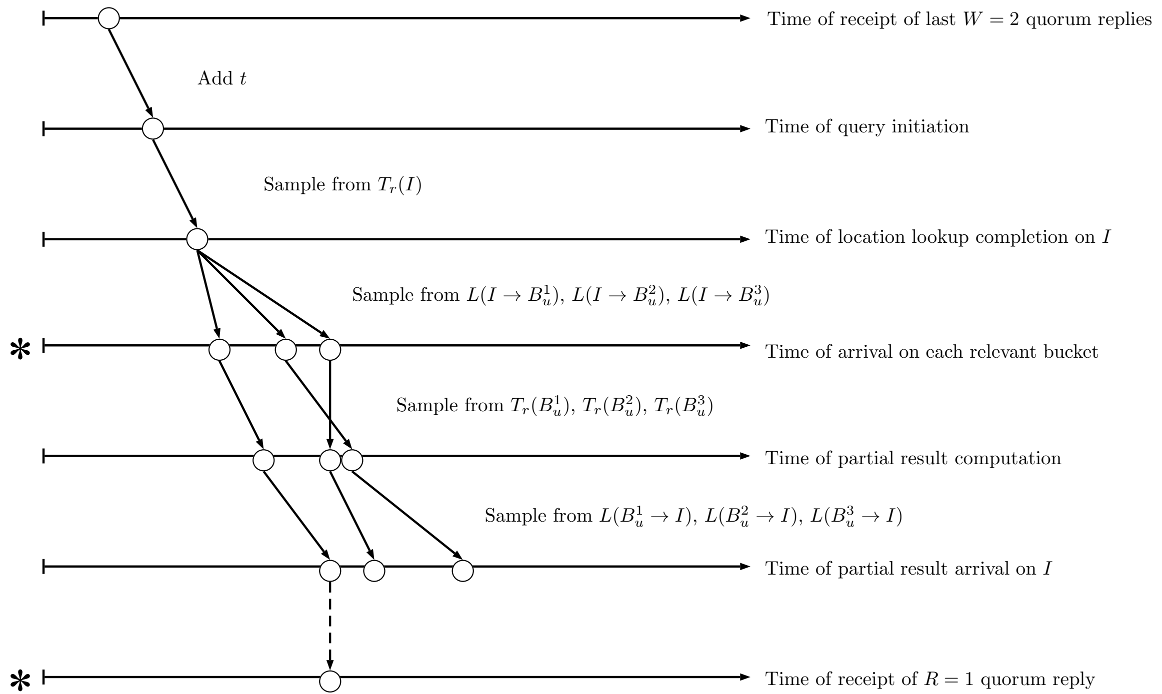


Figure 5.7: A simulation of a single query on a system with $m = 1$, $N = 3$, $R = 1$. Key times are marked with asterisks. After sampling various distributions, the time at which each bucket receives the simulated query request is recorded, as well as the specific buckets that make up the first R responses.

Since the purpose of this section is to explore the impact different system parameters have on consistency, and not to analyze staleness for any specific system, all parameters here are selected to illustrate only the dynamics of the simulation and model.

Unless otherwise stated, N is set to 3 in this configuration, since it is the norm for storing highly available data in key-value quorum systems, as noted in [22]. The read and write quorums W and R are both set to 1, which, above all other non zero values of W and R, prioritizes latency over consistency. Additional quorum configurations, however, are also explored in this section. The total number of bucket sets in the system, m, is 128, similar to the number of subsets used for a typical *vOLAP* system. The coverage C for each query is set to 50%, although other coverages are also explored. The insertion rate f is $20,000$ inserts per second, which may correspond to a *vOLAP* system under medium load. The probability of index desynchronization occurring, δ, is set to 0, as is typically observed in *vOLAP*. Consequently, the distribution of index synchronization times, Δ, does not need to be defined.

The latency distributions for sending messages across the network and reading or writing local data, that is, the parameters $L(I \rightarrow B)$, $L(B \rightarrow I)$, $T_w(I)$, $T_w(B)$, $T_r(I)$ and $T_r(B)$, are all exponential distributions, measured in milliseconds, with $\lambda = 0.01$. Exponential distributions are used here, as we expect their long tails to emulate the effects of blocking and queuing encountered in a distributed multi-core environment. Each distribution uses the same λ value to model a "neutral" system, where insert and query operations take the same amount of time. Figure 5.8 illustrates the latency distributions used by their cumulative distribution functions. Since the tail of the distribution is infinite, and therefore the maximum value from each distribution is infinity, we slightly compromise our formula for evaluating ϕ. Instead of summing the max value from $T_w(I)$, $L(I \rightarrow B)$ and $T_w(B)$, we instead sum the 99th percentile of the three distributions. Doing this, we get a ϕ of approximately $460.52 * 3$, or 1381.56 milliseconds. Thus, the number of inserts required for the insert stream to be sufficiently large is $20,000 * 1.38156 = 27,631$.

Name	Description	Value
N	Number of replicas	3
W	Write quorum	1
R	Read quorum	1
m	Number of bucket sets	128
δ	Probability of index desync.	0
C	Query coverage	50%
f	Inserts per second	20,000
n	Number of inserts in stream	27631
$L(I \to B)$	Index to bucket network distribution	Exponential, $\lambda = 0.01$
$L(B \to I)$	Bucket to index network distribution	Exponential, $\lambda = 0.01$
$T_w I$	Index write distribution	Exponential, $\lambda = 0.01$
$T_w B$	Bucket write distribution	Exponential, $\lambda = 0.01$
$T_r I$	Index read distribution	Exponential, $\lambda = 0.01$
$T_r B$	Bucket read distribution	Exponential, $\lambda = 0.01$

Table 5.3: The set of default system and simulation parameters used for the experiments in Section 5.4

5.4.2 Bounded Staleness

The first set of experiments explores the effect different quorum configurations have on bounded (t, k)-staleness. Figure 5.9 illustrates the probability of bounded (t, k)-staleness for $k = 0$ as a function of t for several different settings of N, W and R. Each point on the graph corresponds to a single simulation composed of 1,000 trials. `[N=3, W=1, R=1]` yields the lowest probability of bounded (t, k)-staleness, as it has the largest number of replicas and the smallest read and write quorums of the set. The probability of consistency increases as the number of replicas is dropped to 2, as seen in `[N=2, W=1, R=1]`, because each insert is guaranteed to be written locally on at least 50% of replicas instead of 33%. Increasing either the read or write quorum by one has a greater positive impact on consistency than decreasing the replica count from 3 to 2. Both quorum configurations `[N=3, W=2, R=1]` and `[N=3, W=1, R=2]` offer a greater chance of consistency compared to `[N=2, W=1, R=1]`. Because aggregate quorum read operations require at least R responses from each bucket set, it is likely given our current latency distributions that more than R replies are received from some replicas by the time at least R responses are received from all replicas. Therefore, increasing the read quorum R by one essentially increases the likelihood that N responses have been received from some bucket sets. This property is unique to aggregate quorum

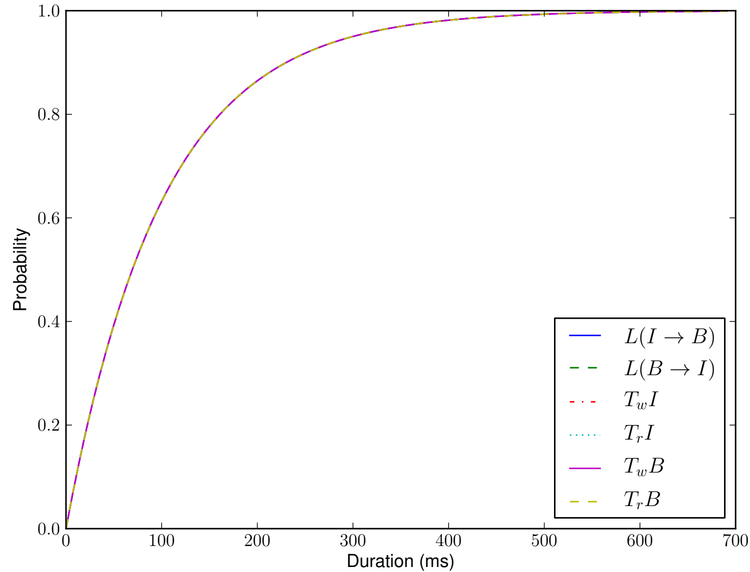


Figure 5.8: Plot of the cumulative distribution functions for the latency distributions used in Section 5.4. Since each latency distribution is the same, all lines overlap each other.

systems, and can make read replies, depending on the system, innately more valuable than write replies, in terms of consistency for partial quorums. This can be observed in the difference between the probabilities for `[N=3, W=2, R=1]` and `[N=3, W=1, R=2]`.

Very close to 100% probability of bounded (t, k)-staleness, but importantly not 100%, is `[N=2, W=1, R=2]`. In key-value quorum systems, `[N=2, W=1, R=2]` constitutes a complete, intersecting quorum, and should thus always be consistent. However, for aggregate quorum systems, this property does not hold. While having a quorum configuration of `[N=2, W=1, R=2]` ensures that all writes are included in the set of partial aggregations received on the index, the combination of partial aggregations from replicas forces the selection of two incorrect partial aggregations. This effect is demonstrated in detail in Section 4.1.5.

The quorum configurations `[N=1, W=1, R=1]`, `[N=2, W=2, R=1]` and `[N=3, W=2, R=2]` lie on the upper edge of the curve. Since `[N=1, W=1, R=1]` provides no replication, it is perfectly consistent, and therefore has 100% probability of bounded (t, k)-staleness. `[N=2, W=2, R=1]` is a complete aggregate quorum, and obeys perfect consistency, as $W = N$ write replies ensures that each insert is written to all replicas

before the query begins. `[N=3, W=2, R=2]` lies extremely close to the 100% probability mark, but does not have a write quorum of $W = N$ and is therefore not a complete quorum or perfectly consistent.

Plots for the slightly more lenient cases of bounded (t, k)-staleness where $k = 1$ and $k = 2$ are presented in Figures 5.10 and 5.11. When $k = 1$, all quorum configurations with non-perfect consistency reach near 100% consistency at $t = 300ms$, about half the time required for $k = 0$. Setting $k = 2$ increases consistency still, reaching nearly 100% probability of all queries being acceptably consistent for all configurations at $t = 200ms$.

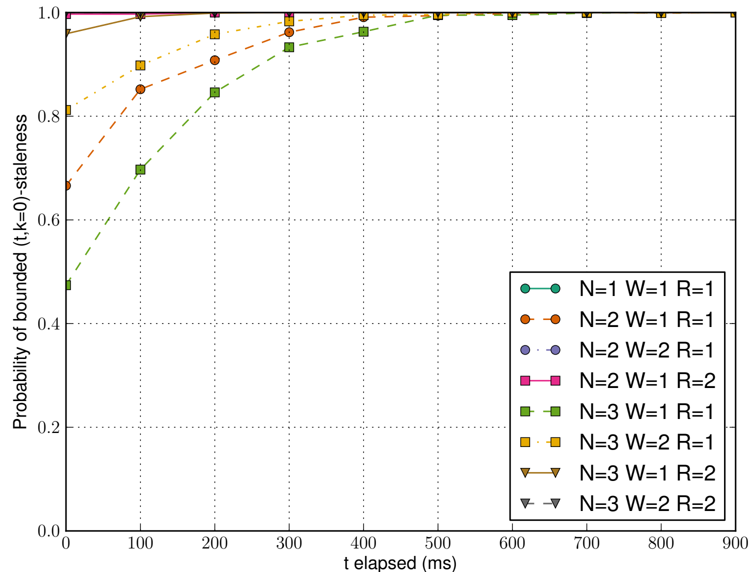


Figure 5.9: Probability of bounded $(t, k{=}0)$-staleness with varying quorum configurations across increasing values of t.

Figure 5.12 shows the probability of bounded $(t, k{=}0)$-staleness for varying coverages C. In the simulation, inserts within the data stream are considered relevant to the simulation with probability C. Essentially, provided the number of inserts in the stream is large, this results in approximately C percent of inserts being relevant to the query. Then, as expected, 100% coverage yields the worst probability of bounded (t, k)-staleness, while the lowest coverage in the plot, 25%, yields the greatest probability.

Figure 5.13 illustrates the average number of missed points in a query using the

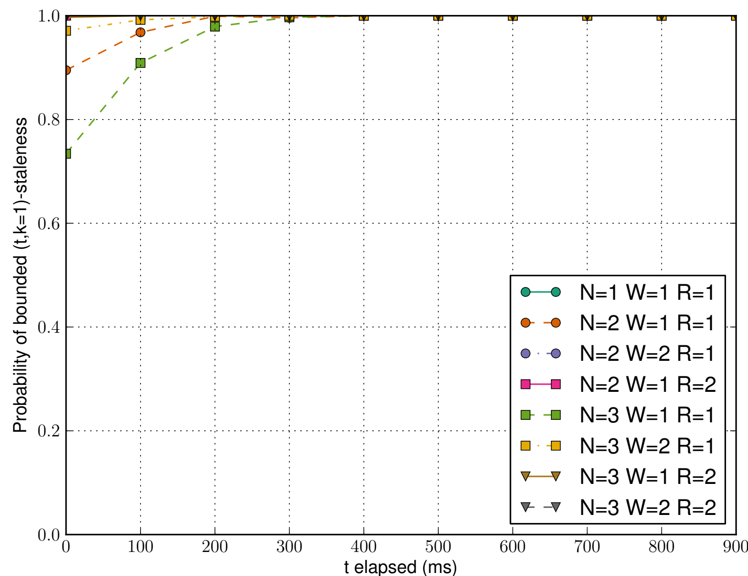


Figure 5.10: Probability of bounded $(t, k{=}1)$-staleness with varying quorum configurations across increasing values of t.

same parameters as the previous experiment. It is important to note that the worst case, quorum configuration `[N=3, W=1, R=1]` with $t = 0ms$, misses only one point on average, despite having a relatively poor probability of bounded $(t, k{=}0)$-staleness. Taking into account that queries in aggregate systems typically aggregate many thousands of points, it seems unlikely, at least for `[N=3, W=1, R=1]`, that a query which does not obey $(t, k{=}0)$-staleness has a noticeable difference in result from a query which does.

5.4.3 Latency

Figure 5.14 describes the average latency of inserts with varying quorum configurations within the simulation. Here, we observe that by comparing quorum configurations `[N=1, W=1, R=1]`, `[N=2, W=1, R=1]` and `[N=3, W=1, R=1]`, insert latency decreases as the replication factor N increases. This is because an insert operation's latency is determined by the insert latency of the fastest W of N replicas, and increasing N therefore allows for more samples from the insert latency distribution. This is a reasonable conclusion in cases where compute resources within the system are relative to N, and therefore the increased load of replication is mitigated. It follows then that

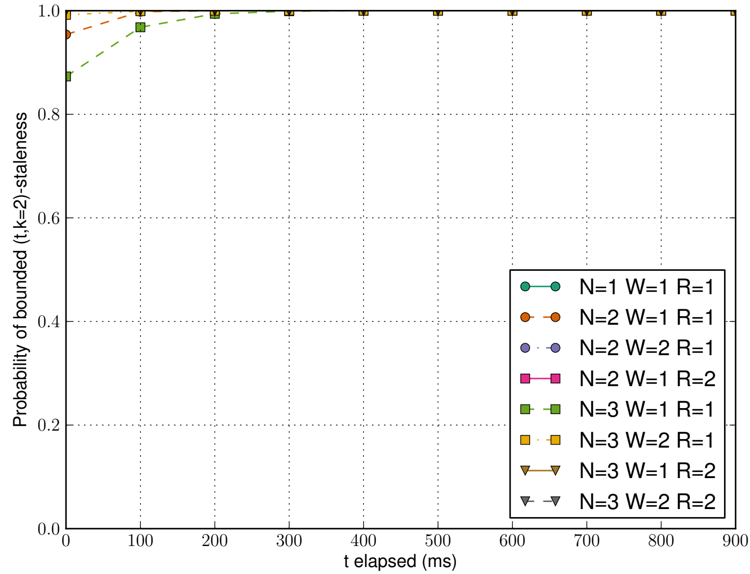


Figure 5.11: Probability of bounded $(t, k{=}2)$-staleness with varying quorum configurations across increasing values of t.

increasing the value of W has a negative impact on latency. Both of these traits are also present in key-value quorum replication.

In Figure 5.15, we see the average latency of a query with varying quorum configurations. The effects on query latency for different quorum configurations is similar to that of insert latency; greater values of N decrease latency, as we wait for only the first R replies from each bucket set. Likewise, greater values of R increase latency. One important distinction which differentiates query latency from insert latency are their comparative latencies. Since queries in this specific simulation must wait for a reply from at least $m * R$ different replicas, queries have significantly greater latencies from writes on comparable configurations, with differences close to 400 milliseconds.

5.4.4 Bounded Error

In our evaluation of bounded error, we first describe two distributions to be used for our measure distribution parameter D. An exponential distribution with $\lambda = 1$ is used to emulate the case where the measure data distribution is long-tailed. A folded normal distribution with $\mu = 0, \sigma = 1$, is used to emulate the case where measure data is short tailed. Both distributions yield only non-negative continuous values in order

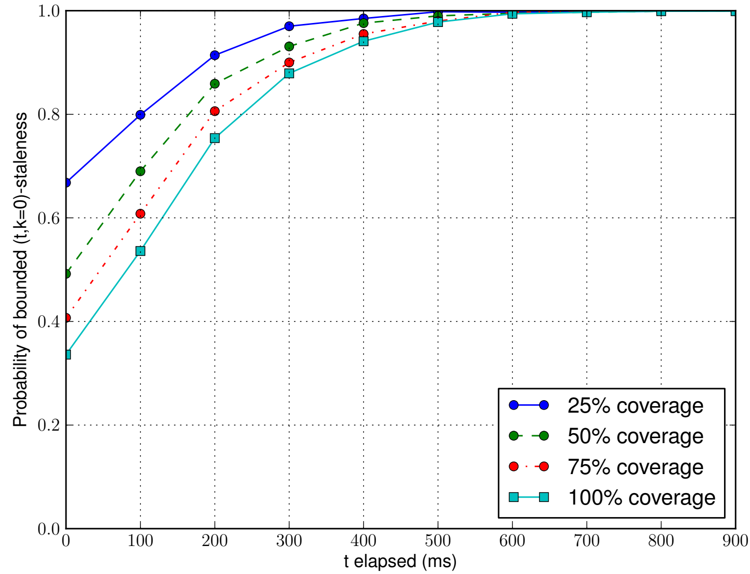


Figure 5.12: Probability of bounded $(t, k{=}0)$-staleness for 25%, 50%, 75% and 100% coverage.

to ensure the monotonicity of *sum*. Figure 5.16 shows the cumulative distribution function of the two distributions.

Additionally, for the following experiments, we select a quorum configuration of `[N=3, W=1, R=1]`, as it has the weakest consistency guarantees of the other configurations mentioned in this section. We explore the relative error of *sum*, as it is a monotonically increasing function whose rate of change is steady regardless of the number of points (similar to *count*), *max*, as its value is essentially determined by a single point (similar to *min*), and *mean*, as it is a non-monotonic aggregation functions whose rate of change drops as the number of points in the aggregation increases (similar to *stdev*).

In Figure 5.17, the probability of bounded $(t, \epsilon{=}0)$-staleness for the aggregation functions *sum*, *max* and *mean* and aforementioned distributions is illustrated. This is somewhat similar to the probability of bounded $(t, k{=}0)$-staleness from Figure 5.9, in that there is no tolerance for error. However, where bounded (t, k)-staleness views error as missing any points within the query, bounded (t, ϵ)-staleness views error in terms of the relative error of the observed and true aggregate results. Because of this, the *max* aggregation function easily achieves near 100% probability of bounded

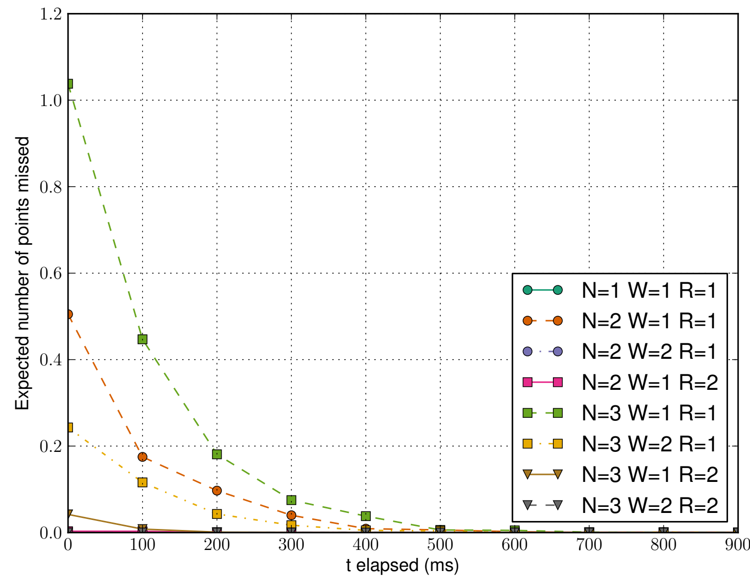


Figure 5.13: Expected number of missed inserts in a query with 50% coverage for varying quorum configurations.

$(t, \epsilon{=}0)$-staleness, despite the lower probability of bounded $(t, k{=}0)$-staleness of the system as a whole. For example, for both measure distributions, the *max* aggregation function has approximately a 100% probability of bounded $(t, \epsilon{=}0)$-staleness, as missing a point in a query only ever has an impact on the final aggregation if its measure value is the largest in the query's coverage. Conversely, missing a single point with a positive value as a measure for the *sum* aggregation function, or any point at all for the *mean* aggregation functions will result in a non-zero relative error. It is because of this that the curve for *sum* and *mean* in Figure 5.17 precisely match the `[N=3, W=1, R=1]` curve for bounded $(t, k{=}0)$-staleness in Figure 5.9.

In Figure 5.18, ϵ is increased to 0.0001, that is, a small amount of slack is given for relative error in bounded (t, ϵ)-staleness. This allows us to better see the impact measure distribution and aggregation function have on bounded (t, ϵ)-staleness. As expected, *max* retains its near 100% probability of bounded (t, ϵ)-staleness. The probability for *mean* and *sum*, however, have now somewhat spread apart from each other, with *mean* yielding better probabilities than *sum*. This is likely because the *mean* function is relatively insensitive to missing values, especially as the number of points in the data stream increases. For *sum*, the exponential distribution, when

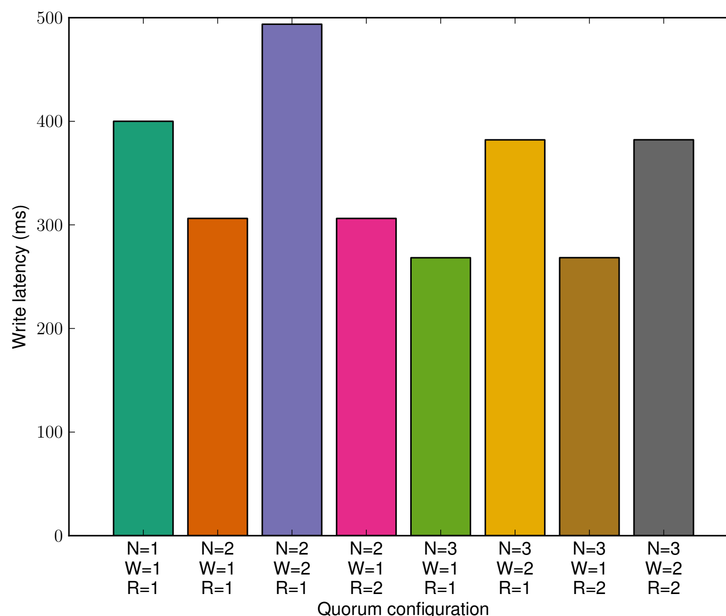


Figure 5.14: Average latency of an insert operation for varying quorum configurations

compared to the folded normal distribution, has negligibly improved probability of bounded (t, ϵ)-staleness, while the exponential distribution on *mean* offers a noticeably poorer probability than the folded distribution.

Figures 5.19, 5.20 and 5.21 show the bounded (t, ϵ)-staleness error for larger values of ϵ, with each curve reaching near 100% probability at $\epsilon = 0.0010$. The average relative error is shown in Figure 5.22.

5.4.5 Data Stream Size

In Figure 5.23, the relationship between data stream size and bounded (t, ϵ)-staleness, where $t = 0$, $\epsilon = 0.0001$ is illustrated. As the number of inserts in the stream increases along the x-axis, we observe the probability of bounded (t, ϵ)-staleness decreases, since all inserts in a stream where $n < n_0$ have a chance of being missed. Because the sequence of inserts in a stream are spaced $1/t$ units of time apart, increasing n adds more points to the start of the stream, which have a lower chance of being missed compared to points at the end of the stream. As n increases, the probability of missing a point near the start of the stream becomes negligible, until $n \geq n_0$, where the probability becomes 0. A little before 10,000 inserts, we observe that the probability

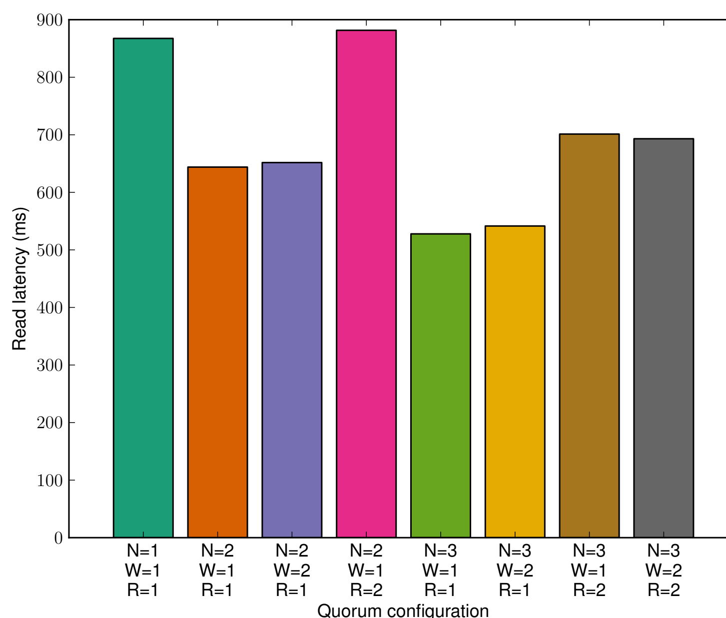


Figure 5.15: Average latency of a query operation for varying quorum configurations

of bounded (t, ϵ)-staleness begins to slope upward in the opposite direction, signifying that any additional inserts to the stream have a very small probability to be missed; the cost of an insert being missed is offset by the benefit the insert has in reducing the impact of other missed inserts on relative error. The dotted blue line represents the point at which the data stream is sufficiently large (n_0). At approximately 300,000 inserts, the probability of bounded (t, ϵ)-staleness is near 100%.

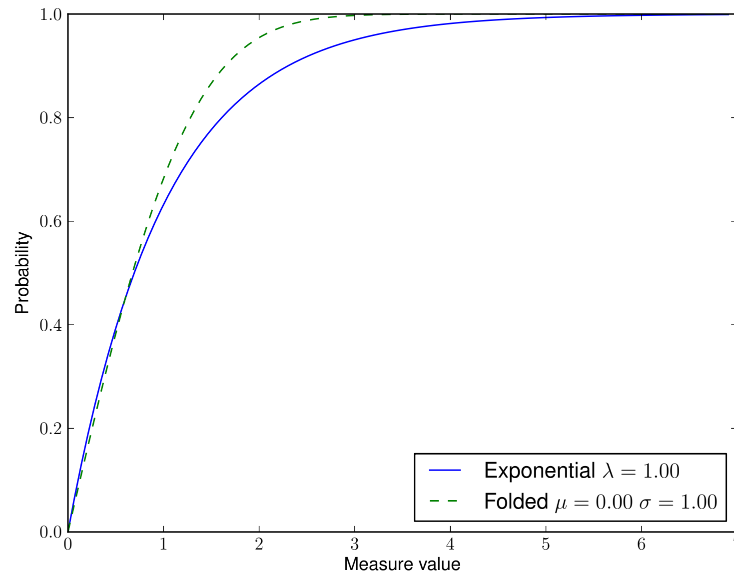


Figure 5.16: Plot of the cumulative distribution functions for the measure distributions used for bounded (t, ϵ)-staleness experiments.

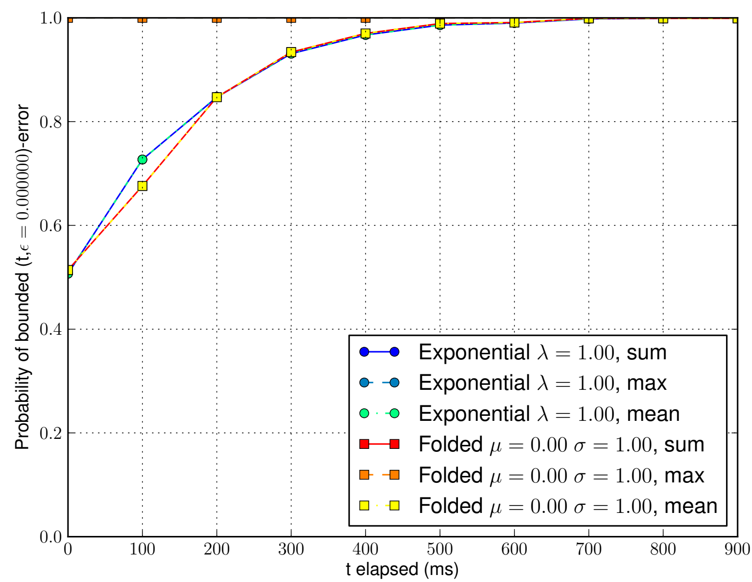


Figure 5.17: Probability of bounded $(t, \epsilon$=0.00000)-staleness with varying aggregation functions and measure distributions for `[N=3, W=1, R=1]` across increasing values of t.

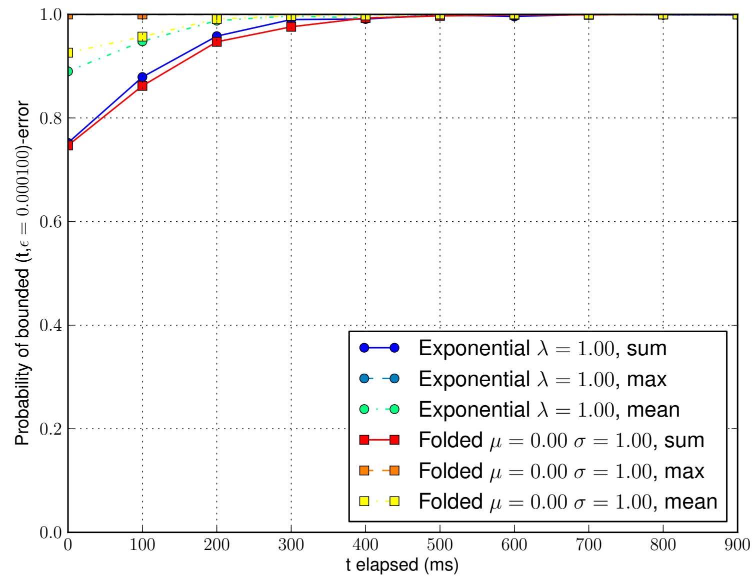


Figure 5.18: Probability of bounded (t, ϵ=0.00010)-staleness with varying aggregation functions and measure distributions for `[N=3, W=1, R=1]` across increasing values of t.

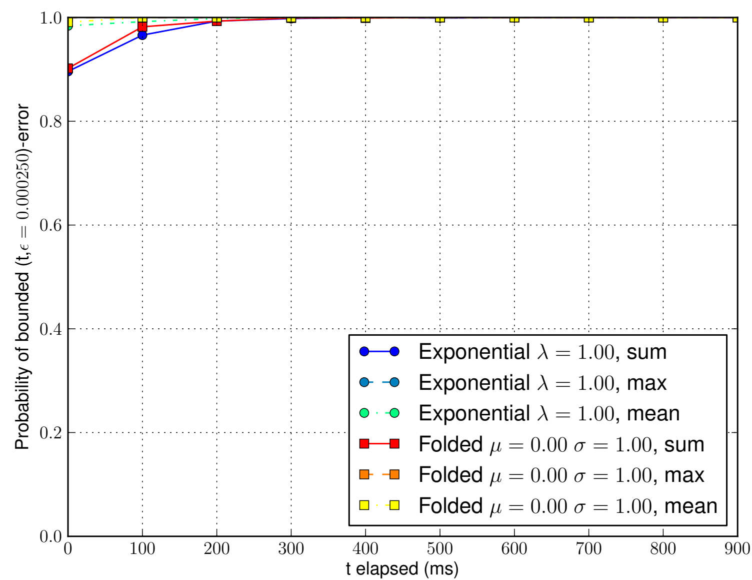


Figure 5.19: Probability of bounded (t, ϵ=0.00025)-staleness with varying aggregation functions and measure distributions for `[N=3, W=1, R=1]` across increasing values of t.

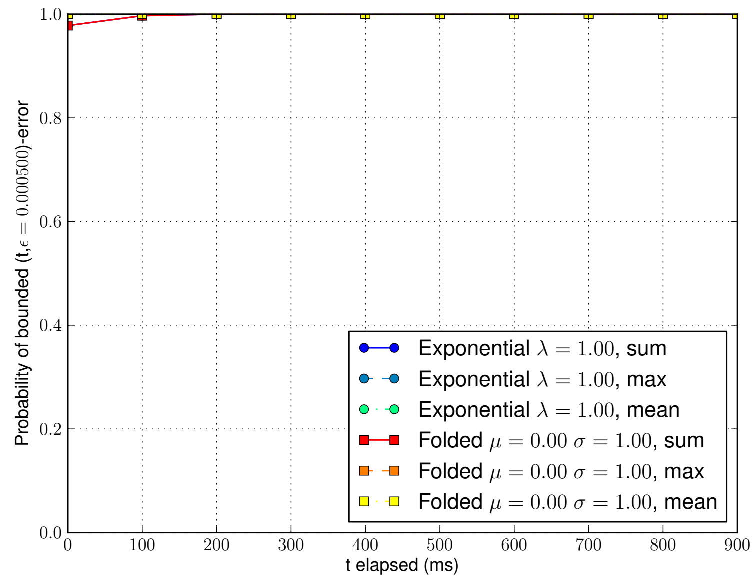


Figure 5.20: Probability of bounded (t, ϵ=0.00050)-staleness with varying aggregation functions and measure distributions for `[N=3, W=1, R=1]` across increasing values of t.

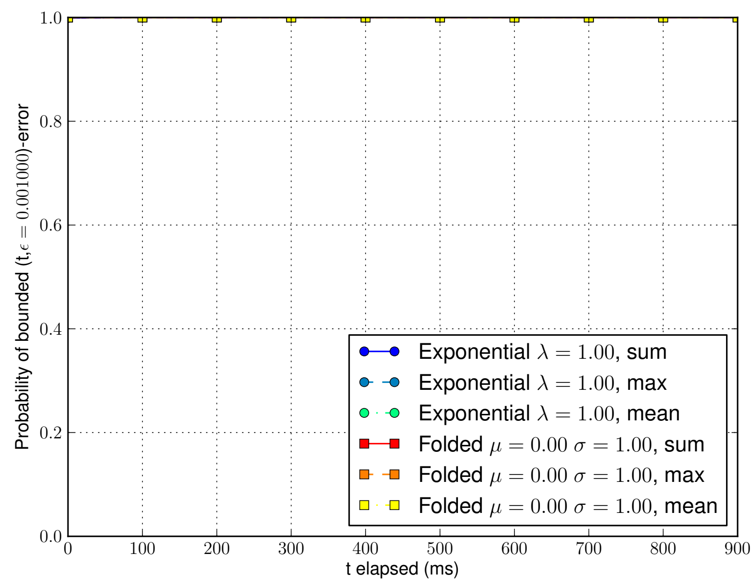


Figure 5.21: Probability of bounded (t, ϵ=0.00100)-staleness with varying aggregation functions and measure distributions for `[N=3, W=1, R=1]` across increasing values of t.

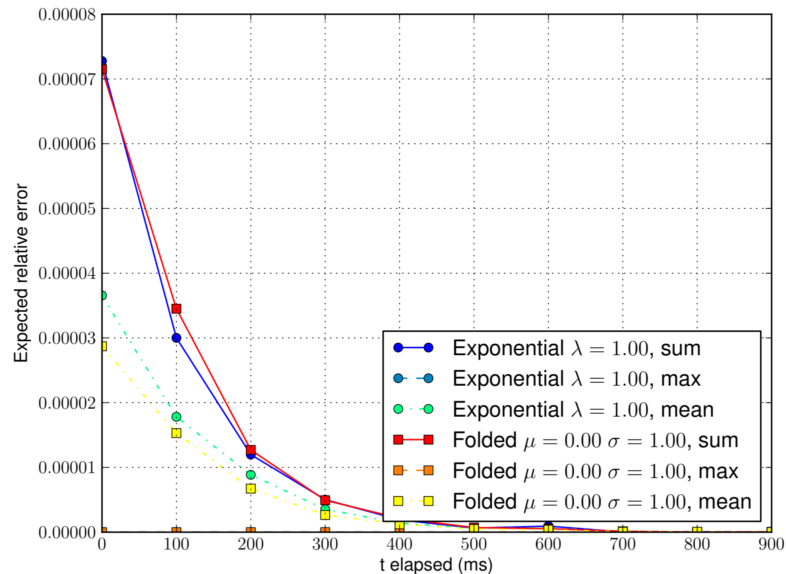


Figure 5.22: Expected relative error for `[N=3, W=1, R=1]` and varying aggregation functions and measure distributions across increasing values of t.

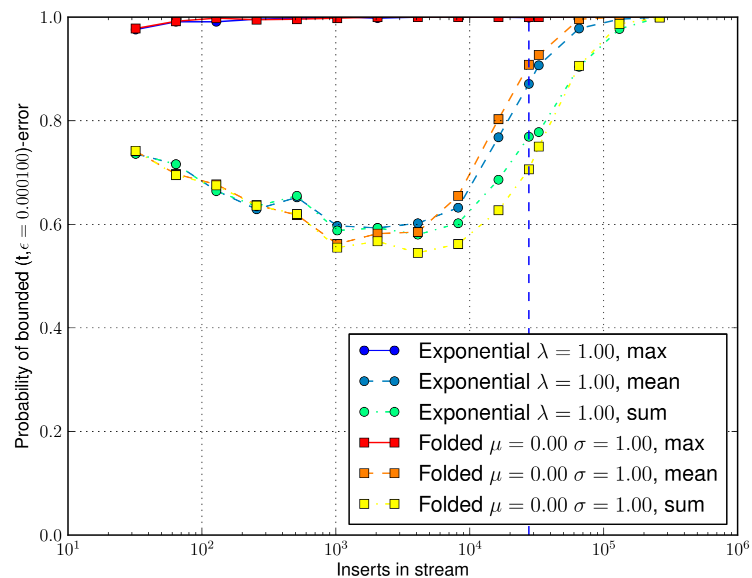


Figure 5.23: Probability of bounded (t, ϵ)-staleness with $t = 0ms$, $\epsilon = 0.00010$ and varying aggregation functions and measure distributions for `[N=3, W=1, R=1]` across increasingly larger insert streams (log scale). The dotted vertical line represents the point at which the stream is said to be sufficient large (for our 99th percentile calculation of ϕ).

Chapter 6

Conclusion

6.1 Summary

In this thesis we explored elasticity, availability and consistency for distributed cloud-based real-time OLAP systems. First, *vOLAP* was moved from a simple cluster-based deployment environment to a managed cloud, with support for automatic elastic provisioning of new resources. In addition, we proposed models and algorithms for applying quorum-based replication to distributed aggregate data stores. We began by constructing a simple, generic model for a distributed aggregate store, and modified it to support replication and quorum inserts and queries. The models and algorithms were then applied to add quorum-based replication to *vOLAP*. *vOLAP* with quorum replication was evaluated on an Amazon EC2 cloud. With a replication factor of 3, *vOLAP* was able to answer 20,000 complex range queries per second, and ingest inserts at a rate of 80,000 per second.

Inspired by PBS [22], we introduced Aggregate Probabilistically Bounded Staleness (A-PBS), a means of analyzing consistency in aggregate data stores. Within A-PBS, methods for examining consistency from the perspective of missed inserts, as well as numerical error of aggregate queries, were presented. A method of simulation was also described, which when given a list of system parameters, allows for the determination of various staleness metrics. For arbitrarily large data streams, A-PBS analysis demonstrated that a query with `[N=3, W=1, R=1]` that covers 50% of points in the stream has approximately a 50% probability of being perfectly consistent. A-PBS analysis also demonstrated that the same query, while often not perfectly consistent, has on average a negligible amount of relative error.

6.2 Future Work

We leave the following topics for future work.

***vOLAP* A-PBS analysis**: By monitoring network and computation latencies from *vOLAP*, simulation distributions for $L(A \rightarrow B)$, $T_w(A)$ and $T_r(A)$ can be determined. Using said latencies, the A-PBS simulation can be used to approximate probabilities for bounded (t, k)-staleness and bounded (t, ϵ)-staleness on *vOLAP*. A *vOLAP* experiment wherein the staleness of a set of queries is determined could also be implemented to determine probabilities for bounded (t, k)-staleness and bounded (t, ϵ)-staleness, and subsequently cross-validate the values generated by simulation.

Impact of geographical distribution of nodes in *vOLAP* and A-PBS: Experimental evaluation of *vOLAP* explored the system's performance while distributed in a single data center. Likewise, analysis of A-PBS presented in this thesis only examined staleness under a system with uniform latency distributions $L(A \rightarrow B)$ across all nodes. We expect operational latency as well as staleness to change when nodes are geographically spaced apart, but not to what extent. Running *vOLAP* on a system composed of nodes from different locations may yield additional insight into the interactions within aggregate quorum systems. A-PBS analysis with distributions obtained from aforementioned experiments would also be useful for further staleness analysis.

Automatic load-based scale-up in *vOLAP*: While elastic-scale up operations are automated in *vOLAP*, they are only initiated by benchmarking scripts at regular intervals. For *vOLAP* to be truly elastic, the manager should provision new nodes depending on worker load and data usage.

Anti-entropy in *vOLAP*: Under normal circumstances, *vOLAP* is an eventually consistent data store. However, there are cases where the data stored in subset replicas can permanently drift. For example, a simple momentary network failure can cause a single replica to never receive an insert other replicas have committed. Since failures like these are common in practice, it is unacceptable for a production system to never correct such missed inserts. Anti-entropy, a method where replicas periodically communicate with each other to correct inconsistencies, is popularly used in eventually consistent systems to rectify this problem [75, 34, 57]. Dynamo [34], for example, periodically passes Merkle trees [66], a hash-tree where each node is the hash of its children, of its key-value data amongst replicas. This allows replicas to quickly determine any inconsistencies between each other and correct them. Exploitation of

vOLAP's underlying tree structure could be used to introduce anti-entropy to *vOLAP* at little computational cost.

Bibliography

[1] All about Chef - Chef docs. https://docs.chef.io/.

[2] AWS — Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting. https://aws.amazon.com/ec2/.

[3] BigQuery. http://developers.google.com/bigquery/.

[4] boto 2.38.0 : Python Package Index. https://pypi.python.org/pypi/boto.

[5] Hadoop. http://hadoop.apache.org/.

[6] OpenStack Open Source Cloud Computing Software. http://www.openstack.org/.

[7] Puppet Labs Documentation. http://docs.puppetlabs.com/.

[8] python-novaclient 2.28.1 : Python Package Index. https://pypi.python.org/pypi/python-novaclient.

[9] Riak NoSQL database. http://docs.basho.com.

[10] SaltStack. https://docs.saltstack.com/en/latest/.

[11] Saltstack Walk-through. https://docs.saltstack.com/en/latest/topics/tutorials/walkthrough.html.

[12] Transaction processing performance council, TPC-DS (decision support) benchmark. http://www.tpc.org.

[13] Twitter storm. http://storm-project.net/.

[14] ZeroMQ socket library as a concurrency framework. http://www.zeromq.org/.

[15] Netezza architecture - IBM redbooks. http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf, 2010.

[16] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB*, 2(1):922–933, 2009.

[17] Ittai Abraham and Dahlia Malkhi. Probabilistic quorums for dynamic systems. In *Distributed Computing*, pages 60–74. Springer, 2003.

[18] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 232–243. IEEE, 1997.

[19] Amitanand Aiyer, Lorenzo Alvisi, and Rida A Bazzi. On the availability of non-strict quorum systems. In *Distributed Computing*, pages 48–62. Springer, 2005.

[20] Athanasia Asiki, Dimitrios Tsoumakos, and Nectarios Koziris. Distributing and searching concept hierarchies: an adaptive DHT-based system. *Cluster Computing*, 13:257–276, 2010.

[21] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[22] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[23] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, 23(2):279–302, 2014.

[24] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[25] R Bruckner, Beate List, and Josef Schiefer. Striving towards near real-time data integration for data warehouses. *Proc. DaWaK*, pages 173–182, 2002.

[26] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. ES^2: a cloud data storage system for supporting both OLTP and OLAP. In *Proc. ICDE*, pages 291–302, 2011.

[27] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[28] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: a distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[30] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[31] Chun Chen, Gang Chen, Dawei Jiang, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Providing scalable database services on the cloud. In *Proc. WISE*, pages 1–19. 2010.

[32] Yanpei Chen, Vern Paxson, and Randy H Katz. Whats new about cloud computing security. *University of California, Berkeley Report No. UCB/EECS-2010-5 January*, 20(2010):2010–5, 2010.

[33] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Review*, 41(6):205–220, 2007.

[35] F Dehne, Q Kong, A Rau-Chaplin, H Zaboli, and R Zhou. A distributed tree data structure for real-time OLAP on cloud architecture. In *Proc. IEEE Big Data*, 2013.

[36] Frank Dehne and Hamdireza Zaboli. Parallel real-time OLAP on multi-core processors. In *Proc. CCGRID*, pages 588–594, 2012.

[37] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[38] Katerina Doka, Dimitrios Tsoumakos, and Nectarios Koziris. Brown Dwarf: a fully-distributed, fault-tolerant data warehousing system. *J. Par. Distr. Comp.*, 71(11):1434–1446, 2011.

[39] Katerina Doka, Dimitrios Tsoumakos, and Nectarios Koziris. Online querying of d-dimensional hierarchies. *J. Par. Distr. Comp.*, 71(3):424–437, 2011.

[40] Patrícia Takako Endo, Glauco Estácio Gonçalves, Judith Kelner, and Djamel Sadok. A survey on open-source cloud computing solutions. In *Brazilian Symposium on Computer Networks and Distributed Systems*, 2010.

[41] Martin Ester, Jörn Kohlhammer, and H-P Kriegel. The DC-tree: a fully dynamic index structure for data warehouses. In *Proc. ICDE*, pages 379–388, 2000.

[42] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[43] David K Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.

[44] David Kenneth Gifford. *Information Storage in a Decentralized Computer System*. PhD thesis, Stanford, CA, USA, 1981.

[45] Seth Gilbert and Nancy Ann Lynch. Perspectives on the CAP theorem. Institute of Electrical and Electronics Engineers, 2012.

[46] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[47] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[48] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, (4):68–74, 1997.

[49] Hyuck Han, Young Choon Lee, Seungmi Choi, Heon Y Yeom, and Albert Y Zomaya. Cloud-aware processing of MapReduce-based OLAP applications. In *Proc. Australasian Sym. on Par. Distr. Comp.*, pages 31–38, 2013.

[50] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques*. Elsevier, 2006.

[51] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems (TOCS)*, 4(1):32–53, 1986.

[52] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX*, volume 8, pages 11–11, 2010.

[53] Dong Jin, Tatsuo Tsuji, and Ken Higuchi. An Incremental Maintenance Scheme of Data Cubes and Its Evaluation. *Proc. DASFAA*, pages 36–48, 2008.

[54] Balachandra Reddy Kandukuri, V Ramakrishna Paturi, and Atanu Rakshit. Cloud security issues. In *Services Computing, 2009. SCC'09. IEEE International Conference on*, pages 517–520. IEEE, 2009.

[55] Quan Kong. Scalable real-time OLAP systems for the cloud. 2014.

[56] Mehmet Can Kurt and Gagan Agrawal. A fault-tolerant environment for large-scale query processing. In *Proc. HiPC*, pages 1–10, 2012.

[57] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Review*, 44(2):35–40, 2010.

[58] Neal Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[59] Hans-J Lenz and Bernhard Thalheim. OLAP databases and aggregation functions. In *ssdbm*, page 0091. IEEE, 2001.

[60] Jiyuan Li, Frank Z Wang, Lingkui Meng, Wen Zhang, and Yang Cai. A map-reduce-enabled SOLAP cube for large-scale remotely sensed data aggregation. *Computers & Geosciences*, 2014.

[61] Jun Luo, Jean-Pierre Hubaux, and Patrick Th Eugster. Pan: Providing reliable storage in mobile ad hoc networks with probabilistic quorum systems. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, pages 1–12. ACM, 2003.

[62] Dahlia Malkhi, Michael Reiter, and Rebecca Wright. Probabilistic quorum systems. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 267–273. ACM, 1997.

[63] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud security and privacy: an enterprise perspective on risks and compliance.* " O'Reilly Media, Inc.", 2009.

[64] Peter Mell and Tim Grance. The NIST definition of cloud computing. 2011.

[65] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB*, pages 330–339, 2010.

[66] Ralph C Merkle. A certified digital signature. In *Advances in Cryptology-CRYPTO89 Proceedings*, pages 218–238. Springer, 1990.

[67] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud computing: an overview. In *Cloud Computing*, pages 626–631. Springer, 2009.

[68] Sumant Ramgovind, Mariki M Eloff, and Elme Smith. The management of security in cloud computing. In *Information Security for South Africa (ISSA), 2010*, pages 1–7. IEEE, 2010.

[69] R J Santos and Jorge Bernardino. Real-time data warehouse loading methodology. *Proc. IDEAS*, pages 49–58, 2008.

[70] Ricardo Jorge Santos and Jorge Bernardino. Optimizing data warehouse loading procedures for enabling useful-time data warehousing. *Proc. IDEAS*, pages 292–299, 2009.

[71] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: shrinking the PetaCube. In *Proc. ACM SIGMOD*, pages 464–475, 2002.

[72] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems.* Prentice-Hall, 2007.

[73] Doug Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[74] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB*, pages 1626–1629, 2009.

[75] Robbert Van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 6. ACM, 2008.

[76] Panos Vassiliadis. Modeling multidimensional databases, cubes and cube operations. In *Scientific and Statistical Database Management, 1998. Proceedings. Tenth International Conference on*, pages 53–62. IEEE, 1998.

[77] Panos Vassiliadis and Timos Sellis. A survey of logical models for OLAP databases. *ACM Sigmod Record*, 28(4):64–69, 1999.

[78] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[79] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proc. ACM SIGMOD*, pages 591–602, 2010.

[80] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient B-tree based indexing for cloud data processing. In *Proc. VLDB*, pages 1207–1218, 2010.

[81] F Yang, E Tschetter, X Leaute, N Ray, G Merlino, and D Ganguli. Druid: A real-time analytical data store. In *Proc. ACM SIGMOD*, 2014.

[82] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An efficient multi-dimensional index for cloud data management. In *Proc. Int. W. Cloud Data Management*, pages 17–24, 2009.