UNIVERSITY OF CALIFORNIA
Santa Barbara

Data and Application Management in an Open Cloud Platform

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

by

Navraj Chohan

Committee in Charge:

Professor Chandra Krintz, Chair

Professor Divy Agrawal

Professor P. Michael Melliar-Smith

Professor Louise Moser

December 2012
UCSB Technical Report 2012-01

The Dissertation of
Navraj Chohan is approved:

Professor Divy Agrawal

Professor P. Michael Melliar-Smith

Professor Louise Moser

Professor Chandra Krintz, Committee Chairperson

December 2012

Data and Application Management in an Open Cloud Platform

UCSB Technical Report 2012-01

by

Navraj Chohan

Dedication and Gratitude

I would like to dedicate this dissertation to my beloved family: my mother, Kamal Kaur, my father, Harminder Chohan, my sister, Navneet, and my brother, Jovan. They have supported me throughout my life and continue to do so with unconditional love. To my late grandfather, Darshan Chohan, who said, ”Navraj is going to be a doctor” while I was still a child, I want to say I did it, and while it was probably not the kind of doctor you meant it’s still pretty close.

Chandra Krintz has been my adviser and mentor since my undergraduate days, and I would not be where I am today without all of her support, encouragement, and guidance.

I would like to thank Divy Agrawal, P. Michael Melliar-Smith, and Louise Moser for serving on my Ph.D. committee.

I am grateful to Chris Bunch for being my good friend and collaborator throughout my graduate career. I would like to thank Rich Wolski, Selim Gurun, and Ye Wen for their mentorship during my early graduate career.

Lastly, my fellow graduate students from the RaceLab have made my stay most enjoyable, and for this I am deeply grateful.

Acknowledgements

The contents of Chapters 3–8 is a partial reprint of the content as it appears in the conference proceedings listed below.

Chapter 3: Publication [22] in the ICST International Conference on Cloud Computing (CloudComp 2009) and Book Chapter [55] in Open Source Cloud Computing Systems

Chapter 4: Publication [11] in the IEEE International Conference on Cloud Computing (IEEE CLOUD 2010), Publication [20] in the IEEE International Conference on Cloud Computing (IEEE CLOUD 2011), and Publication [19] in Journal of GRID Computing.

Chapter 6: Publication [24] in the USENIX Conference on Web Applications (WebApps 2012).

Chapter 7: Publication [23] in the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud2010).

Chapter 8: Publication [25] in the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud2012).

Curriculum Vitæ

Navraj Chohan

Education

2012	**Doctor of Philosophy in Computer Engineering,**
	University of California, Santa Barbara.
2008	**Master of Science in Computer Engineering,**
	University of California, Santa Barbara.
2005	**Bachelor of Science in Computer Engineering,**
	University of California, Santa Barbara.

Experience

2006 – 2012	**Graduate Research Assistant,**
	University of California, Santa Barbara.
2012 – present	**Co-Founder,**
	AppScale Systems, Santa Barbara, CA.
2011	**Research Intern,**
	Lawrence Livermore National Lab, Livermore, CA.
2010	**Research Intern,**
	IBM, T.J. Watson Research Center, Hawthrone, NY.

2005 – 2006 **Software Engineer,**

Applied Signal Technologies (Acquired by Raytheon), Sunnyvale, CA.

Publications

C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hegde, and A. Srivastava, A Pluggable Autoscaling Service for Open Cloud PaaS Systems, IEEE/ACM International Conference on Utility and Cloud Computing, to appear: November, 2012.

Navraj Chohan et al. North by Northwest: Infrastructure Agnostic and Datastore Agnostic Live Migration of Private Cloud Platforms. *In the Proceedings of USENIX HotCloud (2012), Boston, MA, USA.*

Navraj Chohan et al. Hybrid Cloud Support for Large Scale Analytics and Web Processing. *In the Proceedings of USENIX WebApps (2012), Boston, MA, USA.*

C. Bunch, B. Drawart, N. Chohan, C. Krintz, L. Petzold, and K. Shams, Language and Runtime Support for Automatic Configuration and Deployment of Scientific Computing Software over Cloud Fabrics, Journal of Grid Computing, Special Issue on Data Intensive Computing in the Clouds, Mar, 2012.

C. Bunch , N. Chohan, and C. Krintz, Supporting Placement and Data Consistency Strategies Using Hybrid Clouds, IEEE Aerospace Conference, March, 2012.

C. Krintz, C. Bunch, and N. Chohan, AppScale: Open-Source Platform-As-A-Service, in Open Source Cloud Computing Systems: Practices and Paradigms, IGI Global, Luis Vaquero, Juan Ciceres, and Juan Hierro, Eds., ISBN-13: 978-1466600980, January, 2012.

Navraj Chohan et al. Datastore-Agnostic Transaction Support for Cloud Infrastructures. *In the Proceedings of IEEE Cloud (2011), Washington DC, USA.*

C. Bunch , N. Chohan, C. Krintz, and Khawaja Shams (JPL), Neptune: A Domain Specic Language for Deploying HPC Software on Cloud Platforms, ACM ScienceCloud Workshop, June, 2011

Chris Bunch, Navraj Chohan, et. al. Neptune: A Domain Specific Language for Deploying HPC Software on Cloud Platforms. *In the Proceedings of ACM ScienceCloud (2011), San Jose, CA, USA.*

Navraj Chohan, et al. See Spot Run: Using Spot Instances for MapReduce Workflows. *In the Proceedings of Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2010), Boston, MA, USA.*

Chris Bunch, Navraj Chohan, et. al. Key-Value Datastore Comparison in AppScale. *In the Proceedings of IEEE Cloud (2010), Miami, Florida, USA.*

Navraj Chohan, et al. AppScale: Scalable and Open AppEngine Application Development and Deployment. *In the Proceedings of CloudComp (2009), Munich Germany.*

Ye Wen, Wei Zhang, Rich Wolski, and Navraj Chohan. Simulation-Based Augmented Reality for Sensor Network Development. *In the Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys 2007), Sydney, Australia.*

Ye Wen, Selim Gurun, Navraj Chohan, Rich Wolski and Chandra Krintz. Accurate and Scalable Simulation of Network of Heterogeneous Sensor Devices. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, 2007.*

Ye Wen, S. Gurun , N. Chohan , R. Wolski and C.Krintz. Full-System, Cycle-Close Simulation of the Stargate Sensor Network Intermediate Node. *In the Proceedings of the IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS-VI 2006), Samos, Greece.*

Field of Study: Computer Engineering

Abstract

Data and Application Management in an Open Cloud Platform

Navraj Chohan

Cloud computing has had tremendous uptake in the global market and is expected to grow well into the future. The commoditization of computational and storage resources has given massive capabilities to individuals and companies to acquire such resources on demand, and to relinquish them when no longer required, without the need to budget for additional hardware and management.

Platform-as-a-Service (PaaS) architectures have arisen in the past years to alleviate the burdens of resource management for developers who may now focus strictly on application development. This faster time-to-value has increased productivity for both developers and their respective organizations. Developers no longer have to worry about lower level details such as CPU consumption, bandwidth limitations, memory consumption, and disk usage, as it has been common in the past. The scaling of applications is now the burden of the platform system. PaaS systems have become the operating systems of the datacenter.

Our research has been focused on developing a PaaS system which can give the aforementioned attributes in an open and pluggable way. We emulate the Google App

Engine PaaS system as it was one of the first to come to market and offered the promise of infinite scalability at the front end of application servers and the backend of large data storage, all powered by Google's robust infrastructure.

We call our PaaS solution AppScale. AppScale is an open cloud platform capable of transparently executing Google App Engine applications at scale and without modification. AppScale is a cloud-based web framework which provides multiple services that provide cloud infrastructure control, data persistence, caching and a number of other common application technologies. AppScale both simplifies and facilitates the benchmarking of the execution of scalable cloud technologies using real applications.

This Ph.D. thesis discusses the design, implementation, and evaluation of AppScale. It considers the many components of AppScale with a focus on the data management layer for scalable storage, transaction semantics, scalable queries, analysis of "Big Data", and live migration support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has revolutionized the means of which corporations, both large and small, maintain and operate their IT departments and infrastructure. The availability of compute and storage resources has increased tremendously for organizations as large cloud providers such as Amazon [2] and Rackspace [80] have servicized access of these resources via well defined APIs and web consoles. Individuals and corporations can now provide web services without the need to set up in-house resources. Customers can outsource their IT infrastructure to specialized cloud providers who have been able to consolidate and streamline processes to drive down costs, allowing them to focus on their core competencies.

Providers of cloud services make their resources available to the general public with certain characteristics which are indicative of cloud computing: scalability, elasticity, and fault tolerance. The amount of resources which can be attained on-demand are virtually unlimited. The on-demand nature of access to these resources gives users the

flexibility to grow and shrink based on their workloads–having them pay for only what they use. Faults to the underlying physical resources are generally hidden from users with access abstractions.

There are multiple layers in cloud computing, with many public and private commercial offerings throughout the cloud stack. The lowest level offering is Infrastructure-as-a-Service (IaaS) which provides virtualized machines, with Amazon Web Services being a prime example and an early leader in dictating the public IaaS market. Moreover, there is a market for private cloud implementations of IaaS for the self service of IT resources within an organization. There are many private cloud implementations of AWS's IaaS Elastic Compute Cloud (EC2) API including Eucalyptus, OpenStack, and CloudStack.

Additionally, Platform-as-a-Service, a higher level abstraction than IaaS, has many public and private offerings. Public offerings includes Heroku, Microsoft Azure, Google App Engine (GAE), Amazon Elastic Beanstalk, and many others, while private cloud PaaS offerings include AppScale, CloudFoundry, and OpenShift. These services and products abstract away the IaaS layer and provide a fully managed application stack which allows for simple deployment and ideally no maintenance for the developer.

The cost model for PaaS is based on metered usage with the capability to scale in and out as required by the workload. Customers of such a platform pay for the usage of APIs which interfaces with the infrastructure. Moreover, the provider will either

automatically scale up or down an application for any of the required services (i.e., GAE), allow the developer to dictate scaling rules (i.e., Azure), or give an easy-to-use web console for manual scaling (i.e., Heroku). The APIs for differing IaaS and PaaS offerings are similar in nature but are not easily interchangeable and readily portable.

Cloud architectures, IaaS and PaaS, have been designed and optimized for web services, or also referred to as Software-as-a-Service (SaaS), the most common application domain. This paradigm shifts the application hosting from the client (native binaries) to a remote service which is accessible via a web browser or through a REST interface.

As the popularity of REST and service oriented architecture (SOA) has risen, the support by high level languages has populated many offerings for web frameworks such as Ruby on Rails (RoR), Django for Python, and Node.js for JavaScript. Developers can use these mature frameworks which have a common set of reusable libraries on top of a service provider who then takes care of server maintenance, backup, and scaling, where scaling is a key concern due to the possiblity of attaing a large user base.

With the increase in internet usage and the proliferation of web enabled devices, online services such as Gmail, Facebook, and Salesforce.com see tremendous traffic– traffic which is logged in detail resulting in terabytes to petabytes of data. These logs provide insight on how users are utilizing the service and other business critical information such as fraud and anomaly detection.

Data sets at such sizes must be distributed across a collection of machines, and as the datasets grow, existing relational technologies have shown themselves to slow down to fulfill sophisticated SQL queries requiring a new set of technologies built for large data. NoSQL datastores have been adopted to handle these data sets, providing replication, fault tolerance, and high availability.

There are multiple public and private storage offerings for NoSQL. Amazon provides Simple Storage Service (S3), a highly scalable, highly reliable, and highly available key/value storage service. GAE offers Google's BigTable technology, a highly scalable column-oriented key/value store [16], as an API for web applications. Since the publication of the BigTable technology, many open source options have arisen including HBase, Hypertable, and Cassandra [45, 49, 14]. These open source technologies are currently deployed in companies such as Facebook, Reddit, and Baidu.

Many common relational features have been removed from NoSQL datastores to attain the feature set of high availability, high throughput, and fault tolerance. These features include full SQL and transaction support which can cause bottlenecks, high latency response times, and even system failure when dealing with extremely large datasets.

Because NoSQL datastores do not have a built-in fully expressive query language, additional technologies and adapters have been added to analyze data. The most prevalent technology is MapReduce, made popular by Google's paper in 2004 [30]. MapRe-

duce and open source implementations, such as Hadoop, are able to run mappers and reducers on a data set which spans across multiple machines while automatically handling faults. Higher level tools such as Pig and Hive [74, 67] were created to abstract away the need to write mappers and reducers and supply a familiar SQL interface to access the data. These tools can take large volumes of data and facilitate ad-hoc analytically queries.

Developers now face multiple challenges when selecting a NoSQL datastore to rely on, as there has been a proliferation of NoSQL technologies. Any particular technology has an API locking-in an application and hence moving from one technology to another requires an expensive porting effort. Any porting effort requires the user to learn the installation process, the procedure to start a distributed deployment, the API to interface with the datastore, how to add and remove nodes, maintenance procedures, backup methods, and how to correctly bring down or restart the system. This can be a costly engineering endeavor. Furthermore, because these technologies are in their infancy compared to SQL offerings, there are new releases of the software often, and this requires porting from one version to another to get bug fixes, new features, and performance improvements.

Businesses also face challenges related to the multitude of choices for NoSQL technologies. In particular, there is no easy way for developers to compare and contrast different offerings without having to become an expert in a set of datastores; there is

no simple way to move between datastores as the creators of such datastores are not incentivized to provide migration tools; there are lacking features developers have come to expect such as transactions and an expressive query language. Applications which are written for high scale may find that there is no one best-for-all-scenarios datastore, and that different applications get better performance using a particular datastore. Moreover, portions of an application may require features such as transactions, at the expense of extreme scale.

1.1 Thesis Question

The primary research question that we explore in this dissertation can be stated as follows:

> *How can we facilitate the portability and development of data-intensive applications across cloud infrastructures and storage systems while expanding functionality for analytics and migration via novel hybrid cloud techniques?*

To answer this question, we investigate novel support for the most commonly used component of these types of applications: the datastore layer. In particular, we investigate the design and implementation of a datastore-agnostic software layer for cloud platforms. The layer separates and buffers applications from the implementation of the underlying datastore. There are currently over 150 NoSQL datastores today [72] that

can be plugged in our datastore layer, which currently include Cassandra, Hypertable, HBase, Voldemort, Redis, MongoDB, and others.

Each datastore implementation requires a one time port to a unifying datastore API, installation procedure, and deployment process. Our system then automatically does installation, configuration, and starting of processes on a distributed system. Applications which run using our unifying datastore API are then able to change the underlying technology underneath without modification. We outline this process in Chapter 4.

We then export this level of indirection that this layer imparts in three unique ways to enable a wider range of datastore functionality that is lacking for most NoSQL options today:

- **Expanded NoSQL Features** Implement a common support infrastructure to provide application critical features including a limited form of ACID transactions and secondary indexing, outline and detailed in Chapter 4 and Chapter 5, respectively.

- **Analytics** Implement a hybrid cloud offline analysis system where we asynchronously replicate data from an online transactional system cloud deployment. The offline system provides expanded functionality which is restricted by the cloud platform. The data replication also lends itself for disaster recovery, capable of swap-over upon system failures (cf. Chapter 6).

- **Live Application Migration** Implement the ability to move applications and their data between cloud deployments where the underlying software stack is updated or modified with minimal to no downtime (cf. Chapter 8).

To investigate the design and implementation of these contributions, we have developed the first open source cloud platform. This platform-as-a-service technology is called AppScale (cf. Chapter 3). AppScale is API compatible with GAE, and thus any application written for GAE can also be run on AppScale without modification. AppScale is infrastructure agnostic, i.e., it executes over virtualized systems as well as public and private cloud infrastructures. This facilitates application portability across cloud and non-cloud clusters.

In summary, with this dissertation, we investigate new approaches to supporting data-intensive cloud applications that simplify and facilitate portable use of emerging storage and analytics tools (NoSQL datastores and MapReduce technologies) and that employ multiple clouds in concert (hybrid cloud computing) to do so. In particular, we contribute the following with this dissertation:

- An open source PaaS for the research community to use and extend
- A datastore abstraction layer for plug and play interchangeability of datastores
- A datastore agnostic transaction support for NoSQL stores
- A datastore agnostic secondary indexing support for NoSQL stores

- Live migration of cloud applications
- A hybrid cloud system for data analytics and disaster recovery

The result we believe is a system with which the research community can investigate new approaches to cloud computing as well as new techniques and technologies in the areas of distributed storage, application development, service management, and a hybrid cloud use, among others.

1.2 Dissertation Organization

This dissertation is organized as follows. I first provide a background on related cloud technologies in Chapter 2, which discusses the three layers of cloud computing– SaaS, PaaS, and IaaS, and cloud technologies relevant to this thesis. Chapter 3 gives an overview of AppScale, the first open source PaaS. In Chapter 4 I show the ability to compare and contrast different datastores while also adding datastore-agnostic ACID transactions support. Chapter 5 gives another extension to the datastore-agnostic layer, providing secondary indexing support for expanded querying capabilities. I connect AppScale and GAE to form a hybrid cloud in Chapter 6 to provide offline analytics and show general methods using spot instances to reduce the cost of data analytics platform in Chapter 7. Chapter 8 shows the design and implementation of live migration support

within AppScale. I conclude in Chapter 9 with related work, impact of of my thesis, and future work.

Chapter 2

Background

In this chapter, we provide background on and survey the state-of-the-art in cloud computing fabrics. Cloud computing has been standardized by the NIST to have three distinct layers in the cloud software stack: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS) [70]. Each layer has different levels of abstraction from the physical hardware from which the software runs on top of. We first consider each of the different levels shown in Figure 2.1.

- **Software-as-a-Service** Dynamic services on the internet which are accessible via a browser can be classified as a SaaS product. Through a mixture of HTML, JavaScript, CSS, and other technologies, websites are becoming more dynamic with the capabilities of native applications which were previously only available by installation on the local operating system. State has shifted from the local-host to remote storage whose physical location can be unknown but addressable

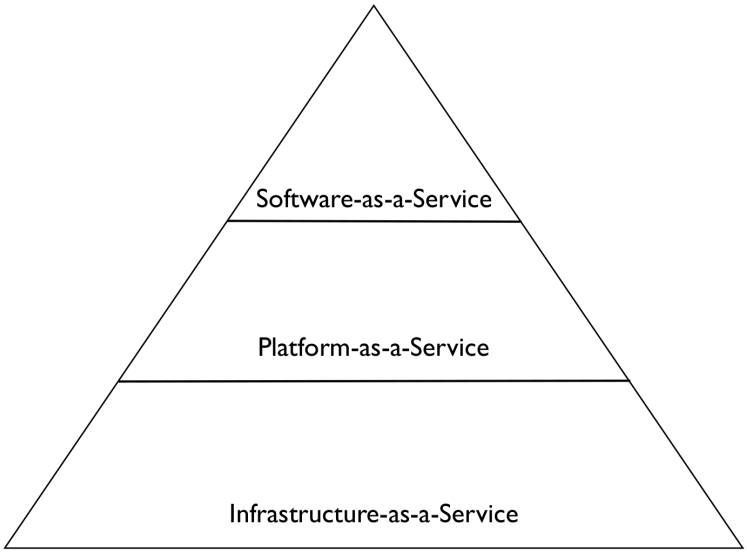


Figure 2.1: Cloud computing software layers.

through a domain name or IP. Updates to state can be administered through HTTP POST request via HTML or through JavaScript.

SaaS products have alleviated many problems software developers used to face when required to ship shrink-wrapped software. Software development iteration cycles can now be much quicker because updates can be server side rather than client side. A user can get the latest version of the service by refreshing a web page. Portability is provided by devices that run standardized browsers removing worries such as whether the machine the client is using is big-endian or little-endian or what operating system is running. Moreover, application intellectual property is protected by having core code on the servers rather than the client machine (although HTML, CSS, and JavaScript are visible to the client).

SaaS describes systems in which high-level functionality (e.g., SalesForce.com [83], which provides customer relationship management software as an on-demand service) is hosted by the cloud and exported to thin clients via the network. The main feature of SaaS systems is that the API offered to the cloud client is for a complete software service and not programming abstractions or resources. Commercial SaaS systems typically charge according to the number of users and application features. Other popular examples of SaaS include Hotmail, Facebook, and Google Apps. It is not uncommon for services to have a free tier (freemium model) and for the service to have a professional tier as well where a user can pay for additional storage or features.

- **Infrastructure-as-a-Service** Hardware virtualization has given the ability to create virtual machines (VMs) that run on physical hardware (with a guest operating system) and have them share resources between different VM instances. With hypervisors, such as Xen and KVM, VMs can be started by booting a disk image on the host operating system. These guest VMs can be allocated a maximum amount of memory, disk space, CPU cores, and network bandwidth.

  The ability to rent and use resources in an on-demand nature was driven by the outgrowth of virtualization in datacenters. Virtual machines can now be multiplexed on physical machines with resource isolation to allow for multitenancy.

Excess capacity can be sold to third parties who no longer have to to pay for upfront costs of hardware acquisition. The provider is able to achieve higher utilization of their resources by selling off spare capacity, recouping system administrator costs as well as power costs.

Customers of these services benefit from the elasticity of being able to grow and shrink based on demand, where they can acquire a fully functional machine in minutes or less. In the past, a machine would have required the installation of an operating system which can take up to 30 or more minutes, granted a machine was available on hand. Furthermore, VMs can be customized and have their image saved to be repeatedly launched, saving the time required to build the entire runtime software stack.

Customers of these resources are able to outsource their IT infrastructure and administration costs and no longer have to worry about provisioning hardware resources. Because resources are now rented on a fixed time granularity (usually on a per-hour-basis), resource consumption can be elastic, growing and shrinking entirely on a need basis. Moreover, the self-serve nature of IaaS allows for less overhead, as developers can request resources without involvement of system administrators.

Providers of public IaaS have become very specialized at IT infrastructure that is fault tolerant, highly available, and fast to provision. Yet, there is also a need to bring these benefits to large datacenters which want the flexibility and elasticity of the public cloud behind their firewall. We refer to this as a private cloud. There are many software systems which enable IaaS for private clouds including Eucalyptus, OpenStack, and CloudStack. These software systems are agnostic to the virtualization layer, being able to provision and manage virtual machines on demand regardless of whether the underlying technology is Xen, KVM, or some proprietary software such as VMWare.

- **Platform-as-a-Service** Higher up the stack from IaaS there is the PaaS layer which abstracts away the infrastructure and OS details. The platform allows developers to focus strictly on application development rather than VM or physical machine management. Concerns about CPUs, memory, and disk are under automatic management and provisioning. The PaaS provider may provide a few options into how things are managed and scheduled (i.e., trading latency for lower cost), but this is generally for more advance users of the system. The cloud provider will provide the full run-time stack and may also restrict how resources can be accessed via well defined APIs. However, some vendors will allow lower level access via SSH to the virtual machine itself for advance users to optimize, debug, and tinker with the run-time.

Some popular public PaaS offerings include Microsoft's Azure, Google App Engine, Heroku, Salesforce, and EngineYard. Any developer can sign up and use these public resources and start within the free tier. When their service requires more resources, the customer can provide a credit card to scale beyond the free quota. Different providers will charge for different metrics, but general ones include data storage, bandwidth, and front end server hours.

Much like IaaS, there is a desire to get the same capabilities of PaaS behind the firewall. Centralization and automation of application hosting can provide cost savings in infrastructure by consolidating resources, rather than forcing a programmer or IT staff to provision isolated resources for each web service or product that needs to be developed. Developers no longer have to block on resource allocation, as it becomes self service, allowing for quicker time to development.

2.0.1 History

Remote access to compute and storage resources has been around since the advent of the internet, yet the emergence of Web 2.0 and the commodization of compute and storage resources has proliferated the amount of web services available to internet users. Additionally, service-oriented architectures (SOA), where services are loosely-coupled and interoperable, has allowed for legacy systems to interface with web services, and for new services to leverage existing ones.

One of the first movers in this industry of IaaS was Amazon's Elastic Compute Cloud (EC2) in August of 2006. Users could use a set of command line tools or a web interface to start up virtual machines in a self service fashion and be charged on a per hour basis. Other charges include the amount of bandwidth used and the storage of each customized VM image.

Rackspace Cloud was another early service provider for a public IaaS with its initial release in March of 2006. Since then many other companies have entered the market to provide public IaaS cloud including Microsoft, Linode, Google, HP, IBM, and Cisco.

Heroku is a PaaS which first supported the Ruby-on-Rails (RoR) framework, and has since added support for other languages including Java and Node.js. EngineYard, another public PaaS supports RoR and PHP, while Amazon's Elastic Beanstalk provides Java, .NET, and PHP. Microsoft released Azure, a .NET cloud framework supporting Visual Basic and C#. Google's initial cloud service was Google App Engine (GAE) supporting the Python programming language, and then adding Java and Go in subsequent releases.

The IaaS and PaaS technologies mentioned are all public services, meaning any user can sign up and use the service in a self service model. Yet, these same technologies with their ability to scale, be fault tolerant, and self-service model, are very appealing to owners of privately operated datacenters and owners of compute clusters.

Private IaaS software packages include Eucalyptus, which emulates the EC2 API, the current market share leader in IaaS. Eucalyptus allows for any applications which were written for EC2, to also run in local clusters without the need to port. This is also appealing in that it eliminates vendor lock-in. CloudStack also emulates the EC2 API, while OpenStack emulates the Rackspace Cloud API. Private cloud technologies for PaaS give the same benefits of the public option yet behind an organization's firewall. CloudFoundry and OpenShift are two open source PaaS solutions currently in the market.

2.0.2 Application Building Blocks

API support for PaaS solutions provide a wide range of capabilities. Some of these include:

- Datastore
- Memcache
- Background Tasks
- Data Processing
- Monitoring
- User Management

- Blobstore
- Authentication

These APIs give access to scalable services which are charged on a usage basis, and allow developers to quickly prototype and build new services without having to reinvent commonly required services. Many PaaS offerings build upon well known frameworks such as RoR, Django, and Spring to attract existing developers familiar with such technologies.

Of these APIs, the most commonly used is the datastore which allows applications to persist their data. The most commonly used interface is SQL-based technologies for datasets that can fit within a single node. For applications that require high throughput and very large datasets (too large to fit on one machine), applications can use NoSQL technologies that are designed for larger scale at the cost of feature sets.

The technologies which first brought NoSQL storage to the forefront was Google's BigTable [16] in 2006 and Amazon's Dynamo [31] in 2007. BigTable provides storage and access structured data in a sorted multi-dimensional map, while Dynamo provides key/value access with high availability and eventual consistency. Both are designed for very large datasets (terabytes to petabytes). These technologies inspired open source implementations such as HBase [46], Hypertable [50], and Accumulo [1], which are

all BigTable clones, while Cassandra [14] is a BigTable and Dynamo hybrid using the column-oriented data model and peer-to-peer architecture.

We target the PaaS cloud layer and the management of data for applications that run on top of a PaaS for this thesis work. Much of the inspiration of this work comes from the design of the Google App Engine cloud platform, from which we have emulated the API in an open source platform called AppScale. AppScale is a private cloud option we have built and maintained at UC Santa Barbara. We detail both AppScale and Google App Engine in Chapter 3.

Chapter 3

AppScale

In this chapter, we detail the AppScale cloud platform which we developed to enable our disseration research as well as the research of others into the next generation of cloud systems and applications. AppScale is a software infrastructure that simplifies the deployment of network-accessible programs over distributed cluster resources by exporting the runtime platform "as a service". AppScale is open source, fully distributed, scalable, fault resilient and executes over virtualized cluster resource including on-premise (private virtualized clusters or on-premise IaaS systems) and public cloud infrastructures (public IaaS systems). AppScale is unique from other cloud offerings in that the APIs it exports and implements are the same as those for Google App Engine. That is, AppScale is API-compatible with Google App Engine.

Google App Engine is a public cloud PaaS that exports scalable and elastic web service technologies via well-defined APIs. These APIs implement messaging, key-value data storage, map-reduce, mail, and user authentication, among other services.

The platform facilitates easy asynchronous multi-tasking, web server support, elasticity, and resource management. The most important of the APIs is the datastore which persists data. GAE builds this API on top of two technologies: BigTable [16] and Megastore [6]. BigTable provides scalable storage with the capability to do range queries based on keys. Megastore builds on top of BigTable, adding transaction support and high availability.

App Engine applications developers debug and test their programs using an open-source software development kit (SDK) provided by Google that implements non-scalable versions of the APIs. Developers then upload their code and data to Google clusters and use Google cluster resources and services on a free (up to some fixed set of per-resource quotas) and pay-per-use (resource rental) basis.

AppScale implements the APIs of App Engine by replacing the SDK implementations with distributed, scalable, and fault tolerant versions. Like App Engine, AppScale implements multiple language runtimes – Java, Python, and Go – (via elastic application servers). We employ a wide range of open source technologies for their implementations. We overview the APIs and their implementation techologies for both App Engine and

By providing API-compatibility, any application that executes over Google App Engine also executes over AppScale without modification. Since AppScale executes over any virtualized cluster resources, users can deploy an AppScale cloud (a platform

that emulates Google App Engine) on-premise using cluster or IaaS resources or over a public IaaS system. Currently, AppScale provides support for automatic deployment over Eucalyptus and Amazon EC2. AppScale is not a replacement for Google App Engine or any other public cloud technology however. In particular, AppScale is only as scalable as its underlying physical resource pool. Instead, AppScale is a robust and extensible research infrastructure and private cloud platform that provisions the available resources across multiple applications.

This chapter details the design and implementation of AppScale from a high level and defines key terms which are used throughout this dissertation. Futuremore, it showcases the APIs supported by AppScale and their implementations. Lastly, it gives initial experiments given real applications along with an evaluation analyzing the results.

3.1 Background

The open-source offering most similar to AppScale is Typhoonae [86] which came out six months after our initial release. Typhoonae runs GAE python applications and does so with more scalable components than the SDK provides.

There are multiple differences between AppScale and Typhoonae. First, Typhoonae (and any GAE applications that execute using it) is hosted entirely using a single guestVM image, which places significant limitations on IaaS usage/accounting, performance, scalability, and fault tolerance. AppScale is able to be deployed on many

machines and has been run up to as many as 96 nodes. Our API implementations are meant for a distributed system whereas Typhoonae has a single point of failure for all APIs. Moreover, AppScale has cloud support for IaaS software such as EC2, Eucalyptus, and OpenStack or the vast selection of datastores.

3.2 AppScale

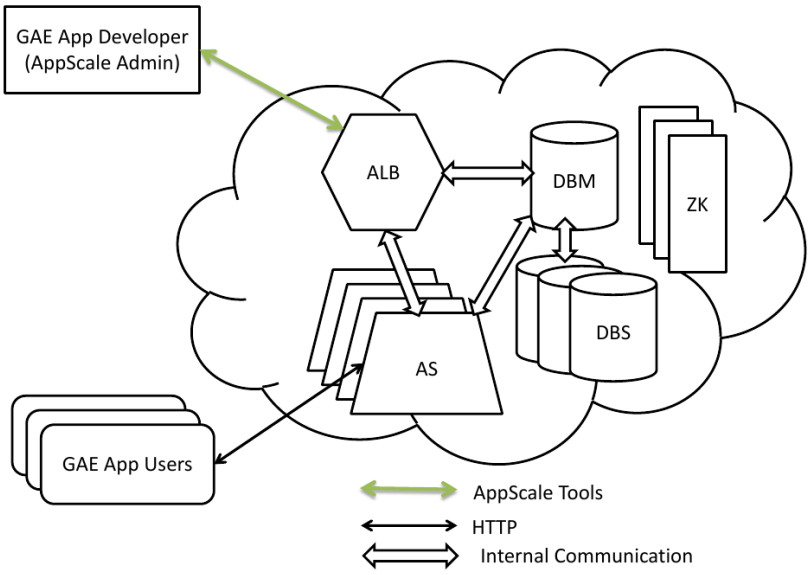


Figure 3.1: Overview of the AppScale design. The AppScale cloud consists of an AppLoadBalancer (ALB), a Database Master (DBM), one or more Database Slaves (DBS), one or more ZooKeepers (ZK), and one or more AppServers (ASs). Users of GAE applications interact with ASs or indirectly through the load balancer; the developer deploys AppScale and her GAE applications through the head node (i.e. the node on which the ALB is located) using the AppScale Tools. AppControllers (ACs) on each node interact with the other nodes in the system; ASs interact with the DBM via HTTP or HTTPS.

To provide a platform for GAE application execution using local and private cluster resources, to investigate novel cloud services, and to facilitate research for the next-generation of cloud software and applications, we have implemented AppScale. AppScale is a multi-language, multi-component framework for executing GAE applications. Figure 3.1 overviews the AppScale design and its high level roles and components.

AppScale consists of a toolset (the AppScale Tools), three primary components, the AppServer (ASs), the database management system, and the AppLoadBalancer (ALB), and an AppController (AC) for inter-component communication. AppServers are the execution engines for GAE applications which interact with a Database Master (DBM) via HTTP for data storage and access. Database Slaves (DBSs) facilitate distributed, scalable, and fault tolerant data management. The AppController is responsible for setup, initialization, and tear down of AppScale instances, as well as cross component interaction. In addition, the AppController facilitates deployment of and authentication for GAE applications. The ALB serves as the head node of an AppScale deployment and initiates connections to GAE applications running in ASs. The AC of the head node also monitors and manages the resource use and availability of the deployment. All communications across the system are encrypted via the secure socket layer (SSL).

A GAE application developer interacts with an AppScale instance (cloud) remotely using the AppScale Tools. Developers use these tools to deploy AppScale, to submit

GAE applications to deployed AppScale instances, and to interact with and administer AppScale instances and deployed GAE applications. We distinguish developers from *users*; users are the clients/users of individual GAE applications.

An AppScale deployment consists of one or more virtualized operating system instances (guestVMs). GuestVMs are Linux systems (*nodes*) that execute over the Xen virtual machine monitor, the Kernel Virtual Machine (KVM) [57] or IaaS systems such as Amazon's EC2 and Eucalyptus For each AppScale deployment, there is a single AppLoadBalancer (ALB) which we consider the head node, one or more AppServers (AS), one Database Master (DBM) and one or more Database Slaves (DBSs). A node can implement any individual component as well as any combination of these components; the AppScale configuration can be specified by the developer via command line options of an AppScale tool.

We next detail the implementation of each of these components. To facilitate this implementation we employ and extend a number of existing, successful, web service technologies and language frameworks.

3.2.1 ZooKeeper (ZK)

Roles of components of nodes is stored in ZooKeeper (ZK), which is fault tolerant and fast locking system to handle distributed coordination. ZK nodes use the Paxos algorithm to keep data synchronized between nodes. All roles must register with ZK

before proceeding with normal operation, and they must periodically check ZK to make sure they correctly registered. ZK is also used by the transaction system explained in the following chapter.

3.2.2 AppController (AC)

The AppController (AC) is a SOAP client/server daemon written in Ruby. The AC executes on every node and starts automatically when the guestVM boots. The AC on the head node starts the ALB first and initiates deployment and boot of any other guestVM. This AC then contacts the ACs on the other guestVMs and spawns the components on each node. The head node AC first spawns the DBM (which then starts the DBSs) and then spawns the AppServers, configuring each with the IP of the DBM (to enable access to the database via HTTP or HTTPS).

The AC on the head node also monitors the AppScale deployment for failed nodes and for opportunities to grow and shrink the AppScale deployment according to system demand and developer preferences. The AC periodically polls (currently every 10 seconds) the AC of every other node for a “heartbeat” and to collect per-application behavior and resource use (e.g. CPU and memory load). When a component fails, the AC restarts the component with the use of the Ruby process ’god’.

Although in this chapter we evaluate the static default deployment of AppScale, we can also use this feedback mechanism to spawn and kill individual nodes of a de-

ployment to respond to system load and performance. Killing nodes reduces resource consumption (and cost of resources are being paid for) and consists of stopping the components within a node and destroying the guestVM. We spawn nodes to add more AppServers, LoadBalancers, or Database Slaves to the system.

3.2.3 AppLoadBalancer (ALB)

The AppLoadBalancer is a Ruby on Rails [82] application that employs a simple HTTP server (nginx [71]) to select between three replicated Mongrel application servers [66] (for head-node load balancing). The ALB distributes initial requests from users to the AppServers (ASs) of GAE applications. Users initially contact the ALB to request a login to a GAE application. The ALB provides and/or authenticates this login and then selects an AS randomly. It then redirects the user request to the selected AS. The user, once redirected, continues to use the AppServer to which she was routed and does not interact further with the ALB unless she logs out or the AppServer she is using becomes unreachable. We also have support for the ALB to act as a full proxy rather than a reverse proxy.

3.2.4 AppServer (AS)

An AppServer is an extension to the development server distributed freely as part of the Google AppEngine SDK for GAE application execution for the Python, Java,

and Go languages. Our extensions to the development server enable fully automated execution of GAE applications on any virtualized cluster to which the developer has access, including EC2, Eucalyptus, and OpenStack. Our extensions provide a generic datastore interface through which any database technology can be used. We have implemented this interface to HBase, and Hypertable, open-source implementations of Google's BigTable that execute over the distributed Hadoop File System (HDFS) [41]. We also have plugins for MySQL Cluster [68], Cassandra [14], Voldemort [88], and more.

We intercept the protocol buffer requests from the application and route them over HTTP to/from the DBM front-end called the *PBServer*. The PBServer implements the interface to every datastore available and routes the requests to the appropriate datastore. The interaction is simple but fully supported by a number of different error conditions, and includes:

- Put: add a new item into the table (create table if non-existent)
- Get: retrieve an item by ID number or unique name
- Query: limited SQL query semantics
- Delete: delete an item by ID number or unique name

Chapter 4 does into further details of this datastore interface.

Our other extensions facilitate automatic invocation of ASs and authentication of GAE users. The AC of the node sets the location of the datastore (passed in from a request from the head node AC), upon AS start. The AS also stores and verifies the cookie secret that we use to authenticate users and direct the component to authenticate using the local AppController (AC).

An AS executes a single GAE application at time. To host multiple GAE applications, AppScale uses additional ASs (one or more per GAE application) that it isolates within their own AppScale nodes or that it co-locates within other nodes containing other AppScale components.

3.2.5 Data Management

In front of the Database Master (DBM) sits the The PBServer is the front-end of the DBM. This Python program processes protocol buffers from a GAE application and makes requests on its behalf to read and write data to the datastore. As mentioned previously, AppScale currently supports HBase and Hypertable datastores. Both execute over HDFS within AppScale which performs replication, fault tolerance, and provides reliable service using distributed Database Slaves. The PBServer interfaces with HBase, Hypertable, Cassandra, and Voldemort using Thrift for cross-language interoperation.

The AC on the DBM node provides access to the datastore via these interfaces to the other ACs and the ALB of an AppScale system. The ALB stores uploaded GAE applications as well as user credentials in the database to authenticate the developer and users of GAE applications.

3.2.6 AppScale Tools

The developer employs the AppScale tools to setup an AppScale instance and to deploy GAE applications over AppScale. The toolset consists of a small number of Ruby scripts that we named in the spirit of Amazon's EC2 tools for AWS. The tools facilitate AppScale deployment on Xen-based clusters as well as IaaS infrastructures. The latter two systems require credentials and service-level agreements (SLAs) for the use, allocation (killing and spawning of instances) of resources on behalf of a developer; the EC2 tools (for either IaaS system) generate, manage, distribute (to deployed instances), and authenticate the credentials throughout the cluster. The AppScale tools sit above these commands and make use of them for credential management in IaaS settings. In a Xen-only setting, no credential management is necessary; the tools employ ssh keys for cluster management. The tools enable developers to start an AppScale system, to deploy and tear down GAE applications, to query the state and performance of an AppScale deployment or application, and to manipulate the AppScale configura-

tion and state. There is currently a limit of 10 on the number of applications that can be uploaded, yet this number can be arbitrarily changed.

3.2.7 Tolerating Failures

There are multiple ways in which AppScale is fault tolerant. One component which is fault tolerant is the AppController which executes on all nodes. If the AC fails on a node with an AS, that AS can no longer authenticate users for a particular GAE application but authenticated users proceed unimpeded. Users that contact an ALB to re-authenticate (acquire a cookie) are redirected to a node with a functioning AS/AC to continue accessing the application. If the AC fails on the node with the ALB, no new users can reach any GAE applications deployed in the AppScale instance and the developer is not able to upload additional GAE applications; extant users however, are unaffected. This scenario (AC on the ALB node failure) is similar to AC failure on the DBM node. In this scenario (AC on the DBM node failure), ASs and users are unaffected.

The database system continues to function as long as at least one DBS is available with a replica. Similarly, the system is tolerant to failure of the PBServer (DBM front-end). If the PBServer fails on the DBM, the ASs will temporarily be unable to reach the database until the AC on the node restarts the PBServer. The ASs are not able to continue to execute (GAE applications will fail) if the DBM goes down or becomes

unreachable. In this scenario, the ALB will restart the DBM component but unless the data from the original DBM is available to restore, the restart is similar to restarting AppScale.

Although, coupling multiple components per node reduces the number of nodes (resource requirements) and potentially better utilizes underlying resources, it also increases the likelihood of failure. For example, if all components are located in a single node, node failure equals system failure–a single point of failure. If the node containing the ALB and DBM fails, the system fails. In these scenarios, component failure does not equal node failure however; the AC in the head node will attempt to restart components with god. The DBM issues 3 replicas by default of tables for DBSs to store, thus user data is available on failure of any individual DBS component. The Ruby god process will automatically start up a process if it crashes for whatever reason, and will also monitor a process and kill and restart it if it takes more than a threshold of memory.

We distribute AppScale as a single Linux image and the AppScale Toolset. The image contains the code for the implementation of all of the components and a 64-bit Linux kernel and Ubuntu distribution. The system is available from `http://appscale.cs.ucsb.edu/`; all new programs that we have contributed carry the Berkeley Software Distribution (BSD) License.

3.3 API Support

This section enumerates the many GAE APIs available for developers to use easing their development process. Each API is scalable and some are built upon others. All APIs must have identical interfaces and side effects so as the applications which run on top of them do not know whether they are running on Google's infrastructure or on AppScale.

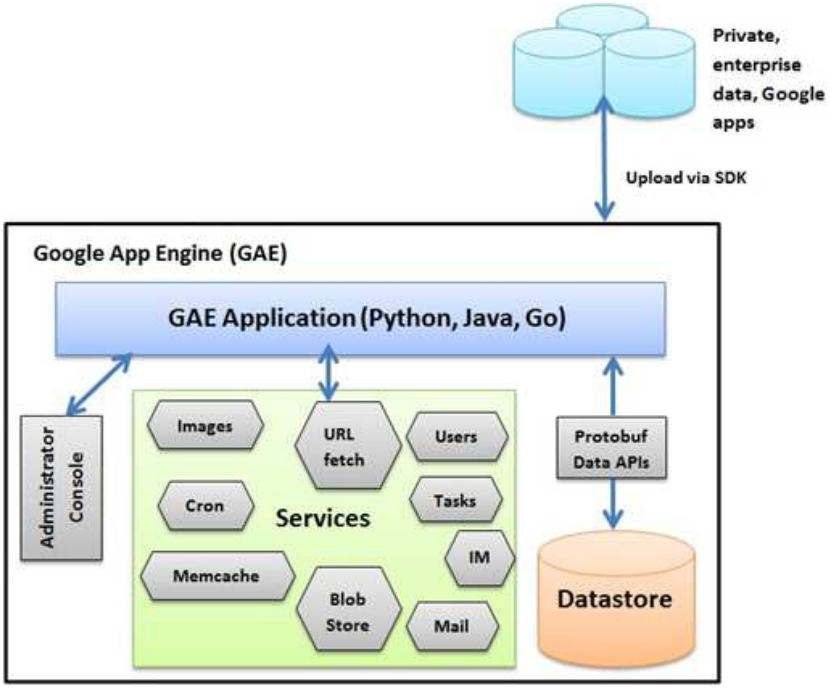


Figure 3.2: APIs in AppScale.

Datastore The Datastore API allows for the persistence of application data. The API provides both a key/value interface along with query support. The Google Query Language (GQL) is similar to and is a subset of SQL; fundamentally, it lacks relational

operations such as JOIN and MERGE to enable scale and elasticity. An example of such a query for an application where users sign a guestbook is as such:

```
SELECT * FROM Greetings ORDER BY date DESC LIMIT 10")
```

which states to get ten greetings sorted by descending date.

AppScale initially employed in-memory filters for GQL statements while later supporting property indexing for scalable queries (detailed in Chapter 5), emulating Google App Engine's use of BigTable and Megastore for data persistence.

AppScale implements transactional semantics (multi-row/key atomic updates) using the same semantics as Google App Engine. Transactions can only be performed within an entity group. Entity groups are programmatic structures that describe relationships between datastore elements. Similarly, transactions are expressed within a program via application functions passed to the Google App Engine run as transaction function. All AppScale datastore operations within a transaction are ACID-compliant with READ-COMMITTED isolation. Transaction support is expanded on in Chapter 4 along with the capability of allowing for the pluggability of different NoSQL datastore options.

This chapter uses the Cassandra plug in for its evaluation, while the subsequent chapter evaluate a wider range of datastore technologies for the given API support with and without transactions.

Namespace The Namespace API implements the ability to segregate data into different namespaces. For example, developers can test their application in production without tampering with live production data. The Namespace API can also be used with the Memcache and Task Queue APIs.

It is implemented by appending the given namespace to the entity kind for table naming, separated by a delimiter. This provides isolation between namespaces.

Memcache The Memcache API permits applications to store their frequently used data in a distributed memory grid via a key/value API. This can serve as a cache to prevent relatively expensive re-computations or database accesses. Developers must be aware that it is possible that entries may be evacuated to create space for new updates.

The Memcache API is implemented in AppScale using memcached, an open source, distributed memory object caching system. AppScale places a memcache role on each machine which is running the same node as an AS. The keys have the application name preprended to provide isolation between applications.

Blobstore Google App Engine's Blobstore API is the primary method of storing large objects. There are two methods of getting blobs uploaded, one is the Files API, in which you directly supply a large binary object programmatically, and the other is via an HTML form.

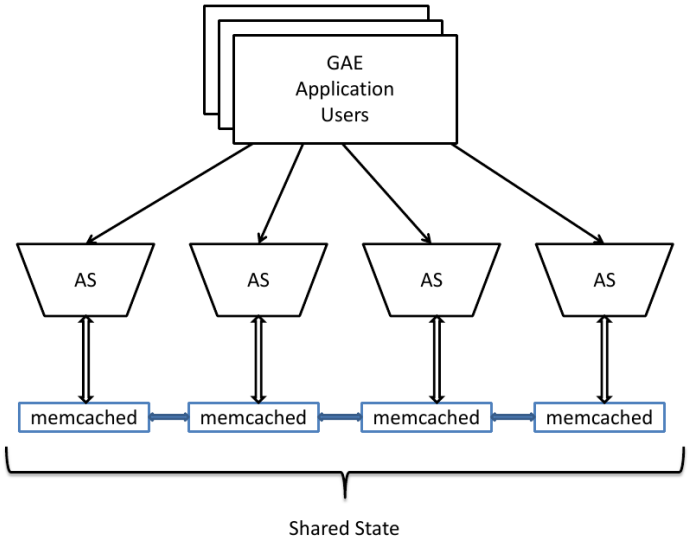


Figure 3.3: Memcache is distributed on multiple nodes and is shared state between different ASes. ASes by themselves are stateless and can be started and stopped as needed.

The Blobstore service in AppScale uses a server solely responsible for uploading data into the datastore. This server is a python tornado server which handles request from all applications. Figure 3.4 shows the flow for uploading blobs.

1. The user requests a web page which has an upload file form

2. The application will create a blobstore session and store the session info into the datastore to prevent unauthorized uploads and then return a unique path to the blobstore server

3. The action path of the HTML form contains the path from the previous step

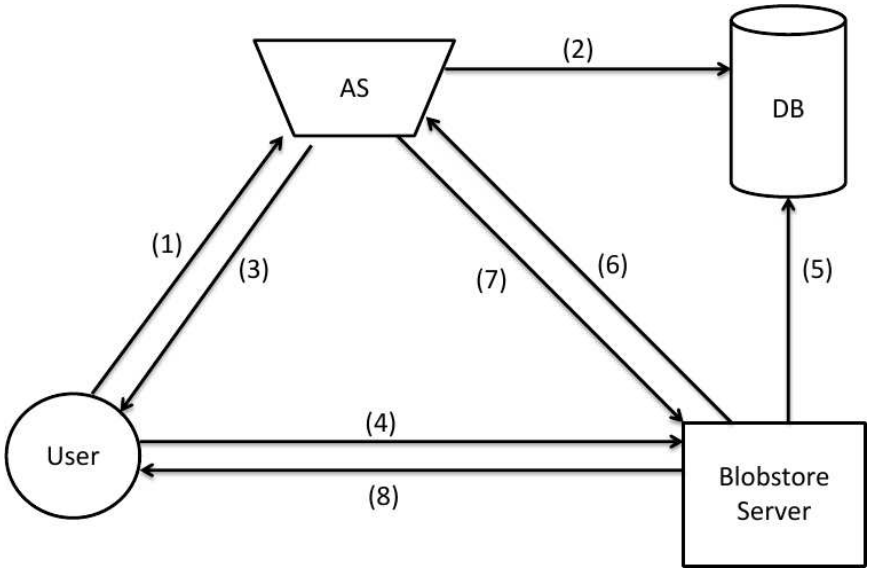


Figure 3.4: Blobstore allows for the uploading of large files which are stored in the datastore in 1MB chunks.

4. When the user submits the form it goes to the blobstore server

5. The blobstore server interacts with the datastore by verifying the session, storing blob information, storing the file in 1MB chunks, and the removing the session

6. A POST is done to the successful path (stored as meta data) and all form elements are forwarded

7. The successful path handler does a redirect

8. The redirect is forwarded to the user client

XMPP The XMPP API gives user the ability to receive and send messages using a valid XMPP account. Google App Engine leverages the Google Talk infrastructure

while AppScale employs Ejabberd, an instant messaging software written in Erlang and highly scalable.

Channel The Channel API allows for the pushing of messages to a client's JavaScript code. AppScale's scalable implementation is built using Ejabberd and StropheJS, two open source projects.

There are two sets of APIs for the developer. First is the API which consists of *create_channel(app_client_id)* and *send_message(app_client_id, message)*. The create channel API uses the XMPP service implementation of AppScale. We are able to leverage Ejabberd to take care of the distribution and sending of messages. We must create temporary accounts with each new channel created. This requires garbage collection of channels which live on longer than a prescribe period of time.

The second set of APIs is for the JavaScript client which can be included into the developer's code by adding an import statement within the header of the user's HTML code.

This API allows for the creation of connections using StropheJS. Strophejs is a robust and open source project that enabled BOSH connections to Ejabberd. The creation of a channel socket uses StropheJS's connections, as well as its message callbacks. The functions have the same name and functionality to preserve the API, but the implemen-

tation is different, where Google's implementation uses Google Talk and their internal XMPP service.

Users The Users API provides authentication for web applications through the use of HTTP cookies. Google App Engine's implementation leverages the Google Accounts infrastructure, so users with a Google Account can use it to access App Engine apps. Since AppScale does not have access to this infrastructure, it requires that users create an account through an AppScale portal URL. Alternatively, AppScale can be extended to employ other authentication services, e.g. those provided by the Eucalyptus open source cloud infrastructure or via LDAP. The AppScale implementation of this API distinguishes between regular users, application administrators, and cloud administrators (the latter categories possessing greater privileges).

Mail AppScale allows for outgoing mail using the Unix command *sendmail*, whereas Google leverages their GMail infrastructure. AppScale does not, however, currently support incoming mail.

Images The Images API facilitates programmatic manipulation of images. Popular functions in this API include the ability to generate thumbnails, perform rotations, composition, conversion between formats, and cropping images. AppScale is able to use the SDK implementation.

URL Fetch An application can perform POST and GET requests on remote resources using the URL Fetch API. In addition, the application can access REST APIs from third parties using this API. Certain outgoing ports are limited for security reasons. The SDK implementation is used, but modified to also allow for asynchronous fetches.

Task Queue The Task Queue API facilitates asynchronous computation (tasks) by applications. Such background computation is important for applications that perform operations other than those in response to a web request. In AppScale, cloud administrators can set both the (inline) computation duration limit and asynchronous task duration limit if desired. Moreover, tasks can be chained so that one task can pick up where a previous one left off. The implementation is discussed in detail in Chapter 6.

3.4 Evaluation

We next present the basic performance characteristics of AppScale default deployment of four nodes using a full proxy at the ALB. We note that this study presents a baseline from which we will work to improve the performance and scalability of the system over time and serves as a snapshot in time. Our goal with AppScale to provide a research framework for the community, thus, we and others will likely identify ways to improve its performance over time. We simply provide a framework with which to

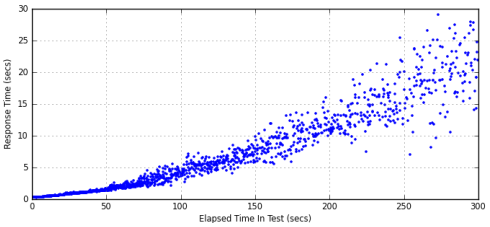
investigate existing open source GAE applications, services, and execution characteristics using local cluster resources.

3.4.1 Methodology

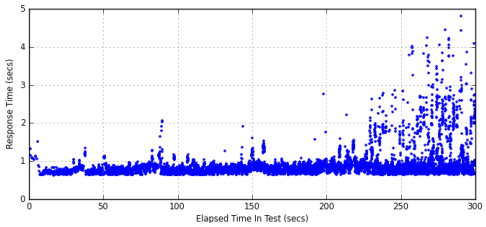
We use the Multi-Mech framework capable to simulating multiple concurrent users. We use the configuration of a max of 60 concurrent threads for a run time of 300 seconds. Threads are ramped up evenly over the course of the experiment until all 60 threads are operating, simulating organic growth. For each experiment, we investigate throughput (number of transactions per second) and latency. Numbers are calculated and put into 30 second buckets.

Our testbed uses Eucalyptus 3.0. We acquired 4 VMs of m1.large with the characteristics of 4 cores with 7500 MB of RAM and 20 GB of instance disk space.

Our benchmarks consist of real applications that run on the GAE framework. Here we look at three different applications: guestbook, shell, and sieves. The guestbook application allows for users to sign a post. Of our test users, half either request to see the signing, or sign the guestbook themselves. The shell application allows for python commands to be run on a shell-like setting. Our test users all run the command ”a = 5; b = 2; a + b;”. The sieves application prints out the first 1000 prime numbers, and, unlike the other two applications, does not access the datastore.

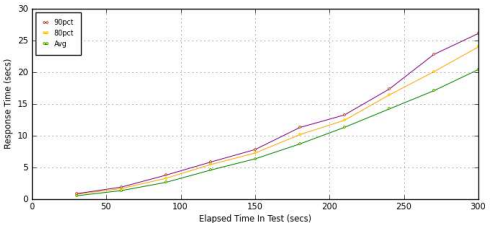


(a) AppScale

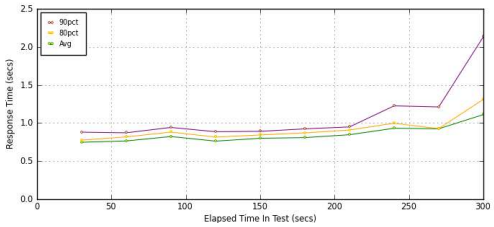


(b) Google

Figure 3.5: Points of latency of request over time as users are ramped up over time for the guestbook application.



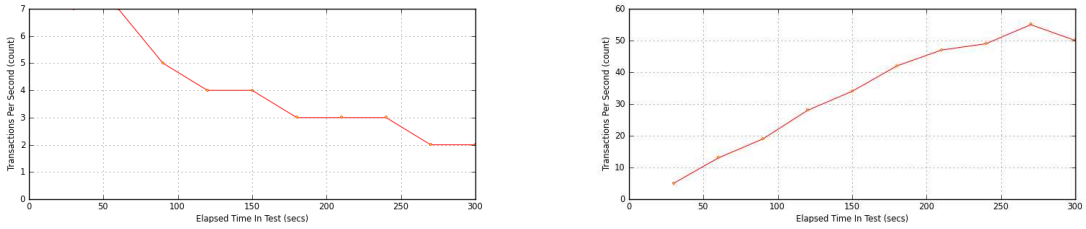
(a) AppScale



(b) Google

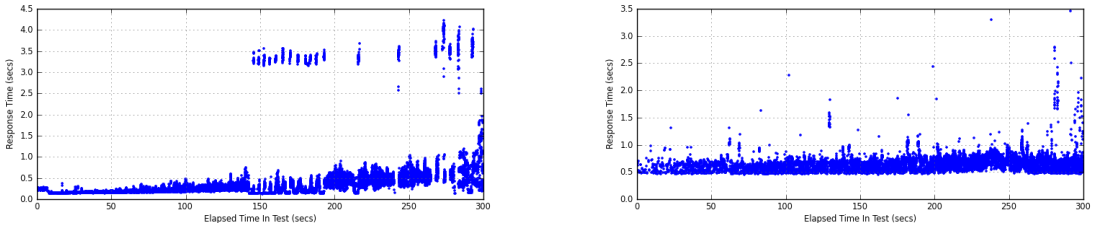
Figure 3.6: Time series of latency of request over time showing the 80th percentile, 90th percentile, and average as users are ramped up over time for the guestbook application.

The guestbook application consists of two database accesses. The first is a query for the last ten items posted on the site. The other is a posting to the site. Figure 3.5 shows the plotting of all latencies for each request. We see that in AppScale the latency gradually increases as more and more users enter the system, while Google maintains some consistency before getting more sporadic results. Figure 3.6 shows the average of the points as well as the 80th and 80th percentile. Here we see while AppScale increases in its latency, Google maintains consistency latency until the 220 second mark. Figure 3.7 has the number of transactions per second achieved by AppScale and Google, and it



(a) AppScale (b) Google

Figure 3.7: Throughput over time as the number of users are ramped up over time for the guestbook application.

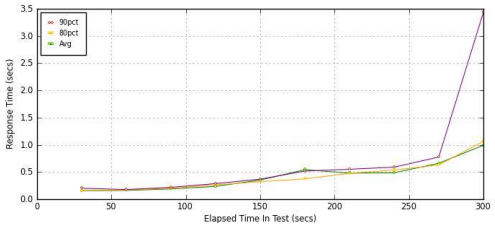


(a) AppScale (b) Google

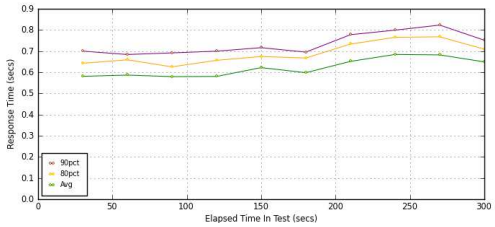
Figure 3.8: Points of latency of request over time as users are ramped up over time for the shell application.

can be seen that throughput drops over time for AppScale, while Google is able to scale well with its virtually unlimited resources.

The primary reason for the inadequate performance by AppScale is because query support brings the entire table into the memory before being filtered. Hence, there is not only additional load as users are posting but also adding to the amount of data which must be filtered in memory. This inefficiency is addressed in Chapter 5 in which range query support is used to get O(1) timing much like how Google is able to achieve with its BigTable/Megastore storage layer.

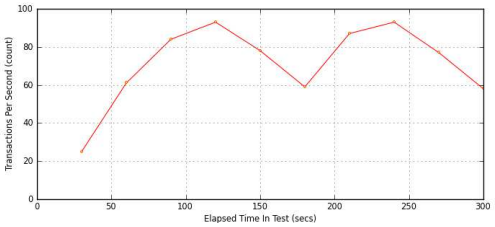


(a) AppScale

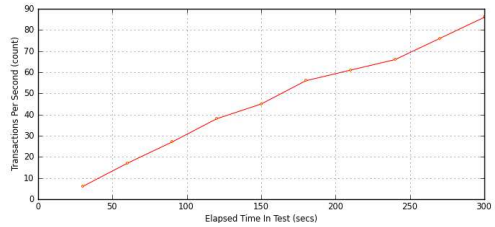


(b) Google

Figure 3.9: Time series of latency of request over time showing the 80th percentile, 90th percentile, and average as users are ramped up over time for the shell application.



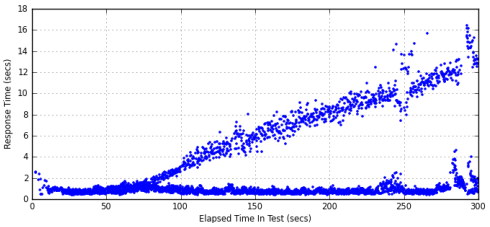
(a) AppScale



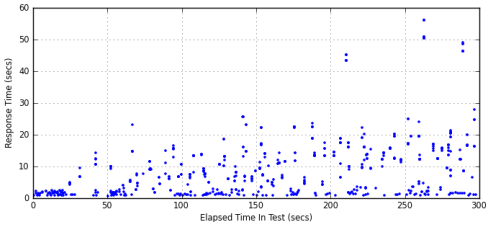
(b) Google

Figure 3.10: Throughput over time as the number of users are ramped up over time for the shell application.

The shell application sees better performance compared to guestbook. Here each operation does writes and reads to the datastore to store the variables given by the user in the command line. Figure 3.8 shows slower growth in latency but also shows a bi-modal latency where a majority of latencies are less than one second, but there are also segments of latencies after the 140 second mark of higher than 3 seconds, which explains the temporary drop in throughput shown in Figure 3.10(a). Google's responsiveness is more variable compared to AppScale's in the first 150 seconds. AppScale is able to achieve higher throughput more quickly, but kneels over after the 110 second mark. Google gets to the 80 transactions per second mark right before the end of the

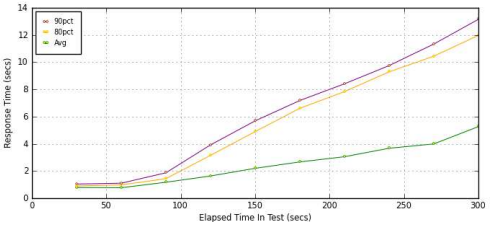


(a) AppScale

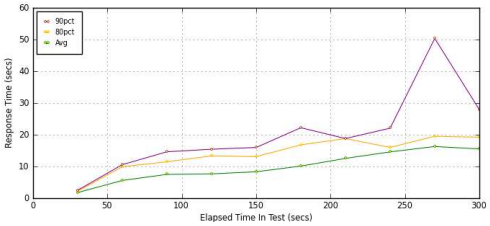


(b) Google

Figure 3.11: Points of latency of request over time as users are ramped up over time for the sieves application.



(a) AppScale

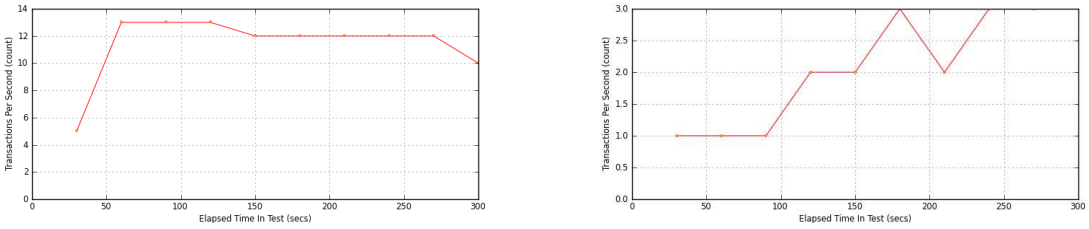


(b) Google

Figure 3.12: Time series of latency of request over time showing the 80th percentile, 90th percentile, and average as users are ramped up over time for the sieves application.

experiment, but it should be noted that our load generator machine was in the same subnet as that of AppScale, whereas Google suffered a higher level of latency due to round trip time which we measured at 51ms. We also see higher variance at the end of the experiment for Google where the throughput tapered off.

The sieves application is only computational, and does not access the datastore. Here in Figure 3.11(a) and Figure 3.12(a) we see that AppScale has two levels of responsiveness, one which is under two seconds in latency, and the other which grows as the number of users grows. By comparison, Figure 3.11(b) and Figure 3.12(b) has much higher variability in its responsiveness. AppScale maintains a much higher level of



(a) AppScale (b) Google

Figure 3.13: Throughput over time as the number of users are ramped up over time for the sieves application.

transactions per second for the sieves application at 12 per second, compared to Google which ranges between 1.0 and 3.0 per second (Figure 3.13(a) and Figure 3.13(b), respectively).

3.5 Summary

We present AppScale, an open source PaaS cloud computing research framework that emulates the Google AppEngine-based cloud offering. AppScale is easy to use and to extend and automatically deploys itself and GAE applications over Xen-based cluster resources and IaaS clouds such as Amazon EC2 and Eucalyptus. AppScale implements a number of different components that facilitate deployment of GAE applications using local (non-proprietary resources). Moreover, AppScale provides a framework with which cloud researchers and application developers can investigate new techniques (services, tools, schedulers, optimizations), and the performance and behavior of these techniques, and for real (GAE) applications.

This chapter gave an overview of the research platform that is central to the the dissertation thesis. AppScale provides the framework for which the subsequent chapters are based off of where real GAE application can run. Later chapters expand on AppScale on thesis topics related to scalable data management, including transaction support for NoSQL datastores and big data analytics. The next chapter looks to extend AppScale by supplying a pluggable interface for datastores and the means for a middleware for supplemental feature sets such as transactions.

Chapter 4

A Database-Agnostic Cloud Platform with Transaction Support

Given the availability of vast compute and storage resources available on-demand, along with virtually infinite amounts of information (financial, scientific, social) via the Internet, applications have become increasingly data-centric and our data resources and products have grown explosively in both number and size. One prominent way in which a wide range of applications access such data is via well-defined structures that facilitate data processing, manipulation, and communication. Structured data access (via database/datastore systems) is a mature technology in wide-spread use that provides programmatic and web-based access to vast amounts of data efficiently.

Public and private cloud providers increasingly employ specialized databases, called key-value stores (or datastores) [17, 28, 31, 16, 46, 14, 81, 88, 64, 50]. These systems support structured data access over warehouse-scale resource pools, by large numbers of concurrent users and applications, and with elasticity (dynamic growing and shrink-

ing of resource and table use). Examples of public cloud datastores include Google's BigTable, Amazon Web Services (AWS) SimpleDB, and Microsoft's Azure Table Storage. Examples of private or internal cloud use of datastores include Amazon's Dynamo [31], and customized versions of open source systems (e.g. HBase [46], Hypertable [50], Cassandra [14], etc.) is in use by Facebook, Baidu, SourceForge, LinkedIn, Twitter, Reddit, and others.

To enable high scalability and dynamism, key-value stores differ significantly from more traditional database technologies (e.g. relational systems) in that they are much simpler (entities are accessed via a single key) and exclude support for multi-table queries (e.g. joins, unions, differencing, merges, etc.) and other features such as multi-row (multi-key) atomic transaction support. Extant datastore offerings differ in query language, topology (master/slave vs peer-to-peer), data consistency policy, replication policy, programming interfaces, and implementations in different programming languages. Moreover, each system has a unique methodology for configuring and deploying the system in a distributed environment.

In this chapter, we address two growing challenges brought up by the thesis questions with the use of cloud-based datastore technologies. The first is the vast diversity of offerings: applications written to use one datastore must be modified and ported to use another. Moreover, it is difficult to ”test drive” public offerings extensively without paying for such use, and challenging to configure and deploy distributed open source

technologies in a private setting. The second challenge is the lack of support for atomic transactions across multiple keys in a table. Most datastores offer atomic updates at the row (key) level only which are important for applications where eventually consistent data can be hard to reason about in application logic. The lack of all-or-nothing updates to multiple data entities concurrently precludes many business, financial, and data-analytic applications and significantly limits datastore utility for all but very simple applications.

To address these issues, we present the design and implementation of a database-agnostic, portability layer for cloud platforms. This layer consists of a well-defined API for key-value-based structured storage, a plug-in model for integrating different database/datastore technologies into the platform, and a set of components that automatically configures and deploys any datastore that is plugged into the layer. This layer decouples the API that applications use to access a datastore from its implementation (to enable program portability across datastore systems) and automates distributed deployment of these systems (to make it easy to configure and deploy the systems). Developers write their application to use our datastore API and their applications execute using any datastore that plugs into the platform, without modification. This support enables us to compare and contrast the different systems for different applications and usage models and enables users to select across different datastore technologies with less effort and learning curve.

To address the second challenge, we extend this layer to provide distributed transactional semantics for the datastore plug-ins. Such semantics increase the range of applications that can make use of cloud systems. Our approach emulates and extends the limited transaction semantics of the Google App Engine cloud platform to provide atomic, consistent, isolated, and durable (ACID) updates to multiple rows at a time for any datastore that provides row-level atomicity. To enable this, we rely on ZooKeeper [94], an open-source distributed directory service that maintains consistent and highly available access to metadata using a variant of the Paxos algorithm [59, 15].

We implement this database-agnostic software layer within the open source AppScale cloud platform and integrate a number of different popular open source and proprietary database and datastore systems. These plug-ins include Cassandra, HBase, Hypertable, and MySQL cluster [69] (which we employ as a key-value store), among others. Moreover, since AppScale executes over different infrastructure-as-a-service (IaaS) cloud systems (Amazon EC2 [2] and Eucalyptus [73, 34]) and emulates Google App Engine functionality, developers are given the freedom to choose the infrastructure on which their application runs on, providing far reaching application portability.

In the sections that follow, we present related work and then describe the design and implementation of the abstract database layer that decouples the AppScale datastore API from the plug-ins (implementations of the API). We describe how we extend this layer with ACID transaction semantics in a database-agnostic fashion in Section 4.3.

We then present an evaluation of the system using different datastores in Section 4.6, and conclude in Section 4.7.

4.1 Background

Distributed transactions, two-phase locking, and multi-version concurrency control (MVCC) have been employed in a multitude of distributed systems since the distributed transaction process was defined in [8]. Our design is based on MVCC and uses versioning of data entities. Google App Engine's implementation of transactions uses optimistic currency control [90], which was first presented by Kung et al. in 1981 [56].

There are two systems closely related to our work that provide a software layer implementing transactional semantics over top of distributed datastore systems. They are Google's Percolator [76] and Megastore [7]. Percolator is a system, proprietary to Google, that provides distributed transaction support for the BigTable datastore. The system is used by Google to enable incremental processing of web indexes. Megastore is the most similar to our system as it is used directly by Google App Engine for transactions and for secondary indexing. Our approach is database agnostic and not tied to any particular datastore. Prior approaches tightly couple transaction support to the database. DAT can be used for any key/value store and, with AppScale, provide scale, fault tolerance, and reliability with an open source solution. Moreover, our system is platform agnostic as well (running in/on Eucalyptus, OpenStack [75], EC2, VMWare,

Xen [92], and KVM [57]) while automatically installing and configuring a datastore and the DAT layer for any given number of nodes.

Cloud TPS [89] provides transactional semantics over key spaces in datastores such as HBase. Cloud TPS achieves high throughput because its design is based heavily in in-memory storage. Replication is done across nodes in memory, and the system will periodically flush the data to a persistence layer such as S3 or another cloud storage. DAT differs from Cloud TPS providing higher durability because DAT requires each write to be written to disk. In the case of system wide outages, it is possible to lose all transactions which have not been persisted with Cloud TPS, while in DAT all writes are written to a journal which is replicated on disk at multiple nodes.

In [53] Kossman et al. compared different clouds and datastores, one of which is GAE. GAE has improved over time so the results, while valid at that point in time, are no longer valid. The same can be said for the other clouds which were benchmarked, as each system has evolved over time. Likewise, any results given in this thesis is also a snapshot in time for any given technology.

4.2 The AppScale Database Support and Portability Layer

In this chapter, we investigate a database-agnostic software layer for cloud platforms that decouples the datastore interface from its implementation(s) and automates distributed deployment of datastore systems. We design and implement this layer as

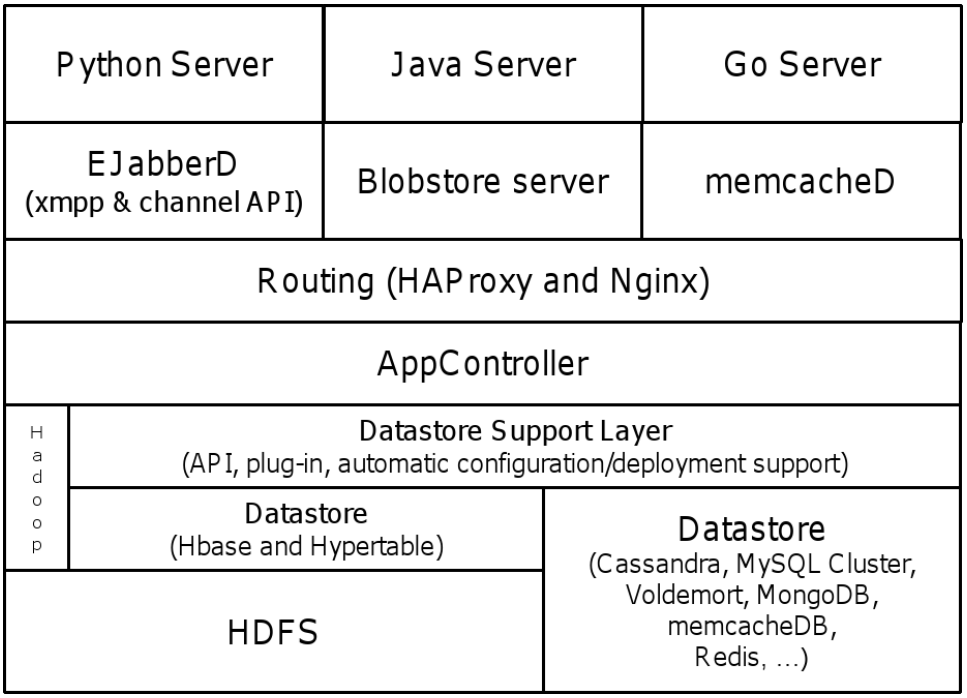


Figure 4.1: The AppScale Software Stack. This chapter presents the design and implementation of the database software layer and its extensions in support of distributed, database-agnostic, multi-key transactional semantics.

part of the AppScale cloud platform and then extend it to support database-agnostic distributed transaction support.

Figure 4.1 shows the AppScale software stack. At the top of the stack are the application servers that serve Python, Java, and Go applications. The AppScale APIs that the applications employ leverage existing open source software such as eJabberD [32] and memcacheD [63], or custom services (e.g. blobstore) that we provide, for their implementations. AppScale uses Nginx [71] and HAProxy [44] to route and load balance requests to the application servers. Nginx provides SSL connections, and HAProxy

performs health checks on servers, routing only to responsive application servers. A background service on each node in AppScale restarts any service that stops functioning correctly. An AppScale cloud consists of a set of virtual machine instances (nodes) working together in a distributed system, each of which implement this software stack.

The AppController is a software layer in the stack that is in charge of service initiation, configuration, and heart beat monitoring, cloud-wide. Below the AppController is the database-agnostic software layer (to which we refer to as the datastore support layer in the figure).

Our new datastore support layer decouples application access to structured data from its implementation. It is this layer we extend with ACID transaction semantics in the next section. This layer exports a simple yet universal key-value programming interface that we implement using a wide range of available datastore technologies. This layer provides portability for applications across datastores, i.e. applications written to access this datastore interface will work with any datastore that implements this interface, without modification. The interface provides full GAE functionality and consists of:

- Put(table, key, value)
- Get(table, key)
- Delete(table, key)

- Query(table, q)

`Put` stores the value given the key and creates a table if one does not already exist. If a `Get` or `Query` is performed on a table which does not exist, nothing is returned. A `Delete` on a key which does exist results in an exception. *Query* uses the Google Query Language (a subset of SQL) syntax and semantics.

The data values that AppScale stores in the datastore are called entities (data objects) and are similar to those defined by GAE [90]. Each entity has a key object; AppScale stores each entity according to its key as a serialized protocol buffer [78].

We implement the datastore API in AppScale using popular open source, distributed datastore systems. These include HBase [46], Hypertable [50], Cassandra [14], Redis [81], Voldemort [88], MongoDB [65], SimpleDB [84], and MySQL Cluster [69]. HBase and Hypertable both rely on HDFS [42] for their distributed file system implementations, as does the Map-Reduce API which integrates Hadoop MapReduce [43] support.

Each AppScale cloud deployment implements a single datastore (cloud-wide). The AppController in the system interacts with a template that configures and deploys each datastore dynamically upon cloud instantiation. The set of scripts configure, start, stop, and test an instantiated datastore using the following API:

- start_db_master()

- start_db_slave()
- setup_db_config_files(master_ip, slave_ips, creds)
- stop_db_master()
- stop_db_slave()

Each datastore must implement these calls. To set up the configuration files, the AppController provides template files and inserts node names as appropriate. The ”creds” argument is a dictionary in which additional, potentially datastore-specific, arguments are passed, e.g. the number of replicas to use for fault tolerance.

4.3 Database-Agnostic Distributed Transaction Support

We next extend the datastore support layer in the cloud platform with ACID transaction semantics. We refer to this extension as database-agnostic transactions (DAT). Such support is key for a wide range of applications that require atomic updates to multiple keys at a time. Thus, we provide it in a database-agnostic fashion that is independent of any datastore but that can be used by all datastores that plug into the database support layer.

4.3.1 DAT Design

To enable DAT, we extend the AppScale datastore API with support for specifying the boundaries of a transaction programmatically. To ensure GAE compatibility, we use the GAE syntax for this API:

```
run_in_transaction
```

which defines the transaction block.

We make three key assumptions in the design of DAT. First, we assume that each of the underlying datastores provide strong consistency. Most extant datastores provide strong consistency either by default (e.g. HBase, Hypertable, MySQL-cluster) or as a command-line option (e.g. Cassandra). Second, we assume that any datastore that plugs into the DAT layer provides row-level atomicity. All the datastores we have evaluated provide row-level atomicity, where any row update provides all-or-nothing semantics across the row's columns. Third, we assume that there are no global or table-level transactions; instead, transactions can be performed across a set of related entities. We impose this restriction for scalability purposes, specifically to avoid slow, coarse-grain locking across large sections or tables of the datastore.

To enable multi-entity transactional semantics, we employ the notion of entity groups as implemented in GAE [90]. Entity groups consist of one or more entities, all of whom

share the same ancestor. This relationship is specified programmatically. For example, the Python code for an application that specifies such a relationship looks as follows:

```
class Parent(db.Model):
  balance = db.IntegerProperty()
class Child(db.Model):
  balance = db.IntegerProperty()
p = Parent(key_name="Alice")
c = Child(parent=p, key_name="Bob")
```

A class is a model that defines a `kind`, an instance of a kind is an entity, and an entity group consists of a set of entities that share the same root entity (an entity without a parent) or ancestor. In addition, entity groups can consist of multiple `kinds`. An entity group defines the transactional boundary between entities.

The keys for each of these entities are:

`app_id\Parent:Alice`

and

`app_id\Parent:Alice\Child:Bob`

for p (Alice) and c (Bob), respectively. Alice is a root entity with attributes type (kind), `key_name` (a reserved attribute), and balance. The key of a non-root entity, such as Bob, contains the name of the application and the entire path of its ancestors, which for this example, consists of only Alice. It is possible to have a deeper hierarchy of entities

as well. AppScale prepends the application ID to each key to enable multitenancy for datastores which do not support dynamic table creation and thus share one key space.

A transactional work-flow in which a program transfers some monetary amount from the parent entity to the child entity is specified programmatically as:

```
def give_allowance(src, dest, amount):
  def tx()
    p = Parent.get_by_key_name(src)
    c = Child.get_by_key_name(dest)
    p.balance = p.account - amount
    p.put()
    c.balance = c.balance + amount
    c.put()
  db.run_in_transaction(tx)
```

A transaction may compose *gets*, *puts*, *deletes* and *queries* within a single entity group. Any entity without a parent entity is a root entity; a root entity without child entities is alone in an entity group. Once entity relationships are specified they cannot be changed.

4.3.2 DAT Semantics

DAT enforces ACID (atomicity, consistency, isolation, and durability) semantics for each transaction. To enable this, we use multi-version concurrent control (MVCC) [8]. When a transaction completes successfully, the system attempts to commit any changes that the transaction procedure made and updates the *valid version number* (the last committed value) of the entity in the system. The operations *put* or *delete* outside of a programmatic transaction are transparently implemented as transactions. If a transaction cannot complete due to either a program error or lock timeout, the system rolls back any modifications that have been made, i.e., DAT restores the last valid version of the entity.

A read (*get*) outside of a programmatic transaction accesses the valid version of the entity, i.e., reads have "read committed" isolation. Within a transaction, all operations have serialized isolation semantics, i.e., they see the effects of all prior operations. Operations outside of transactions and other transactions see only the latest valid version of the entity.

The implementation of transaction semantics GAE and AppScale differ, each having their own set of trade-offs. GAE implements transactions using optimistic concurrency control [9]. If a transaction is running, and another one begins on the same entity group, the prior transaction will discover its changes have been disrupted, forcing a retry. An entity group will experience a drop in throughput as contention on a group

grows. The rate of serial updates on a single root entity, or an entity group depends on the update latency and contention, and ranges from 1 to 20 updates per second [7].

We instead associate each entity group with a lock. DAT attempts to acquire the lock for each transaction on the group. DAT will retry three times (a default, configurable setting) and then throw an exception if unsuccessful. In contrast to GAE, we provide a fixed amount of throughput regardless of contention depending on the length of time the lock is held before being released. A rollback for an active transaction for an entity group does not get triggered when a new transaction attempts to commence for that same entity group as it does for GAE, but a transaction must acquire the lock in DAT before moving forward, a restriction GAE does not have. In practice, our locking mechanism is simple, works well, and provides sufficient throughput in private cloud settings which always consist of orders of magnitude fewer machines than Google's public cloud.

We also have designed DAT to handle faults at multiple levels, although we do not handle Byzantine faults. Failure at the application level is detected by a timeout mechanism. We reset this timeout each time the application attempts to modify the datastore state to avoid prolonged stalls. We also prevent silent updates and failures at the database support layer and describe this further in the next section.

4.4 DAT Implementation

To implement DAT within AppScale, we provide support for entities, an implementation of the programmatic datastore interface for transactions (*run_in_transaction*), and multi-version consistency control and distributed transaction coordination (global state maintenance and locking service). To support entities, we extend the AppScale key-assignment mechanism with hierarchical entity naming and implement entity groups. Each application that runs in AppScale owns multiple entity tables, one for each entity `kind` it implements. We create each entity table dynamically when a *put* is first invoked for a new entity type. In contrast, GAE designates a table for all entity types, across all applications. We chose to create tables for each entity kind to provide additional isolation between applications.

We implement an adaptation of multi-version consistency control to manage concurrent transactions. Typically timestamps on updates are used to distinguish versions [8]. However, not all datastores implement timestamp functionality. We thus employ a different, database agnostic, approach to maintaining version consistency. First, with each entity, we assign and record a version number. This version number is updated each time the entity is updated. We refer to this version number as the *transaction ID* since an update is associated with a transaction. We maintain transaction

IDs using a counter per application. Each entry in an entity table contains a serialized protocol buffer and transaction ID.

To enable multiple concurrent versions of an entity, we use a single table, which we call the *journal*, to store previous versions of an entity. AppScale applications do not have direct access to this table. We append the transaction ID (version number) to the entity row key (in AppScale it is the application ID and the entity row key) which we use to index the journal.

4.4.1 Distributed Transaction Coordinator (DTC)

To enable distributed, concurrent, and fault tolerant transactions, DAT implements a Distributed Transaction Coordinator (DTC). The DTC provides global and atomic counters, locking across entity groups, transaction blacklisting support, and a verification service to guarantee that accesses to entities are made on the correct versions.

The DTC enables this through the use of ZooKeeper [94], an open source, distributed locking service that maintains consistent copies of data in a distributed setting via the Paxos algorithm [59, 15]. ZooKeeper is the open source equivalent to Google's Chubby locking service [13] which is fault tolerant and provides strong consistency for the data it stores. The directory service allows for the DTC to create arbitrary paths, on which both leaves and branches can hold values.

The API for the DTC is

- `txn_id getTransactionID(app_id)`
- `bool acquireLock(app_id, txn_id, root_key)`
- `void notifyFailedTransaction(app_id, txn_id)`
- `txn_id getValidTransactionID(app_id, previous_txn_id, row_key)`
- `bool registerUpdateKey(app_id, current_txn_id, target_txn_id, entity_key)`
- `bool releaseLock(app_id, txn_id)`
- `block_range generateIDBlock(app_id, root_entity_key)`

DAT intercepts and implements each transaction made by an application (*put*, *delete*, or programmatic transaction) as a series of interactions with the DTC via this API. A transaction is first assigned a transaction ID by the DTC (`getTransactionID`) which returns an ID with which all operations that make up the transaction are performed. Second, DAT obtains a lock from the DTC (`acquireLock`) for the entity

group over which the operation is being performed. For each operation, DAT verifies that all entities accessed have valid versions (`getValidTransactionID`). For each *put* or *delete* operation, DAT registers the operation with the DTC. This allows the DTC to track of which entities within the group are being modified, and, in the case where the application forces a rollback (applications can throw a rollback exception within the transaction function) or any type of failure, the DTC can successfully know what the current correct versions of an entity are. The API call of `registerUpdateKey` is how previously valid states are registered. This call takes as arguments the current valid transaction number, the transaction number which is attempting to apply changes, and the root entity key to specify the entity group.

When a transaction completes successfully or a rollback occurs (due to an error during a transaction, application exception, or lock timeout), DAT notifies the DTC which releases the lock on that entity group, and the layer notifies the application appropriately. We set the default lock timeout to be 30 seconds (it is configurable). DAT notifies the application via an exception.

Transactions that start, modify an entity in the entity table, and then fail to commit or rollback due to either a failure, thrown exception, or a timeout, are *blacklisted* by the system. If an application attempts to perform an operation that is part of a blacklisted transaction, the operation fails and DAT returns an exception to the application. Application servers that issue operations for a blacklisted transaction must retry their

transaction under a new transaction ID. Any operations which were executed under a failed transaction are rolled back to the previous valid state.

Every operation employs the DTC for version verification. A *get* operation will fetch from an entity table which returns the entity and a transaction ID number. DAT checks with the DTC whether the version is valid (i.e., is not on the blacklist and is not part of an uncommitted, on-going transaction). If the version is not valid, the DTC returns the valid transaction ID for the entity and DAT uses this ID with the original key to read the entity from the journal. *Get* operations outside of a transaction are read-committed as a result of this verification (we do not allow for `dirty` reads). The result of a query must follow this step for each returned entity. Both GAE and AppScale recommend that applications keep entity groups small as possible to enable scaling (parallelizing access across entity groups) and to reduce bottlenecks.

Lone *puts* and *deletes* are handled as if they were individually wrapped programmatic transactions. For a *put* or *delete* the previous version must be retrieved from the entity table. The version returned could potentially not exist because the entry was previously never written to and thus we assign it zero. The version number is checked to see if it is valid, if it is not, the DTC returns the current valid number. The valid version number is used for registration to enable rollbacks if needed.

Either using the original version (transaction ID) or the transaction ID returned from the DTC due to invalidation, DAT creates a new journal key and journal entry

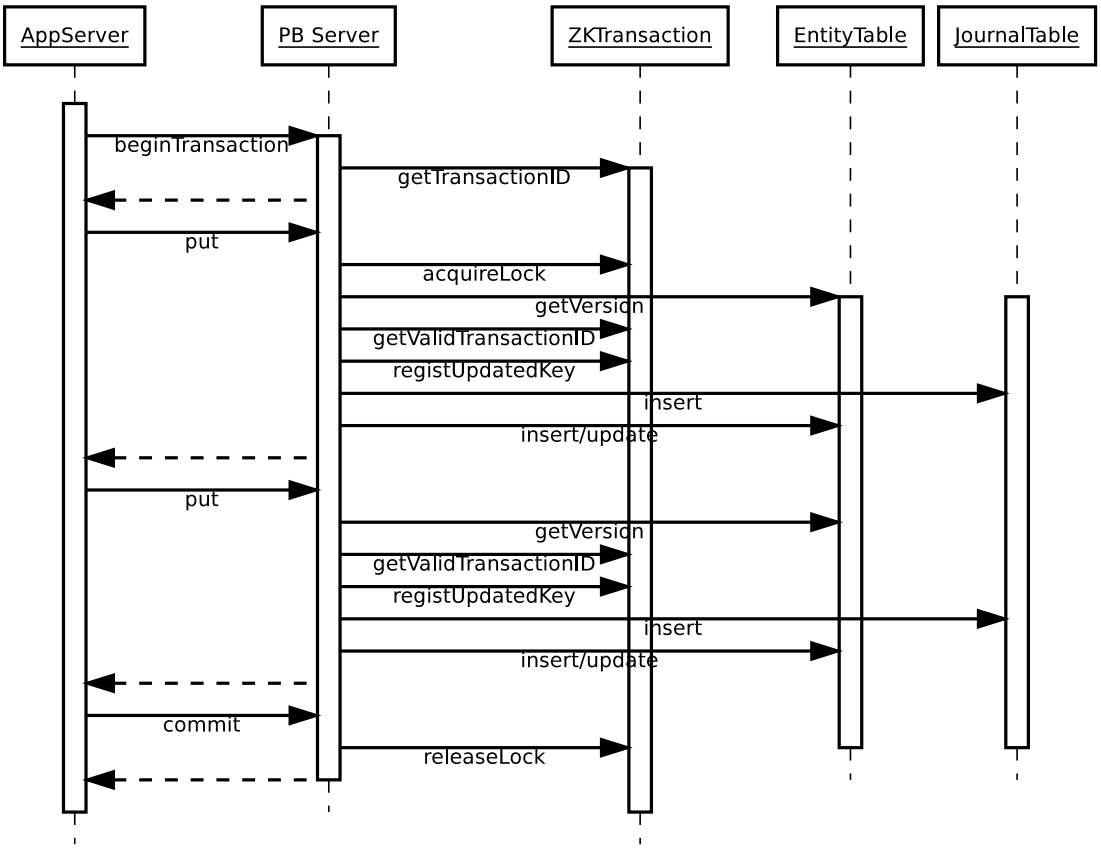


Figure 4.2: Transaction sequence example for two puts.

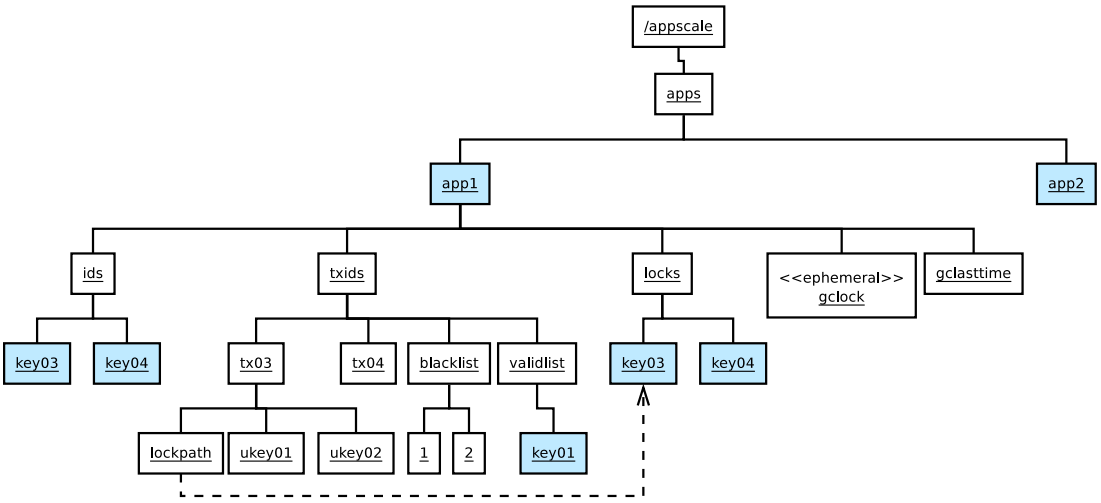


Figure 4.3: Structure of transaction metadata in ZooKeeper nodes.

(journal keys are guaranteed to be unique), registers the journal key with the DTC, and in parallel performs an update on the entity table. We overview these steps with an example in Figure 4.2 and show the DTC API being used during the lifetime of a transaction where two *put* operations take place. It illustrates the transaction starting where a transaction ID is attained; a *put* request then triggers the acquisition of a lock, version validation, key registration for rollback, and entity updates. The second *put* repeats the same steps sans lock acquisition. Lastly, the transaction is committed.

DAT does not perform explicit *deletes*. Instead, we convert all *deletes* into *puts* and use a *tombstone* value to signify that the entry has been deleted. We place the tombstone in the journal as well to maintain a record of the current valid version. Any entries with tombstones which are no longer live are garbage collected periodically.

4.4.2 ZooKeeper Configuration of the DTC

We present the DTC implementation using the ZooKeeper node structure prefix tree (trie) in Figure 4.3. We store either a key name as a string (for locks and the blacklist) or use the node directly as an atomically updated counter (e.g., for transaction IDs). State of ZooKeeper is shared among clients, showing a strongly consistent view. The tree structure is as follows:

- /appscale/apps/app_id/ids: counter for next available transaction IDs for root or child entities.
- /appscale/apps/app_id/txids: current live transactions.
- /appscale/apps/app_id/txids/blacklist: invalid transaction ID list.
- /appscale/apps/app_id/validlist: valid transaction ID list.
- /appscale/apps/app_id/locks: transaction entity groups.

The blacklist contains the transaction IDs that have failed due to a timeout, an application error, an exception, or an explicit rollback. The valid list contains the valid transaction IDs for blacklisted entities (so that we can find/retrieve valid entities).

Transactions implemented by DAT provide transactional semantics at the entity group level. We implement a lock subtree that is responsible for mapping a transaction ID to the entity group it is operating on. The name of the lock is the root entity

key and it stores the transaction ID. We store the locking node path in a child node of the transaction named ”lockpath”. Any new transaction that attempts to acquire a lock on the same entity group will see that this file exists which will cause the acquisition to fail. This lock node is removed when a transaction completes (via successful commit or rollback).

4.4.3 Scalable Entity Keys

We employ ZooKeeper sequential nodes to implement entity counters (these should not be confused with transaction IDs). When entities are created without specifying a key name, IDs are assigned in an incremental fashion. We ensure low overhead on key assignment by allocating blocks of 1000 entity IDs at a time to reduce the overhead of counter access. The block of IDs is cached by the instance of the call handler in the database support layer. Keys are provisioned on a first-come-first-serve basis to new entities which do not have a key name. There is no guarantee that entity IDs are ordered.

Entity IDs use two types of counters for concurrent access. One counter is for root keys of a specific entity type, while another counter is created for each child of a root key. Entity IDs are stored under the inner node upon creation and are removed once committed. The node structure holds values for each entity group as seen in the

/appscale/apps/app1/ids path. The ”ids” node contains the next batch value for all root keys, while $key03$ and $key04$ nodes hold values for the next batch of child keys.

4.4.4 Garbage Collection

In some cases (application error, response time degradation, service failure, network partition, etc.), a transaction may be held for a long period of time or indefinitely. We place a maximum of 30 seconds on each lock lease acquired by applications. We update the value dynamically as needed. Furthermore, for performance reasons we use ZooKeeper's asynchronous calls where it does not break ACID semantics (i.e., removing nodes after completion of a transaction).

In the background, DAT implements a garbage collection (GC) service. The service scans the transaction list to identify expired transaction locks (we record the time when the lock is acquired). The service adds any expired transaction to the blacklist and releases the lock. For correct operation with timeouts, the system is coordinated using NTP. Nodes which were not successfully removed by an asynchronous call to ZooKeeper are garbage collected during the next iteration of the GC.

The GC service also cleans up entities and all related metadata that have been deleted (tombstoned) within a committed transaction. In addition, journal entries that contain entities older than the current valid version of an entity are also collected. We do not remove nodes in the valid version list at this time.

We perform garbage collection every thirty seconds. There is one master garbage collector and multiple slaves checking to make sure the global ”gclock” has not expired. If the lock has expired (it has been over 60 seconds since last being updated), a slave will take over as the master, and will now be in charge of periodically updating the ”gclock”. When a lock has expired, the master will receive a call back from ZooKeeper. At this point the master can try to refresh the lock, or if the lock has been taken, step down to a slave role.

4.4.5 Fault Tolerance

DAT handles certain kinds of failures, excluding byzantine faults. Our implementation of the DTC ensures that the worst case timing scenario does not leave the datastore in an inconsistent state (”Heisenbugs”) [52].

A race condition can occur due to the distributed and shared nature of the access to the datastore. Take for example the following scenario:

- The DTC acquires a lock on an entity group
- It becomes slow or unresponsive
- The lock expires
- It perform an update to the entity table
- The DTC node silently dies

In this case, we must ensure that the entity is not updated (overwritten with an invalid version). We detect and prevent such silent faults using the transaction blacklist and valid versions are retrieved from the journal.

We address other types of failures using the lock leases. Locks which are held by a faulty service in the cloud will be released by the GC. We have considered employing an adaptive timeout on an application or service basis for applications/services that repeatedly timeout. That is, reduce the timeout value for the application/service – or for individual entity groups – in such cases to reduce the potential of delayed update access. Additional state would be required that would add overhead to lookup each timeout value per entity group or application. Currently, the timeout is configurable upon cloud deployment.

Our system is designed to handle complete system failures (power outages) in addition to single/multi node failures. All writes and deletes are issued to the datastore, each write persists on disk before acknowledgment. No transaction which has been committed is lost attaining full durability (granted at least one replica survived). Meta state is also replicated in ZooKeeper for full recovery as well as the transaction journal. Replication factor is also configurable upon cloud deployment.

4.5 Methodology

In this section, we overview our benchmarks and experimental methodology. For our experiments, AppScale employs Hadoop 0.20.2-cdh3u3, HBase 0.90.4-cdh3u3, Hypertable 0.9.5.5, MySQL ndb-7.0.9, Redis 2.2.11, and Cassandra 1.0.7. We execute AppScale using a private cluster via the Eucalyptus cloud infrastructure. Our Eucalyptus private cloud consists of 12 virtual machines with 4 cores, and 7.5 GB of RAM. We also employ our benchmark on Google App Engine, where the infrastructure is abstracted away. We synchronize the clocks using the Linux tool `ntpdate` for our Eucalyptus cluster.

4.5.1 Benchmarking Application

Our benchmark measures reads and writes of each datastore where transactions are enabled as well as disabled, the difference of which gives us the overhead imposed by the DAT layer. AppScale is configured to have the head node act as a full proxy, randomly distributing request across application servers. The benchmark is run for a single node deployment (it acts as both a load balancer and runs application servers), the default four node deployment, and a 12 node deployment.

We use the Apache Benchmark tool as our load generator, which targets a URL at the head node. The datastore is first primed with 1000 entries, for which random

reads are done on. The writes use random keys, but each key always starts with the application name for isolation and lexicographical entity placement.

The Apache Benchmark tool is used with three different load levels: 10, 100, and 1000 concurrent requests. The tool measures latency and throughput for these different loads. Reported numbers are averages of 10 trials.

Each server runs ten process instances of the benchmark application. We set the replication factor to one for these experiments for all datastores, which was the common factor given our one node deployment. For GAE, Google uses its own scheduling and replication policy to enable the scaling of applications, and it is unknown how many physical servers are being employed.

4.6 Results

Figures 4.4 and 4.5 show measurements for the Cassandra datastore with a varying number of machines, for writes and reads of a concurrency level of 10, 100, and 1000, with and without transaction support. Figure 4.4(a) and Figure 4.4(b) chart latency of requests with and without transactions enabled. For all sizes of clusters we see additional latency for writes, regardless of the concurrency level when transactions are enabled. However, reads see no statistically significant overhead when enabled. Latency for reads and writes are in close range to each other when transactions are disabled, but

the overhead of transactions for writes causes asymmetrical latency. Moreover, latency drops as more nodes are added to the cluster with and without transactions.

Figure 4.5(a) has throughput of requests, while Figure 4.5(b) has the same but with transactions disabled. Writes have less throughput when transactions are enabled, while read throughput is unchanged. With the lower latency of additional nodes the throughput rises.

HBase latency is shown in Figure 4.6(a) and Figure 4.6(b), for transactions enabled and disabled, respectively. Compared to Cassandra, HBase performs similarly for latency, yet for the 12 node case, Cassandra is able to get higher throughput for reads and writes as seen in Figures 4.7(a) and 4.7(b). Both datastores see similar drop offs in throughput due to the overhead of transaction support, yet Cassandra sees more than 100 more request per second in throughput compared to HBase when transactions are disabled.

Hypertable has slightly more latency for the single node case for both reads and writes as presented in Figures 4.8(a) and 4.8(b), yet for higher node counts it is comparable to HBase. Hypertable achieves high throughput for reads with over 1000 requests per second as seen in Figure 4.9(a) and Figure 4.9(b). Hypertable, compared to HBase, gets more read and write throughput, where for high load it is over 1000 requests per second. .

Redis performance numbers are presented in Figures 4.10 and 4.11, where we see some deviation from the previously presented datastores. Redis stores data in memory (asynchronously writing to disk which loosens our consistency guarantees for ACID semantics) and does so in the master node which handles all requests. Slaves store copies of the master, yet in these experiments we set replication to one. Where the previous datastores are able to have clients and scale with larger deployments, Redis does not benefit because all request go to a single node causing saturation of the node more quickly, and hence the lower performance in throughput.

The MySQL Cluster deployment does not use the DAT layer for transaction support, but rather its own native implementation. Figure 4.12(a) shows latency numbers for reads and writes. The latency is much higher than previous datastores, along with higher variance. As more nodes are added, the latency does drop, but even at 12 nodes it is over 10 seconds for writes. Reads scale much better for 12 nodes, but with high variance.

Figure 4.13(b) shows a CDF of latency of writes and reads for transactions disabled for Cassandra on a 12 node deployment, while Figure 4.13(a) shows the same with transactions enabled. The overhead can be seen with 1000 writes, where latency is much higher. Reads do no see the same overhead, where the latency stays the same. When transactions are disabled, writes are faster than reads until the 90th percentile, where writes have a longer tail for latency.

The breakdown of an entity *put* for Cassandra is presented in Figure 4.14. Each operation which is a part of the transaction adds some amount of overhead, where the ”Puts” are parallel writes to the entity table and journal table. The majority of overhead comes from checking to see if the current key exists and, if it does, to register that transaction value for any required rollback. This figure measures it for the case where the key did not exist before, which for Cassandra is a higher latency operation than looking for a key which does exist. It should be noted that this is the highest amount of relative overhead because this looks at only a single *put*. If the transaction had multiple reads and writes, then much of the overhead associated with starting a transaction, getting a lock, releasing it, and committing is amortized.

For comparison purposes we also ran the benchmark on GAE. Figure 4.15(a) has the latency of gets and writes for different concurrency levels. Figure 4.15(b) has throughput for the same experiment, showing much less throughput than our scalable datastores, yet in these cases the round trip time is much higher (a ping averaged 27ms to our application hosted by Google) as our load generator is local to our private cluster. Round trip time for our local tests, by comparison, where sub 1ms.

4.6.1 Discussion

We find that Cassandra, HBase, and Hypertable were the most scalable, all being BigTable clones. Cassandra performed best in our study, followed by Hypertable, and

then HBase. Although Cassandra is able to do placement using random partitioning for range query support it requires lexicographical partitioning. Because data is placed based on key names, we see that data is stored on a single node until a tablet server is split and stored on another node. If the data set is based out of one tablet we see certain nodes can become hotspots causing slowdown if the number of clients becomes very large. Larger scale deployments were attempted, but due to the aforementioned placement of data, we found additional nodes saw no improvement in terms of throughput.

MySQL Cluster had much higher latency and lower throughput than the NoSQL stores. MySQL is at a disadvantage as it is not aware of the entity group abstraction. Hence, it uses course grain locks which limit the throughput of updates.

Throughput of transactions for *puts* does drop by as much as 50% compared to transactions being disabled. AppScale allows developers who do not require transaction semantics, and hence the required overhead, to disable them by the use of namespaces. Any namespace which preprends *notrans* will give direct access to the datastore, where access to DAT is circumvented. Any application which uses transaction semantics will still work although no ACID correctness guarantees are given.

Read throughput remains unchanged when transactions are enabled, as the only overhead associated with the read is to check to make sure the version of the entity is not from an on going transaction, or a black listed transaction. Many systems and workloads are read heavy, one example of which comes from the Megastore paper [7] which

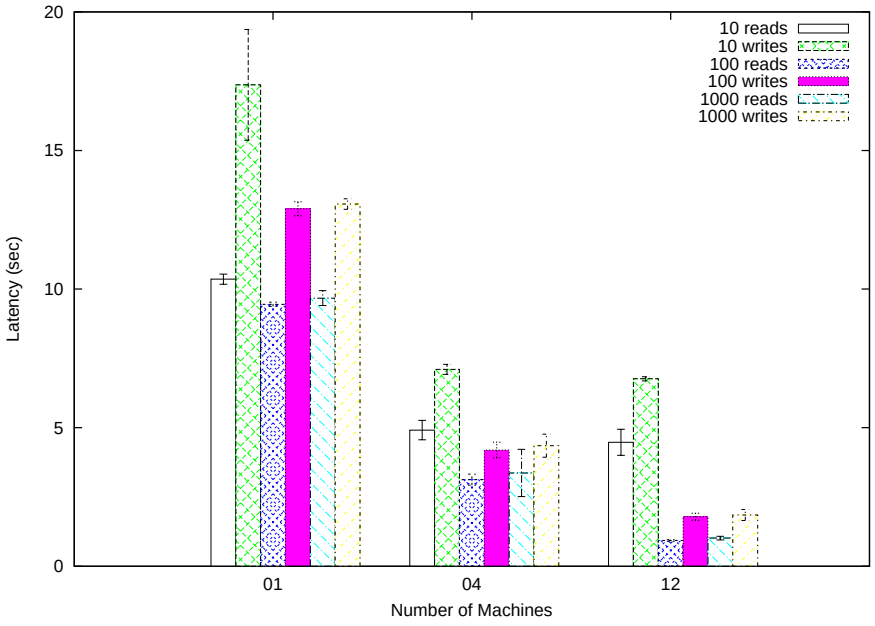
used 20:1 read to write ratios for their evaluations similar to what they see internally at Google.

4.7 Summary

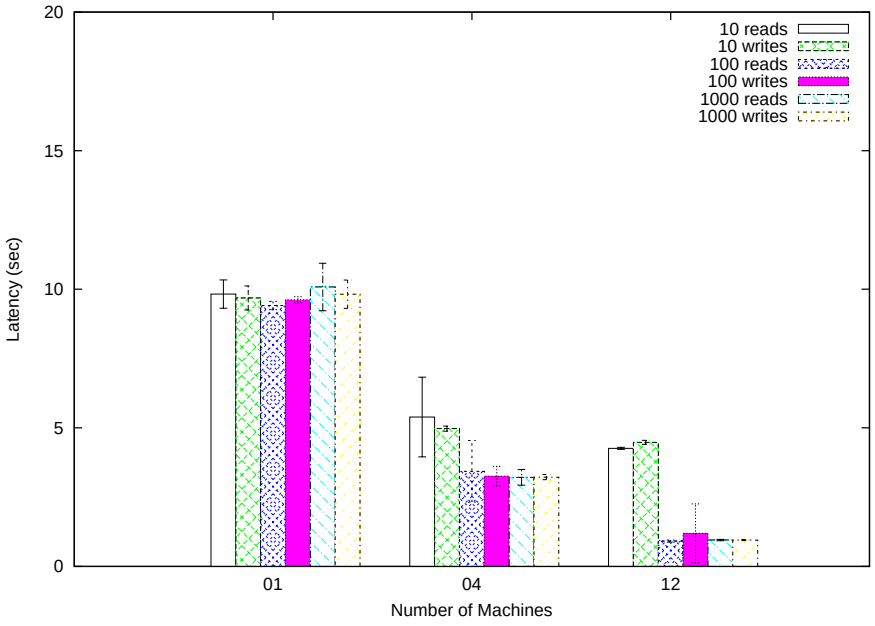
In this chapter, we investigate the trade offs of providing cloud platform support for multiple distributed datastores automatically and portably. To enable this we design and implement a database support layer, i.e. a cloud datastore portability layer, that decouples the datastore interface from its implementation(s), load-balances across datastore entry points in the system, and automates distributed deployment of popular datastore systems. Developers write their application to use our datastore API and their applications execute using any datastore that plugs into the platform, without modification, precluding lock-in to any one public cloud vendor. This support enables us to compare and contrast the different systems for different applications and usage models and enables users to select across different datastore technologies with less effort and learning curve.

We extend this layer to provide distributed ACID transaction semantics to applications, independent and agnostic of any particular datastore system and that does not require any modifications to the datastore systems that plug into our cloud portability layer. These semantics allow applications to update atomically multiple key-value pairs programmatically. We refer to this extension as DAT for database-agnostic transactions.

Since no open source datastore today provide such semantics, this layer facilitates their use by new applications and application domains including those from the business, financial, and data analytic communities, that depend upon such semantics. We implement this layer within the open source AppScale cloud platform. The next chapter further extends our datastore middleware by adding secondary index support to provide developers an SQL-like query language to access their data.

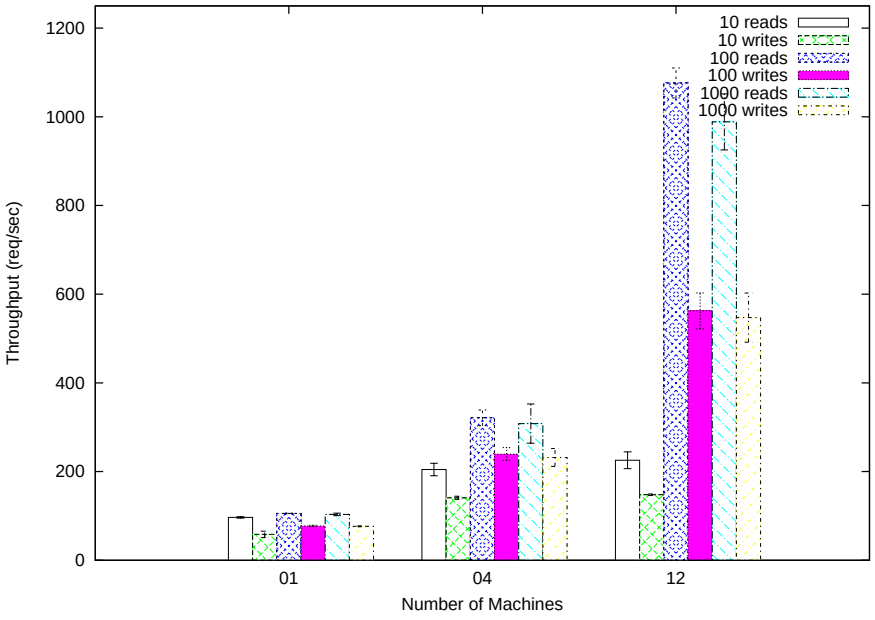


(a) Latency for Cassandra with transactions enabled.

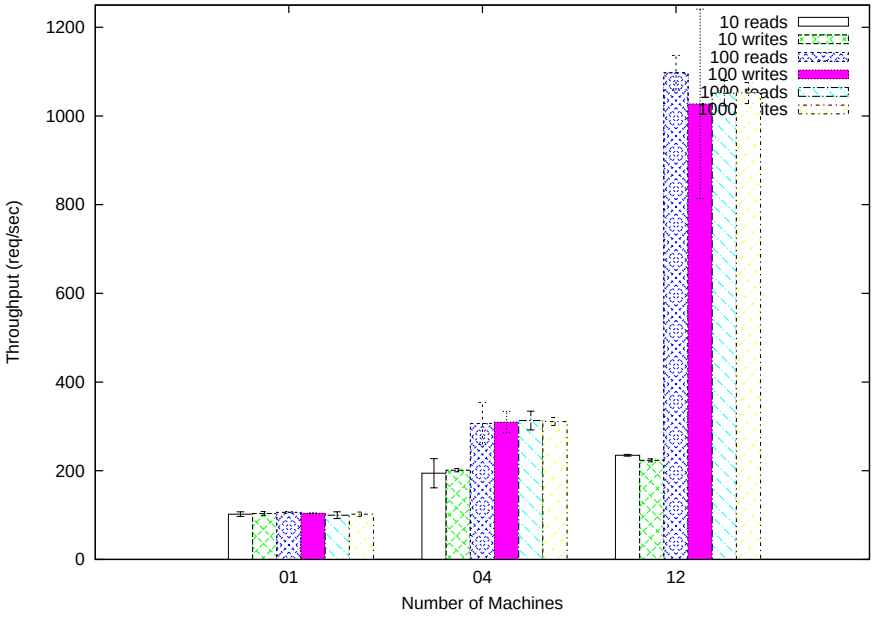


(b) Latency for Cassandra with transactions disabled.

Figure 4.4: Cassandra latency as the number of machines increases.

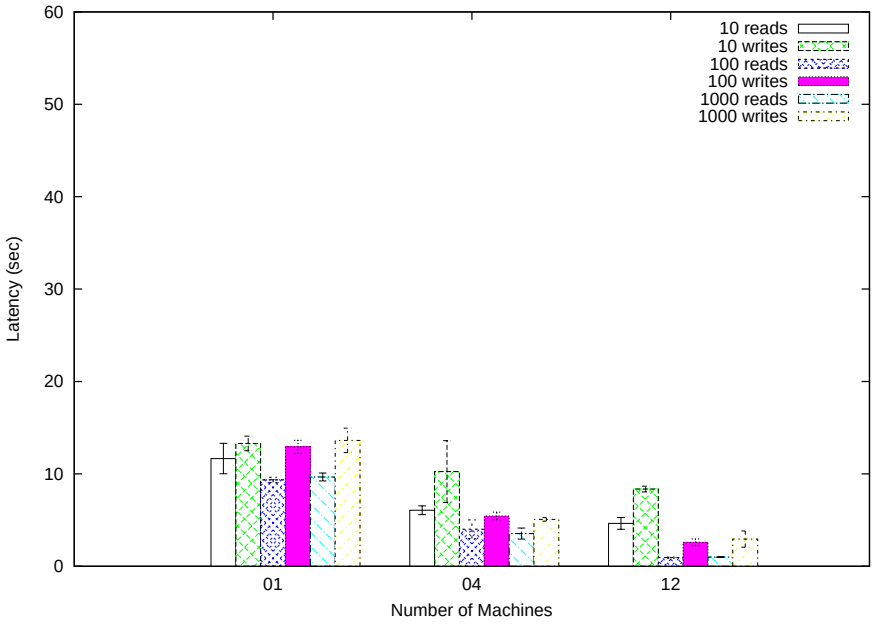


(a) Throughput for Cassandra with transactions enabled.

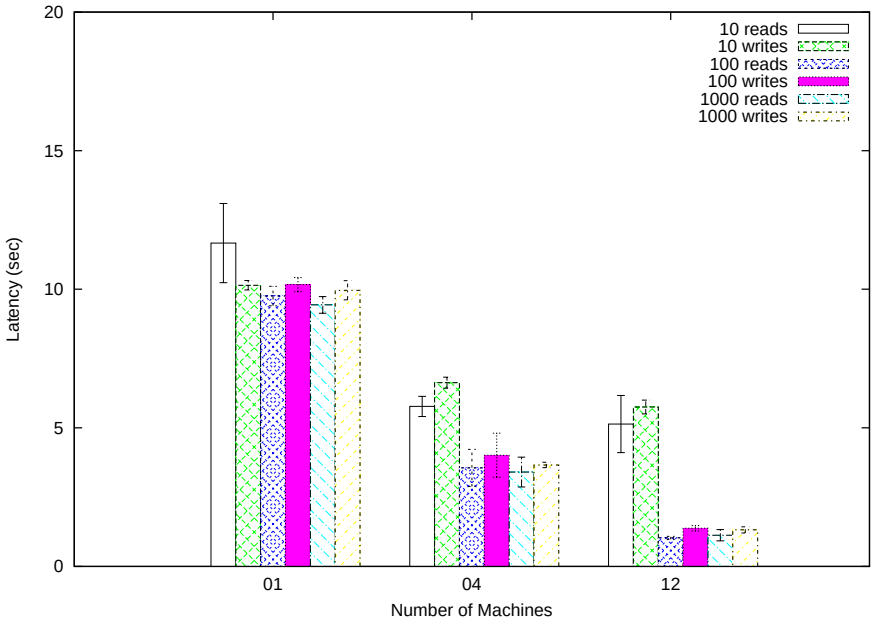


(b) Throughput for Cassandra with transactions disabled.

Figure 4.5: Cassandra results as the number of machines increases.

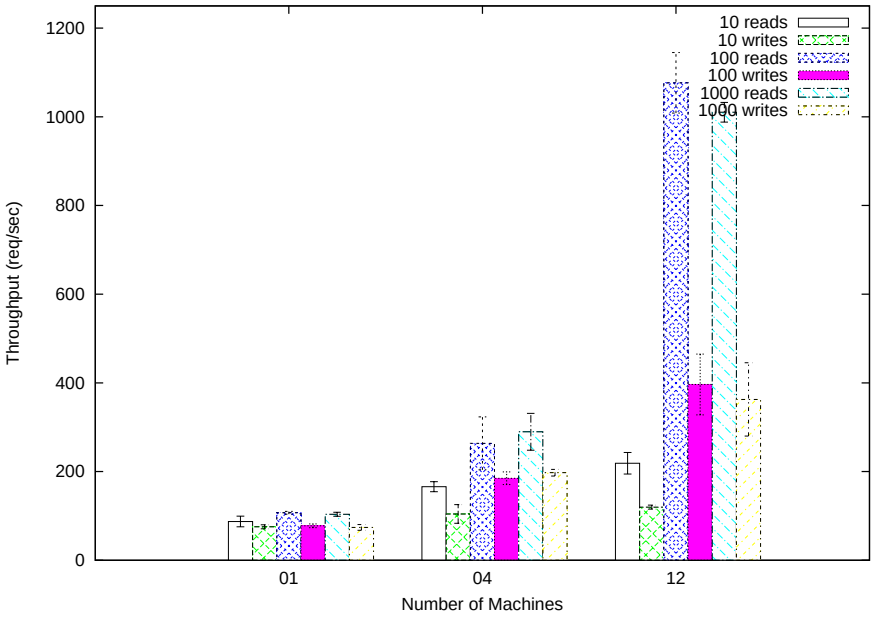


(a) Latency for HBase‘with transactions enabled.

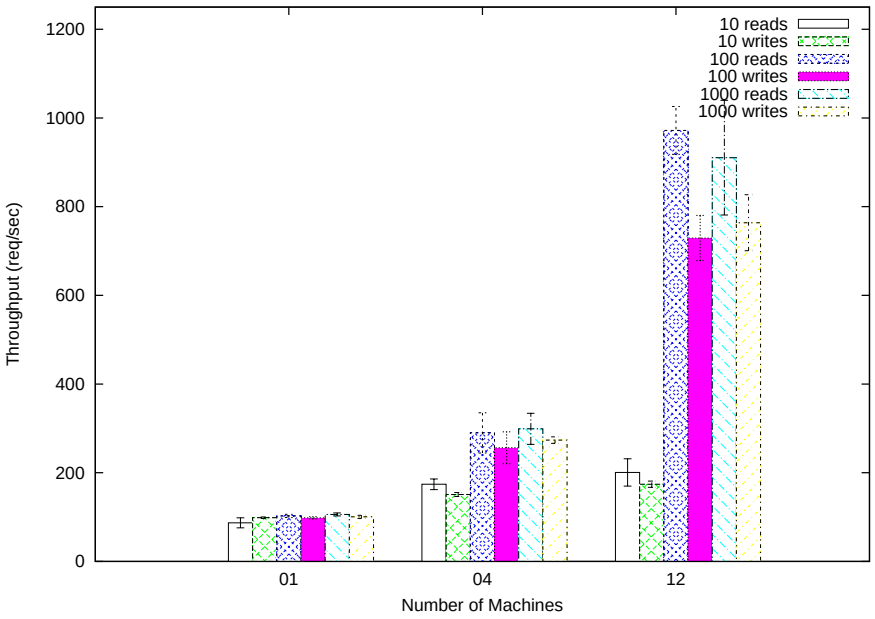


(b) Latency for HBase with transactions disabled.

Figure 4.6: HBase latency as the number of machines increases.

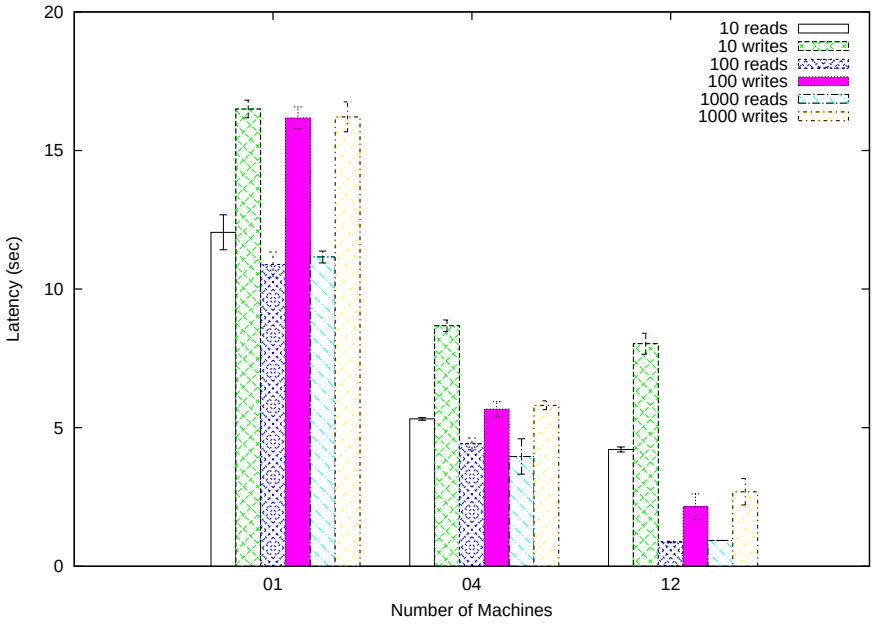


(a) Throughput for HBase with transactions enabled.

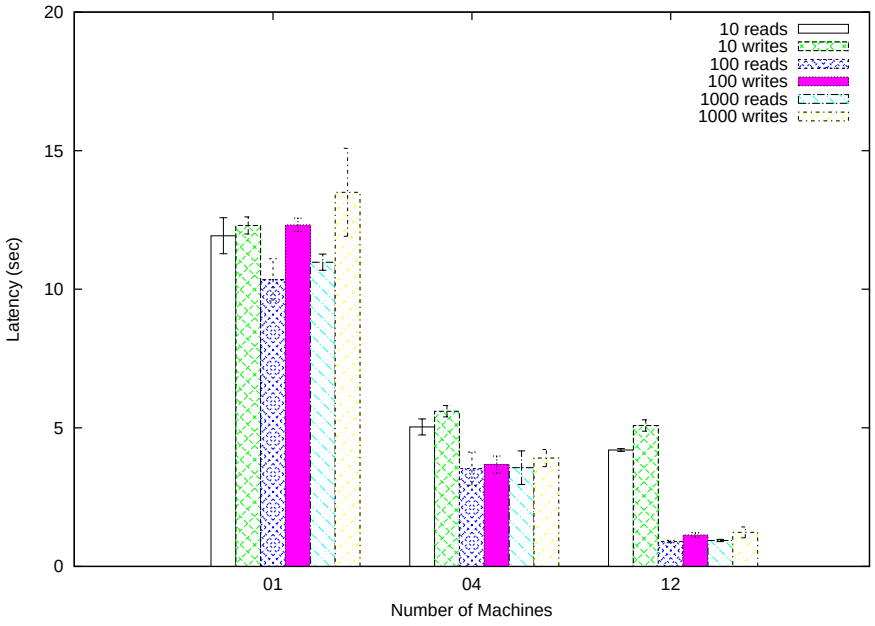


(b) Throughput for HBase with transactions disabled.

Figure 4.7: HBase throughput as the number of machines increases.

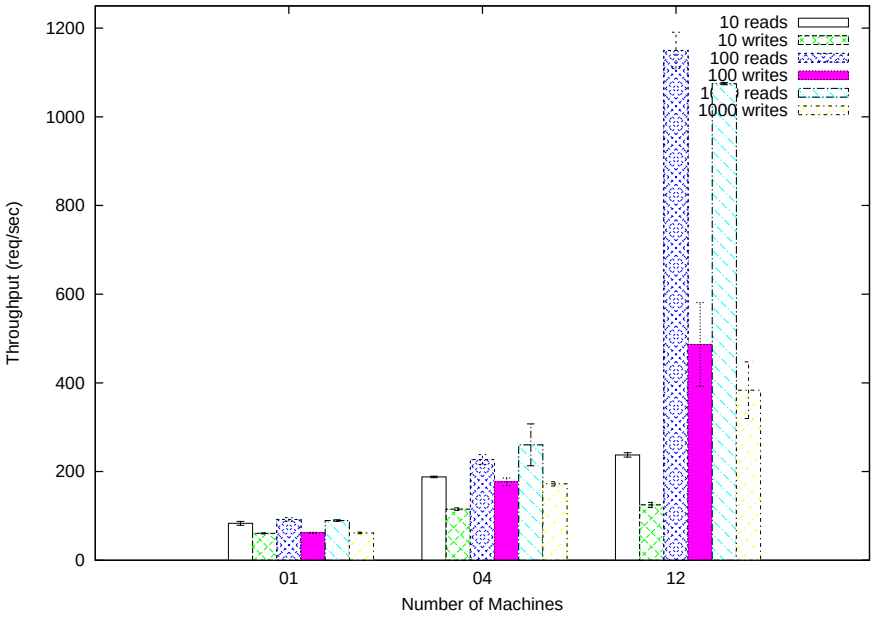


(a) Latency for Hypertable‘with transactions enabled.

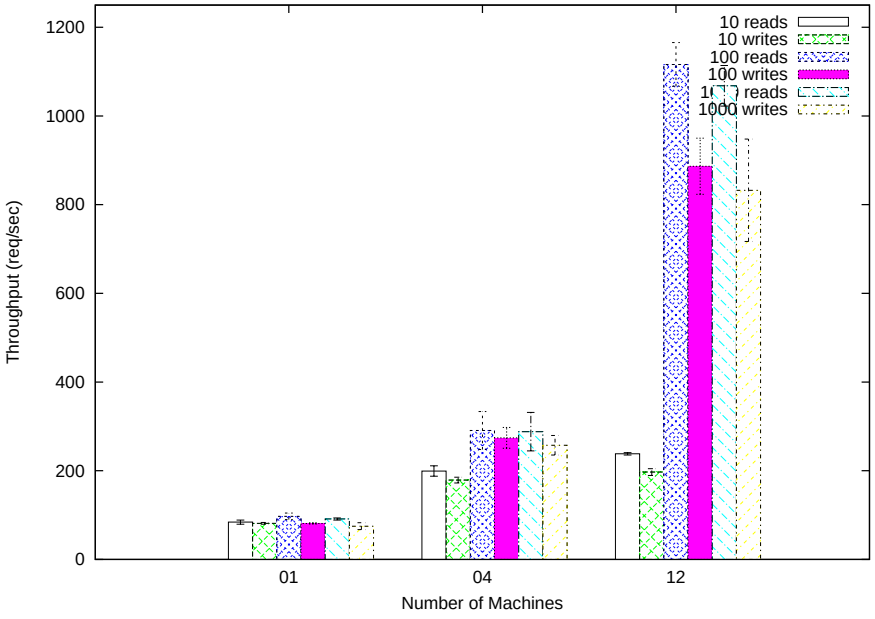


(b) Latency for Hypertable with transactions disabled.

Figure 4.8: Hypertable latency as the number of machines increases.

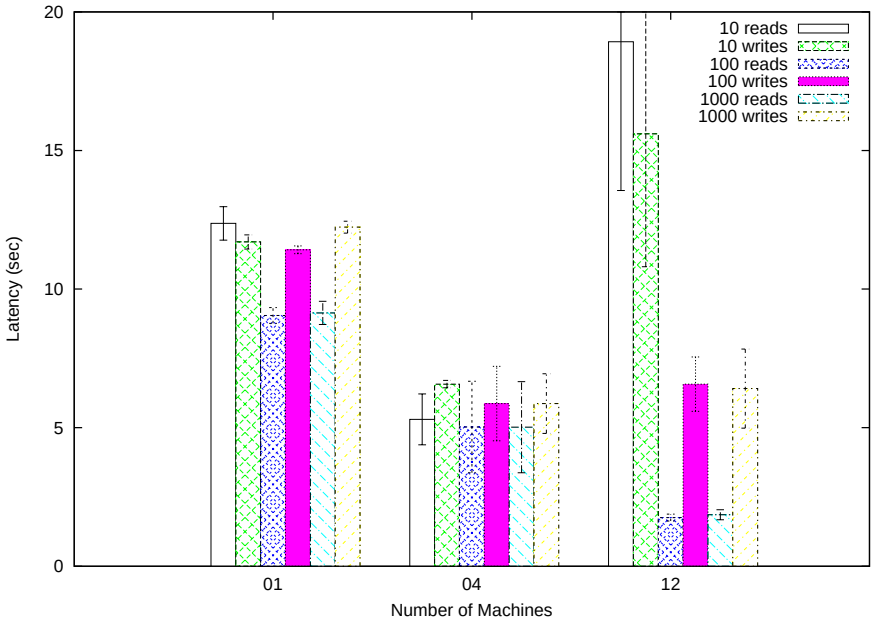


(a) Throughput for Hypertable with transactions enabled.

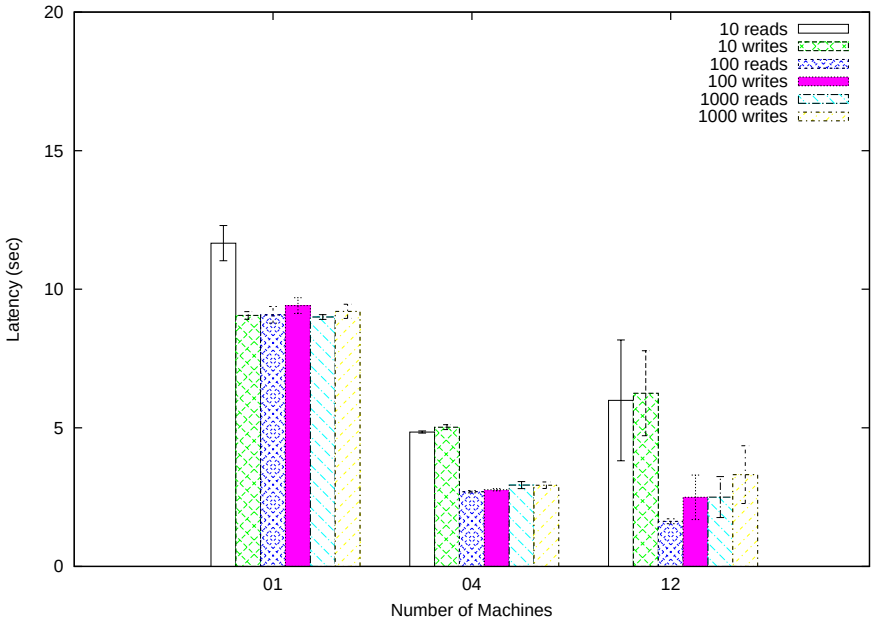


(b) Throughput for Hypertable with transactions disabled.

Figure 4.9: Hypertable results as the number of machines increases.

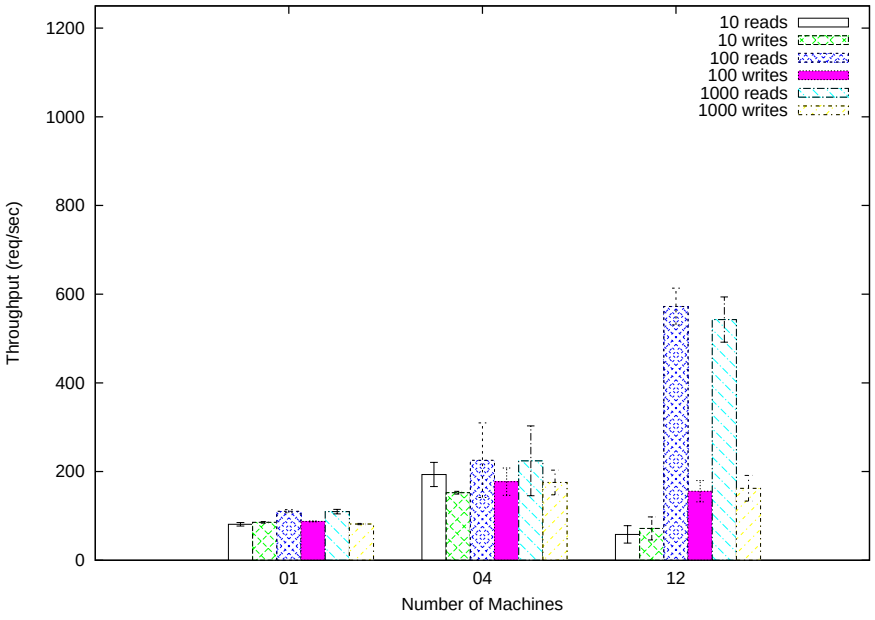


(a) Latency for Redis‘with transactions enabled.

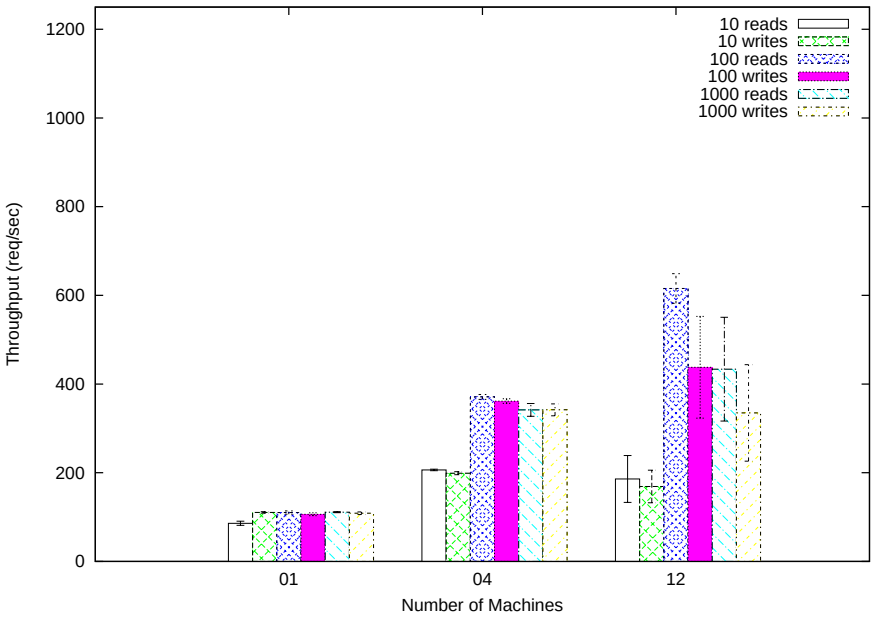


(b) Latency for Redis with transactions disabled.

Figure 4.10: Redis latency as the number of machines increases.

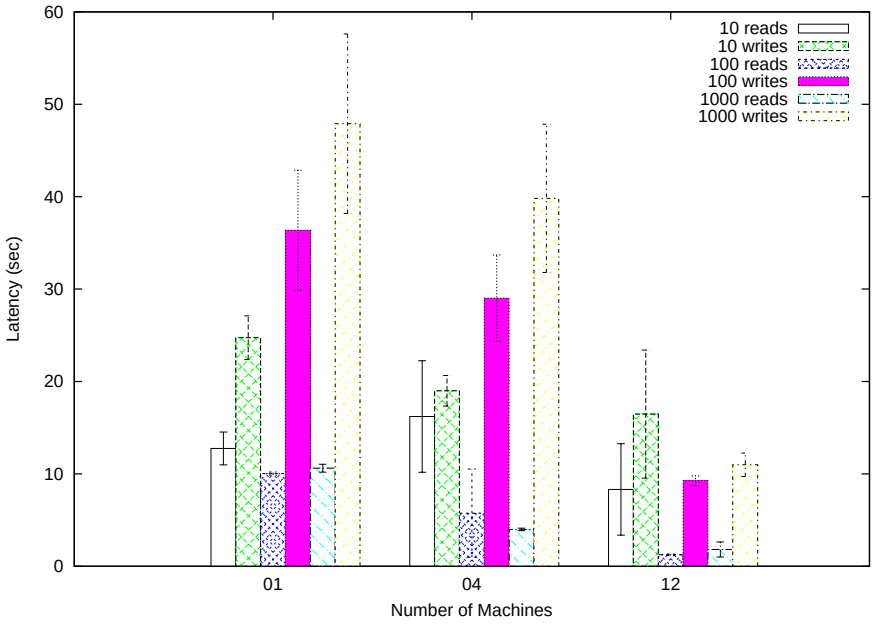


(a) Throughput for Redis with transactions enabled.

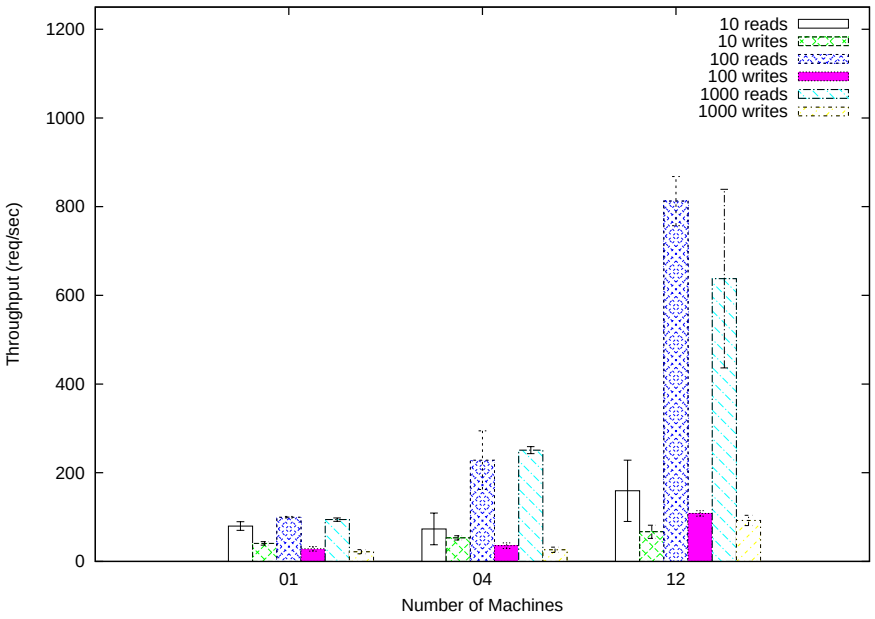


(b) Throughput for Redis with transactions disabled.

Figure 4.11: Redis throughput as the number of machines increases.

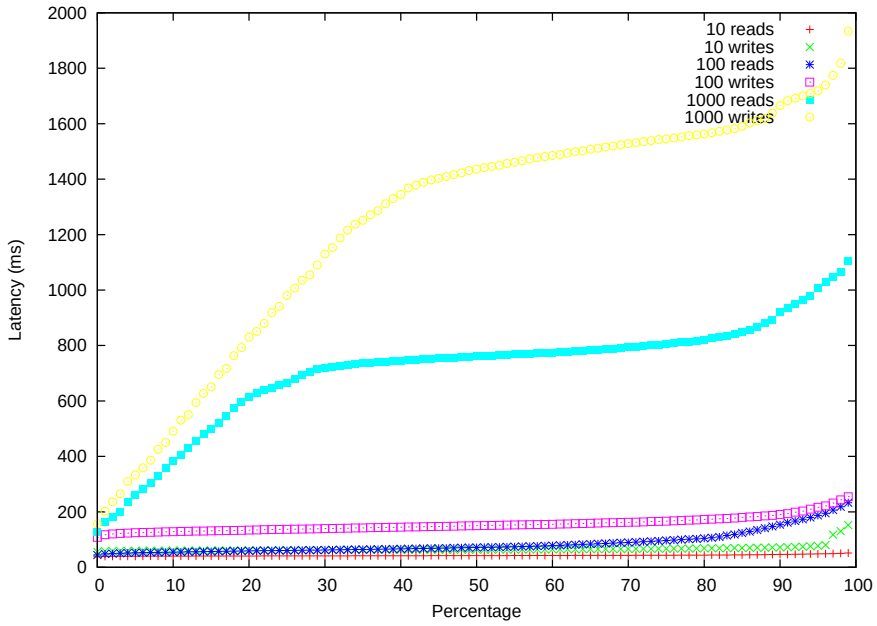


(a) Latency for MySQL with native transactions.

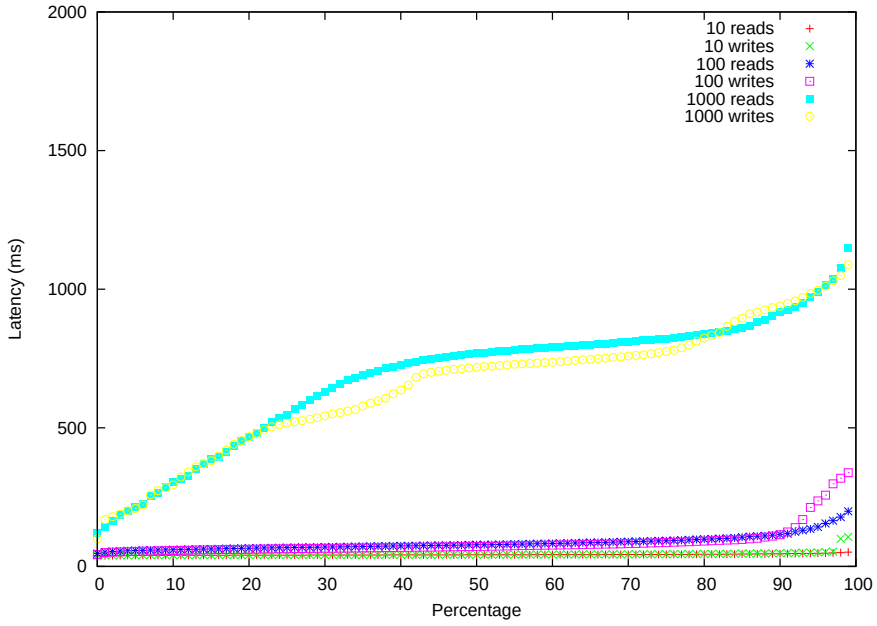


(b) Throughput for MySQL with native transactions.

Figure 4.12: MySQL results as the number of machines increases.



(a) Transactions enabled.



(b) Transaction disabled.

Figure 4.13: Latency CDFs for Cassandra 12 nodes for reads and writes.

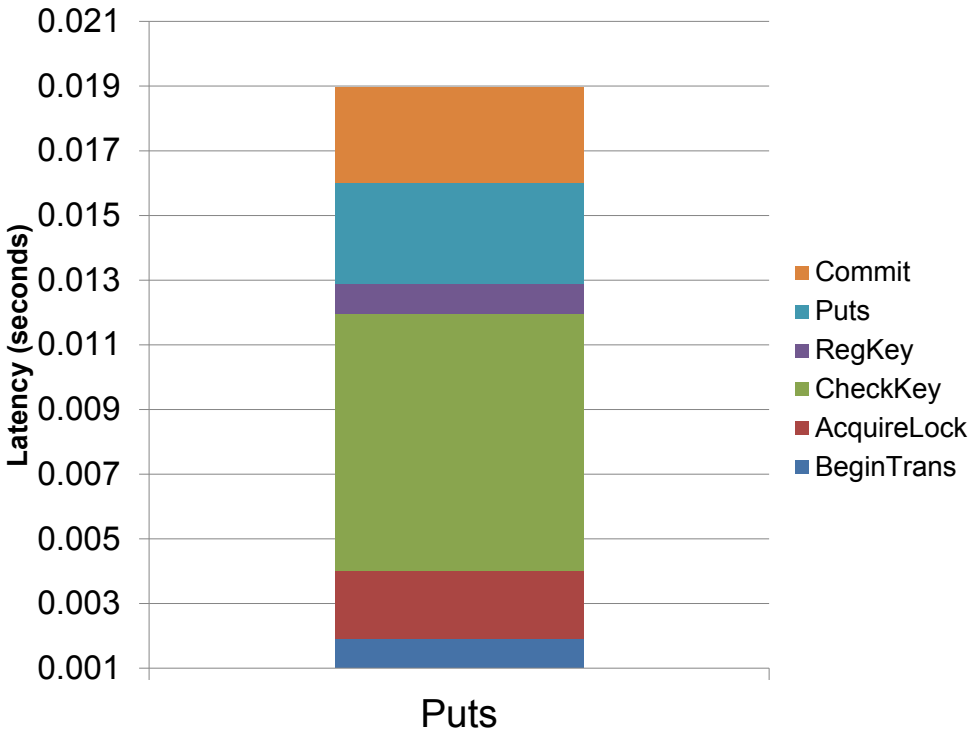
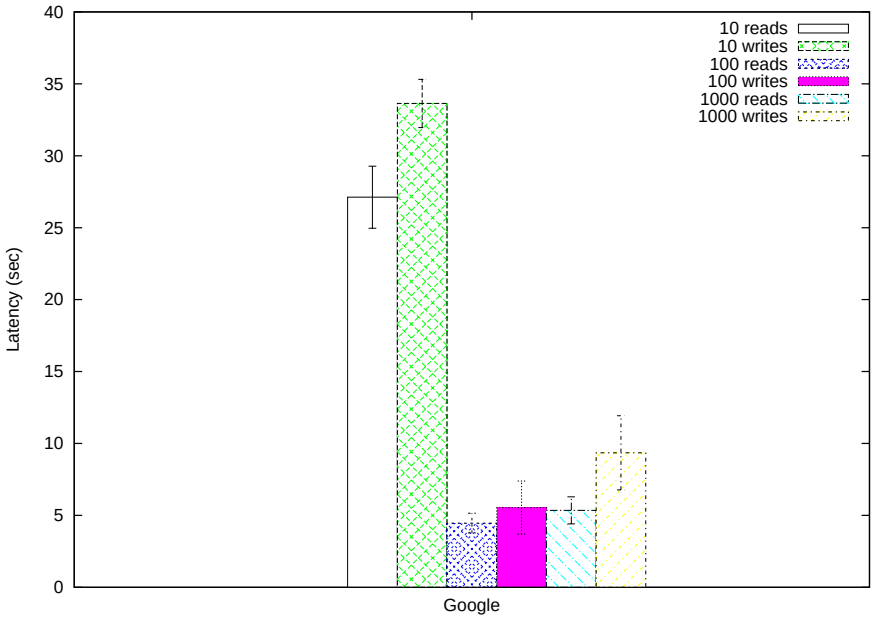
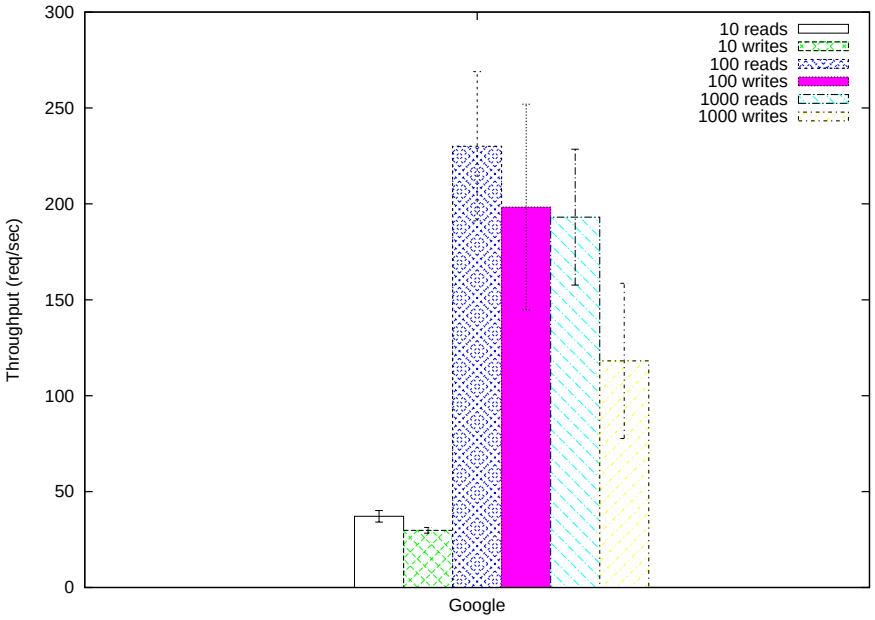


Figure 4.14: Time breakdown of an entity *put*.



(a) Latency for Google App Engine.



(b) Throughput for Google App Engine.

Figure 4.15: Google App Engine results with auto-scaling.

Chapter 5

Scalable Queries with Indexing Support

With the advent of cloud computing and the growing popularity of software-as-a-service (SaaS) offerings (the ability to access applications via remote resources), datasets have exploded in size and number. Public cloud providers increasingly provide pay-per-use and on-demand data management services that scale both in terms of the amount of data that can be stored, as well as the rate with which they are accessed, and which facilitate access via simple and portable REST interfaces.

Many popular public storage services are available such as Amazon's Simple Storage Service (S3) which provides four 9's availability with eleven 9's durability and Amazon's Reduced Redundancy Storage (RRS) which also provides the same service with less reliability. Microsoft offers its Table service which can store large amount of unstructured data with access through a REST interface, while Rackspace has Cloud

Files for on-demand storage and content delivery [80, 4, 2]. Services such as these provide a simple key/value interface for storing and retrieving data.

Google's storage system is BigTable [17], a column-oriented key/value store, which runs many internal large-scale applications. Since BigTable's publication in 2006, there have been open source implementations emulating its data model. These projects include HBase [46], Hypertable [50], and Accumulo [1]. Cassandra, a project initially developed at Facebook, provides a data model similar to BigTable but employs an peer-to-peer architecture similar to Amazon's Dynamo datastore [14]. In addition, other highly scalable datastores have emerged that provide alternative data modules such as document stores (MongoDB), key/value stores (Voldemort), and graph databases (Neo4J). Currently there are over 122 different offerings [72], each with their own API, options, deployment requirements, and idiosyncrasies. These non-relational datastores are referred to by the community as "NoSQL" datastores.

These cloud-based NoSQL datastores offer an alternative to more traditional relational databases which provide a very expressive query language, but are not designed for the same use cases and scale (concurrent use, large volumes of data, offline analytics) as NoSQL datastores. Scaling relational databases requires that users manually shard their datasets. NoSQL datastores, on the other hand, have the capability to distribute large amounts of data automatically, but lack extensive query capabilities of relational databases. NoSQL datastores as a result rely on data processing frameworks

such as MapReduce, and higher level languages such as Hive or Pig to generate to MapReduce jobs. These MapReduce jobs are appropriate for offline analytics, but their high latency make them ill-suited for real-time processing.

Each of the BigTable clones mentioned before do have a limited query language which allows access to a dataset. Yet, each language is different for each datastore, and each provides different feature sets. Developers who use one datastore must rewrite their datastore application code if they decide to switch to a different datastore for performance or feature reasons – even when the underlying functionality is similar (but different APIs are exported). Such a lack of easy portability precludes direct comparisons between datastores using real application workloads.

To address this limitation (lack of portability), and the thesis question on how we can provide additional functionality for cloud developers, we investigate the use of a unifying datastore API and programming model which allows for the simplified use and extensibility of any datastore that plugs into the open source AppScale cloud platform [22]. In particular, we target the design and implementation of a portability layer that provides generic indexing and query support across a wide range of disparate NoSQL datastore offerings (any key/value store with range query support and atomic row updates). We employ the Google App Engine (GAE) datastore API and query language (GQL) as the mechanisms with which developers access the datastores. By adopting the GAE datastore API, not only are we able to run the over 1 million applica-

tions [36] that exist today, but we also gain access to the analytic and datastore libraries written for the GAE environment [24].

Our goal is to enhance the programmability of a varying number of NoSQL datastores by providing a universal interface that expands existing query capabilities and removed the need to reimplement these features (i.e., secondary indexes) by the application developer. Moreover, we simplify the mapping of application datastore models to multiple datastores, allowing for application portability, code reuseability, and feature extendability, without the requirement that the application developer be an expert in any given datastore.

We use our systems to compare three BigTable clones: HBase, Hypertable, and Cassandra. We use a GAE benchmarking application and show the performance characteristics of each datastore. Our experimentation reveals an interesting degradation impact of soft deletions on query performance. Our experiments also consider the proprietary datastore implemented by the GAE public cloud.

In the sections that follow, we first discuss background information and related work, followed by the design and implementation of the system. We give an evaluation in Section 5.4, and conclude in Section 5.5.

5.1 Background

Google App Engine (GAE), a PaaS for web applications written in the Python, Java, and Go programming languages, can autoscale applications from the load balancer, application server, and datastore layers, removing the burdens of technically challenging aspects of application hosting from the developer. To enable virtual unlimited scale for applications, some restrictions are enforced by the GAE runtime. Front-end request can only last for 60 seconds. Any request that must run for some indefinite amount of time must use a background process. For Python applications, only approved libraries or libraries which are pure-Python are allowed, and for Java only libraries which are white listed are allowed. Moreover, no file access is allowed and data should be persisted via the datastore API.

This chapter focuses on the datastore backend API which is powered by BigTable and Megastore, two technologies internal to Google [17, 6]. Applications which use App Engine are able to scale their data as needed, and are monetarily charged based on the total volume of data stored and amount of times accessed. GAE provides applications with a 99.95% uptime SLA, powered by their high replication datastore [37].

5.1.1 Google Query Language

The datastore API for GAE provides a key/value interface for storing entities (objects), along with a SQL-like query language called GQL. It lacks the full-SQL standard, missing operations such as JOIN, MERGE, and queries which modify the datastore. However, it has useful semantics for retrieving entities based on properties/attributes of entities.

There are four different types of queries possible: kindless, ancestor, single property, and composite. The kindless query allows for the retrieval of entities across entity kinds (a kind is a table coming from the relational database model). Ancestor queries are based on entity relationships, where parent entities are assigned to child entities, and the ancestor is the root entity which has no parent of its own. Single property queries allow for the comparison between a property value using the following operators: ==, !=, <, <=, >, >=. Composite queries allow for the same but with a combination of properties, with limitations such that only one property may use an inequality filter.

The following is an example of a composite query:

```
SELECT * FROM KIND WHERE
         PROPERTY1 >= VALUE1
         AND PROPERTY2 == VALUE2
         ORDER BY PROPERTY1 DESC
```

Here we set the inequality filter for the first property and an equality filter for the second. The descending ordering has the restriction that it must be done on the property which had the inequality filter.

5.1.2 AppScale

In 2009, soon after the release of GAE, AppScale, the open source implementation of GAE, was released. AppScale is a private cloud offering with GAE API compatibility (Table **??**) and able to run on a variety of Infrastructure-as-a-Service (IaaS) layers such as AWS, Eucalyptus, and OpenStack. This chapter takes AppScale and expands its query support enabling the GAE datastore API with datastore-agnostic secondary index support.

5.1.3 Related Work

The previous implementation of AppScale's datastore API support was lacking in that all queries required pulling all the data for a given kind (all data from a table), and applying the filters in memory. This has two adverse affects: queries became slower as more data was stored in a given table, and second, once the table size was too big a query could potentially bring down a node due to a lack of memory. This chapter addresses these issues by translating GQL queries directly to the datastore. Moreover,

we add kindless query support which was previously lacking due to the design and implementation from previous work [11, 20].

Previous work which inspired the current design and implementation comes from Megastore and BigTable, which currently power GAE. Our primary distinction from the GAE implementation is that we are providing the API and GQL support across different NoSQL technologies, with the capability to run on a multitude of IaaS layers, whereas GAE is tied only to their closed source implementation and only runs on Google infrastructure. We take Google's lead however in that they spurred on the NoSQL movement with BigTable, and have given an excellent query language to emulate on top of such technologies.

In [62], the authors present how the unification of databases came together with the standardization of SQL, and present the dual of SQL for NoSQL datastores they call coSQL. In 2011, Couchbase and SQLite announced a standardization of a NoSQL query language called UnQL [87], but the project has since become defunct.

YCSB allows for the comparison of different NoSQL technologies and we are similar in that regard, yet we allow so with real GAE applications and we contrast in that we provide expanded query support on top of these technologies to enable ease of portability and programming.

5.2 Design and Implementation

To support GQL within AppScale, we must provide four different types of queries: ancestor, kindless, single property, and composite. Queries are supplied by an application and checked at runtime for validity. Each query is converted to a set of filters and order operators by the application server handling the requested query.

5.2.1 Filters, Orders, and Cusors

A filter is a data structure with fields property name, property value, and operator kind (i.e., greater than). Each filter only applies to a single property and contains a single operator, and although a query can be comprised of multiple filters as in the case for a composite query, only one property can have an inequality filter.

An order, either ascending or descending, gives the direction entities should be sorted. A given query can have multiple orders allowing for sorting on multiple properties. In the case of multiple orders, the property which has an inequality filter must come first.

A cursor can be supplied with any query except for any that use the ”!=” filter, as that does not map directly to a single range query, but rather two separate range queries. A cursor tells the query engine what the starting key should be, which typically means the previously results have already been seen and need to be skipped.

5.2.2 Query System

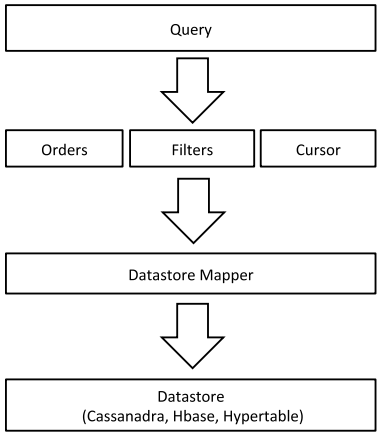


Figure 5.1: Top level design of the query system.

Figure 5.1 shows the top level design of the query system. The highest level is the query supplied by the application. The query can come in two forms, either as a GQL string, or as user created filters and order operators. If it is a GQL string, it will be converted to a set of filters, orders, and if given, a cursor. These items are then passed onto a datastore mapper which translates each query to the AppScale DB API.

5.2.3 AppScale DB API

The AppScale DB API is an abstraction from any datastore which implements the interface. The API consists of the following:

```
create_table(table, columns)
```

```
delete_table(table)

get_batch(table, key_list, columns)

put_batch(table, key_list, cell_values,

          columns)

delete_batch(table, key_list)

range_query(table,

            start_key,

            end_key,

            limit,

            offset,

            start_inclusive,

            end_inclusive)
```

Gets return a dictionary of mappings of keys to column names and values. Range queries return an ordered list of mappings of the same. Gets, puts, and deletes can all be done in batches to minimize multiple trips (something the previous version of AppScale did not have).

Range queries take a start key and an end key. If a start key is not supplied then the query starts at the very top of the table. Likewise, if an end key is not supplied then it will scan until the end of the table. Both keys have flags which tell if the start and end keys should be inclusive in the scan. A limit can be supplied to ensure that only that

prescribe amount is returned. The runtime of a query with grow O(n) in runtime if that subset of keys grows and is left unchecked via the *limit* argument. Lastly, an offset can be supplied to jump ahead a set amount of entities from a datastore range query result.

We have two restrictions as to which datastore can implement this API:

- Batch capabilities for gets, puts, and deletes
- Range queries between a start and end key

Given these two requirements, a datastore can be ported to have support for our system. For this chapter we have implemented Cassandra, Hypertable, and HBase. However, NoSQL datastores which lack these features can still be ported with performance degradation due to emulation of said feature.

5.2.4 Automatic Deployment

Each datastore is automatically configured and deployed. This is done on a variable number of machines, and does not require the user any knowledge of the datastore they've chosen to launch upon initiation of AppScale. AppScale will take template configuration files, and fill them in with the correct parameters based on the infrastructure being used. It will then launch the processes with the correct arguments and temporal procedure (i.e., master process before slave process).

5.2.5 Table and Key Layout

The AppScale design uses a set of tables which are shared between applications. Isolation is provided by the system between applications with each key prepended with the application ID and any particular namespace that application may be using. The tables are as follows:

```
ENTITY(value)
ASCENDING_PROPERTY(reference)
DESCENDING_PROPERTY(reference)
KIND(reference)
IDS(sequence)
```

The ENTITY table stores the entities in a serialized format. The key to an entity includes the application ID, namespace, and full path, where the full path of an entity is comprised of an entity's ancestors. The ASCENDING_PROPERTY and DESCENDING_PROPERTY table store references to the ENTITY table. The keys to the property tables include the application ID, namespace, property name, property value, and the full path. The property value in the descending table is lexicographically flipped to accommodate reverse ordering. Each entry also requires the full path because it is possible to have multiple entities which have the same value, hence this provides a unique

key. The KIND table also has a reference to the ENTITY table, but the path is reversed giving the child first and root last.

Ascending Property Table

Key	Reference
app_id/ns/Greeting/content/hello/Greeting:1	app_id/ns/Greeting:1
app_id/ns/Greeting/content/hi/Greeting:2	app_id/ns/Greeting:2
app_id/ns/Greeting/content/howdy/Greeting:3	app_id/ns/Greeting:3
app_id/ns/Greeting/date/06-12-2012/Greeting:1	app_id/ns/Greeting:1
app_id/ns/Greeting/date/06-13-2012/Greeting:2	app_id/ns/Greeting:2
app_id/ns/Greeting/date/06-14-2012/Greeting:3	app_id/ns/Greeting:3

Entity Table

Key	Value
app_id/ns/Greeting:1	encoded entity
app_id/ns/Greeting:2	encoded entity
app_id/ns/Greeting:3	encoded entity

Figure 5.2: Ascending property table and entity tables for a Greeting kind.

5.2.6 ID Allocation

Entities can have either a name or an ID. If a name or an ID is not assigned during creation, then a unique ID is assigned by the system. Unique IDs are attained from an ID table, where each row is per application. Each datastore server attains a block of IDs which are assigned first-come first-serve. IDs are not guaranteed to be in order due to the distributed allocation of each datastore server.

5.2.7 Put and Deletes

Each Put operation requires updating indexes into the kind and property tables. Take for example the following entity class:

```
class Greeting(Model.db):
  date = db.DateTimeProperty()
  content = db.StringProperty()
```

Any updates requires two separate indexes for a Greeting kind: data and content. The following steps are required for the successful insertion of update of an entity:

- Get the previous entity from the entity table if it exists
- Delete all the indexes from the ascending and descending table
- Insert the new indexes into the ascending and descending table
- Insert a reference to the entity table into the kind table
- Insert the entity into the entity table

5.2.8 Ancestor and Kindless Queries

Ancestor queries require a range query over the entity table. Take for example the following keys to a set of entities:

```
app_id/ns/Parent:Bill!

app_id/ns/Parent:Bill!Child:Alice!

app_id/ns/Parent:Bill!Child:Jim!

app_id/ns/Parent:Sally!

app_id/ns/Parent:Sally!Child:Alice!

app_id/ns/Parent:Sally!Child:Jim!

app_id/ns/Parent:Zack!

app_id/ns/Parent:Zack!Child:Dave!

app_id/ns/Parent:Zack!Child:Chris!
```

Here if we want to get all entities which have the root entity of Bill we will have a start key and end key of

```
start: app_id/ns/Parent:Bill!

end: app_id/ns/Parent:Bill!<tstr>
```

where the *tstr* is a series of the ASCII 255 character. Here we would get 3 entities returned with this range query which are the first three listed entities, Bill, Alice, and Jim. It should be noted that entities can have the same name or ID so long as it has a different ancestry, as shown with the children of Sally who also has children named Alice and Jim. Moreover, an ancestry can go deeper with multiple hierarchies, and they can be of any kind, whether they be the same or different.

Kindless queries operate on the entity table as well taking a key from which to start the range query from, and optionally take an ancestor to determine the end key. Without an ancestor, the query will span across different kinds of an application and a particular namespace.

5.2.9 Single Property Queries

Single property queries do a range query on either the ascending or descending property table, where the direction is given by an ordering in the query (default is ascending). The filter dictates which property is scanned and which operator is used. For an equality filter we use the following start and end key:

```
start: app_id/ns/kind/prop/val/
end: app_id/ns/kind/prop/val/<tstr>
```

For greater than:

```
start: app_id/ns/kind/prop/val/<tstr>
end: app_id/ns/Kind/prop/<tstr>
```

For greater than or equal to:

```
start: app_id/ns/kind/prop/val/
end: app_id/ns/kind/prop/<tstr>
```

For less than:

```
start: app_id/ns/kind/prop/

end: app_id/ns/kind/pop/val/
```

For less than or equal to:

```
start: app_id/ns/kind/prop/

end: app_id/ns/kind/prop/val/<tstr>
```

A not equal to query will run two queries–first a less than query, and then a greater than query.

Each of these scans will return references to the entity table. These references are used in a batch get and returned in the order requested in the query.

5.2.10 Composite Queries

A composite query consists of having two or more filters, and one or more orderings. There may only be one property for which inequality filters (<, <=, >, >=) may be used and other filters on different properties may only use the equality operator.

Composite queries use the ascending and descending tables. The filter with the inequality is applied first within a set window size, and the other filters are applied in memory. The filter will be applied until the limit amount is reached.

Multiple orderings are applied in memory after the property which had the inequality filter. An example of multiple orderings is where the first property is last name, and then first name, followed by birth date.

5.3 Evaluation

We evaluate our query system against three different datastores: Cassandra (1.0.7), Hypertable (0.9.5.5), and HBase (0.90.4-cdh3u3). A 12 node deployment on Eucalyptus 3.0 is deployed with VMs consisting of 14GBs of RAM and 4 CPU cores. The deployment has 6 nodes dedicated to web servers and six dedicated to being database nodes. Four different queries are evaluated for 100K entities: kindless, ancestor, single property, and composite. Moreover, we evaluate a mixed workload of puts and gets workloads and well as analyze the time taken for larger put batch operations. The same benchmark application is used on GAE with a higher workload for comparison purposes. We use an optimization in our implementation where we do not do references to the entity, but rather store the entity itself in our secondary indexes. This removed the overhead of doing batch gets which can add significant overhead. The tradeoff is the additional replication of the entity per index.

Upon creation of the dataset, a set of parent entities are created with each parent then having 100 child entities. Each entity has two integer properties. The first one ranges between 1 and 1,000,000 while the second ranges between 1 and 10.

Ancestor queries pick a random parent entity to query on, limiting the number to 100 (the amount of children each parent has). Kindless queries pick a parent as the last key to start the query from. Single property queries select entities which have the first property value greater than some random number between 1 and 1,000,000. Finally, composite queries do the same as single property queries, but also require the second property to be equal to some random value between 1 and 10. All queries have a limit of 100.

Our benchmark driver uses Multi-Mechanize, a Python based performance testing framework. The test lasts for 300 seconds and has users ramp up from 0 to 1500 within that time frame. The tool reports latency per request as well as throughput in 10 second intervals.

We also look at gets/puts with a read to write ratio of 80:20. For this experiment, we create 1000 entities in the datastore. If the request probability dictates that it is a read than one of the 1000 entities are chosen at random. If the request is write, than a new entity is created in the datastore.

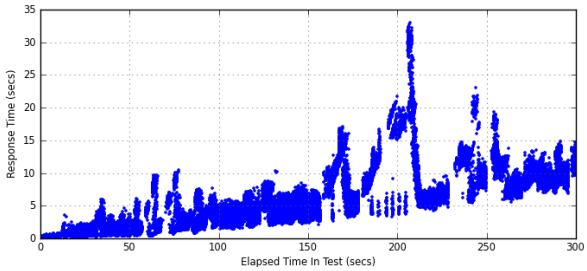


Figure 5.3: Cassandra ancestor query response time.

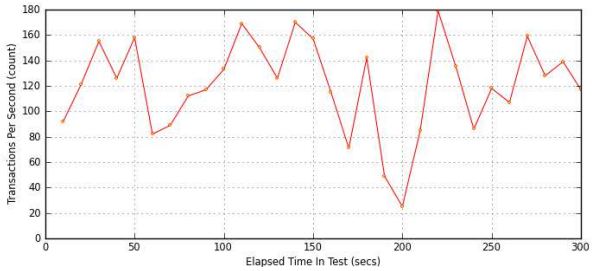


Figure 5.4: Cassandra ancestor query throughput.

5.3.1 Results

We first look at queries for Cassandra. Figure 5.20 shows ancestor queries response time. The response time stays below five seconds for the first 140 seconds as traffic is being increased. We find that there is a drop in throughput (Figure 5.21) after the 140 second mark. The reason for this drop is connection timeouts and retries by the cassandra interface. As more and more connections are used there is instability.

Kindless queries had a similar graph to ancestor queries, and therefore not shown. Single queries (Figure 5.24 and 5.25 show interesting behavior in that it achieves very

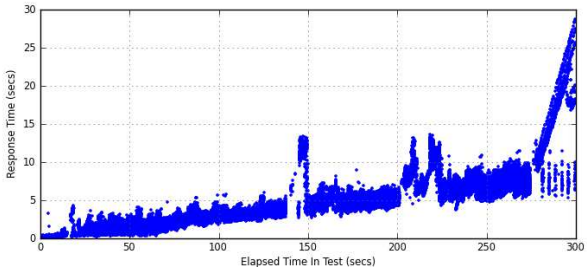


Figure 5.5: Cassandra kindless query response time.

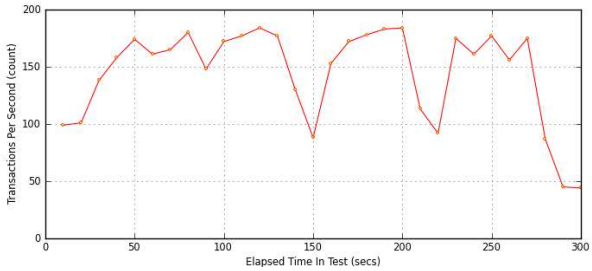


Figure 5.6: Cassandra kindless query throughput.

high throughput initially, but as users increase there is temporal spacing between requests due to connection timeouts.

Composite queries do much more work outside of the datastore because only one property is queried on within the range scan. Hence, much of the latency comes from filtering entities within memory. There is generally high latency here and under high load we hit the 60 second cutoff set forth by the front end load balancer.

Ancestor queries for HBase are much more stable compared to Cassandra. HBase has similar results for ancestor and single queries, although single property indexes did not show the brief lapses of time where no responses came through as shown in Fig-

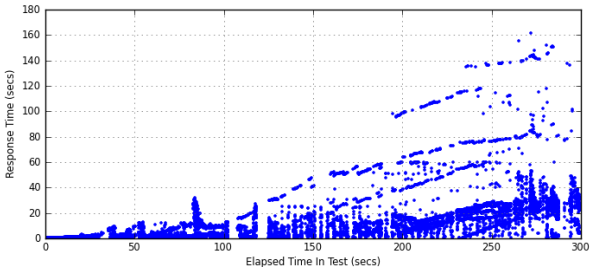


Figure 5.7: Cassandra single query response time.

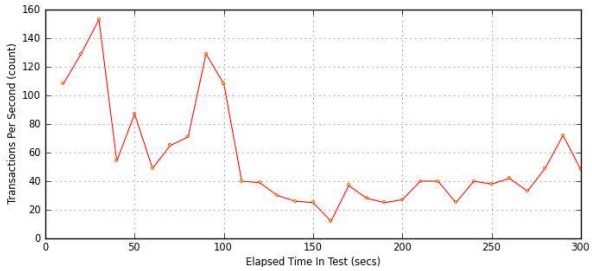


Figure 5.8: Cassandra single query throughput.

ure 5.28. Kindless queries for HBase get the best performance with throughput fluctuating around 100 request per second (Figure 5.31). Figure 5.30 shows the individual request response time and have peculiar behavior with three modes of responsiveness, less than 5 second, between 5-10, and a steady increase towards 25 seconds, all while maintaining approximately 100 requests per second.

Hypertable gets the best performance and stability in queries with over 150 request per second in ancestor queries as seen in Figures 5.32 and 5.33. As more load is added to the system, latency per request increases and does so in a steady yet undulating manner. Single property queries see a wider spread of response times, and achieves

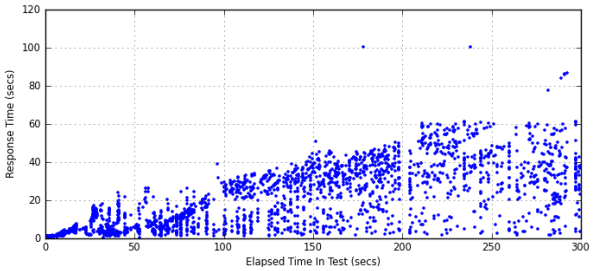


Figure 5.9: Cassandra composite query response time.

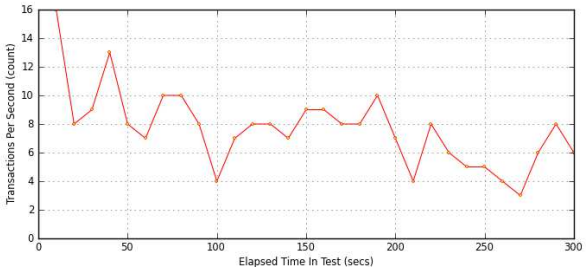


Figure 5.10: Cassandra composite query throughput.

between 75 to 95 request per second (Figures 5.34 and 5.35). Kindless queries saw similar performance to single property queries.

Lastly, we look at a mixture of reads to writes, as our query support does modify our write procedure with the requirement of updating indexes. Figure 5.36 has an experiment where we ramp up users to our experimental AppScale deployment. We can achieve high throughput with over 850 request per second (Figure not shown), and then drops towards zero. Response times show that request were being retried in three second intervals, which is the standard for TCP backoff. Our head node is being overwhelmed with traffic and forces our load generating client to backoff at the transport

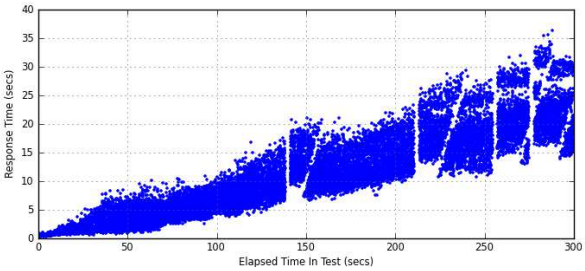


Figure 5.11: HBase ancestor query response time.

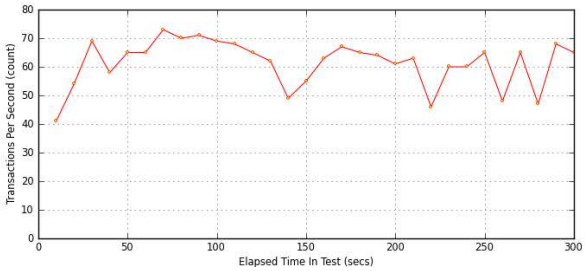


Figure 5.12: HBase ancestor query throughput.

layer. Throughput increases as the underlying TCP layer no longer is throttling traffic, yet the same pattern repeats. We see a similar result with Cassandra an HBase. Future work includes methods to alleviate this issue within AppScale.

5.3.2 Discussion

There are certain differences in the API which can lead to additional overhead when shoehorning to our common datastore API. Hypertable has the limitation of truncating strings that have the null terminating character, and therefore require encoding and decoding of the key. HBase does not include the last key in a range query and requires

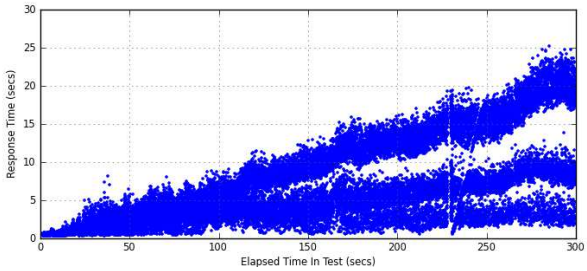


Figure 5.13: HBase kindless query response time.

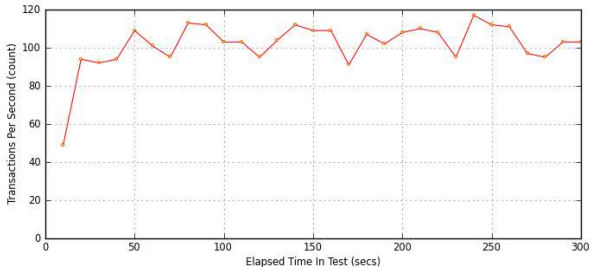


Figure 5.14: HBase kindless query throughput.

additional overhead of fetching the key if the flag for inclusivity is enabled. This overhead is more if the key does not exist, as non-existent keys are much more expensive than existing keys.

5.4 Evaluation

We evaluate our query system against three different datastores: Cassandra (1.0.7), Hypertable (0.9.5.5), and HBase (0.90.4-cdh3u3). A 12 node deployment on Eucalyptus 3.0 is deployed with VMs consisting of 14GBs of RAM and 4 CPU cores. The deployment has 6 nodes dedicated to web servers and six dedicated to being database

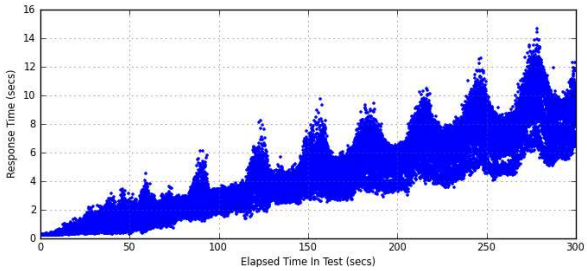


Figure 5.15: Hypertable ancestor query response time.

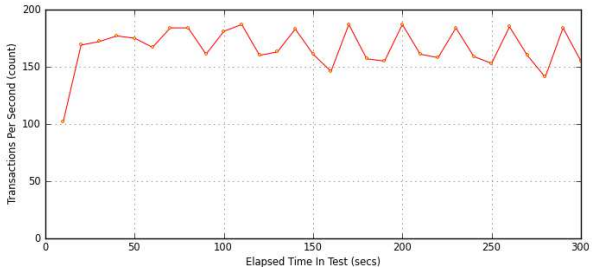


Figure 5.16: Hypertable ancestor query throughput.

nodes. Four different queries are evaluated for 100K entities: kindless, ancestor, single property, and composite. Moreover, we evaluate a mixed workload of puts and gets workloads and well as analyze the time taken for larger put batch operations. The same benchmark application is used on GAE with a higher workload for comparison purposes. We use an optimization in our implementation where we do not do references to the entity, but rather store the entity itself in our secondary indexes. This removed the overhead of doing batch gets which can add significant overhead. The tradeoff is the additional replication of the entity per index.

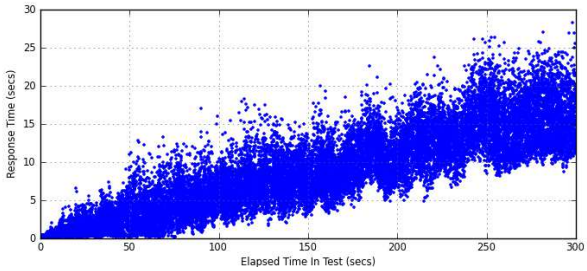


Figure 5.17: Hypertable single query response time.

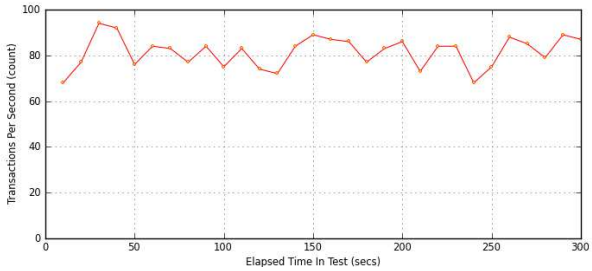


Figure 5.18: Hypertable single query throughput.

Upon creation of the dataset, a set of parent entities are created with each parent then having 100 child entities. Each entity has two integer properties. The first one ranges between 1 and 1,000,000 while the second ranges between 1 and 10.

Ancestor queries pick a random parent entity to query on, limiting the number to 100 (the amount of children each parent has). Kindless queries pick a parent as the last key to start the query from. Single property queries select entities which have the first property value greater than some random number between 1 and 1,000,000. Finally, composite queries do the same as single property queries, but also require the second

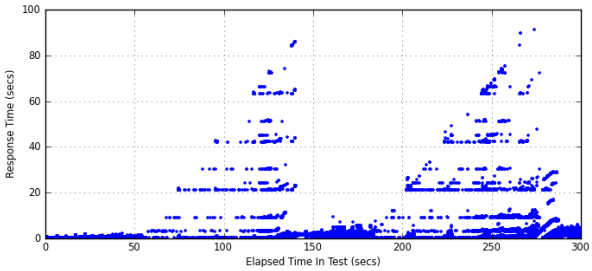


Figure 5.19: Hypertable 80:20 read to write ratio response times.

property to be equal to some random value between 1 and 10. All queries have a limit of 100.

Our benchmark driver uses Multi-Mechanize, a Python based performance testing framework. The test lasts for 300 seconds and has users ramp up from 0 to 1500 within that time frame. The tool reports latency per request as well as throughput in 10 second intervals.

We also look at gets/puts with a read to write ratio of 80:20. For this experiment, we create 1000 entities in the datastore. If the request probability dictates that it is a read than one of the 1000 entities are chosen at random. If the request is write, than a new entity is created in the datastore.

5.4.1 Results

We first look at queries for Cassandra. Figure 5.20 shows ancestor queries response time. The response time stays below five seconds for the first 140 seconds as traffic

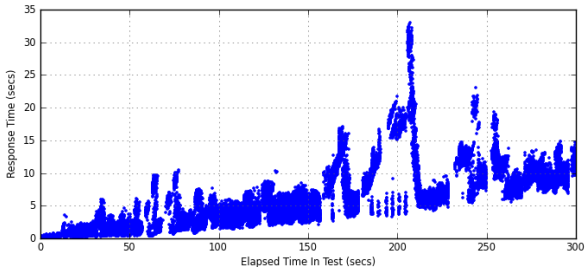


Figure 5.20: Cassandra ancestor query response time.

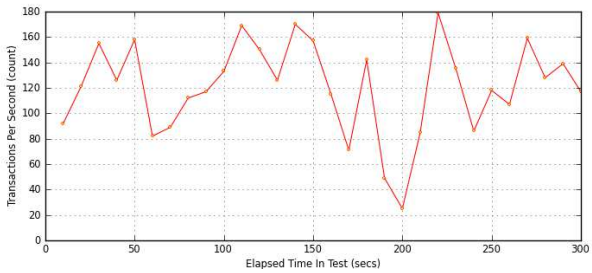


Figure 5.21: Cassandra ancestor query throughput.

is being increased. We find that there is a drop in throughput (Figure 5.21) after the 140 second mark. The reason for this drop is connection timeouts and retries by the cassandra interface. As more and more connections are used there is instability.

Kindless queries had a similar graph to ancestor queries, and therefore not shown. Single queries (Figure 5.24 and 5.25 show interesting behavior in that it achieves very high throughput initially, but as users increase there is temporal spacing between requests due to connection timeouts.

Composite queries do much more work outside of the datastore because only one property is queried on within the range scan. Hence, much of the latency comes from

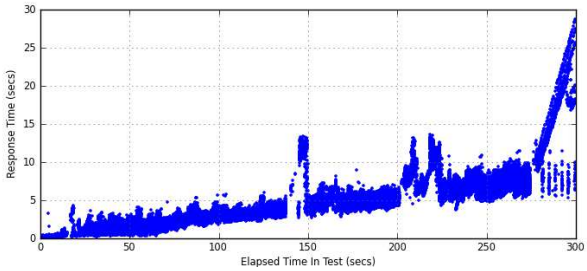


Figure 5.22: Cassandra kindless query response time.

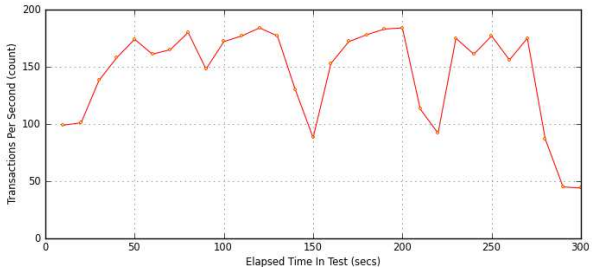


Figure 5.23: Cassandra kindless query throughput.

filtering entities within memory. There is generally high latency here and under high load we hit the 60 second cutoff set forth by the front end load balancer.

Ancestor queries for HBase are much more stable compared to Cassandra. HBase has similar results for ancestor and single queries, although single property indexes did not show the brief lapses of time where no responses came through as shown in Figure 5.28. Kindless queries for HBase get the best performance with throughput fluctuating around 100 request per second (Figure 5.31). Figure 5.30 shows the individual request response time and have peculiar behavior with three modes of responsiveness,

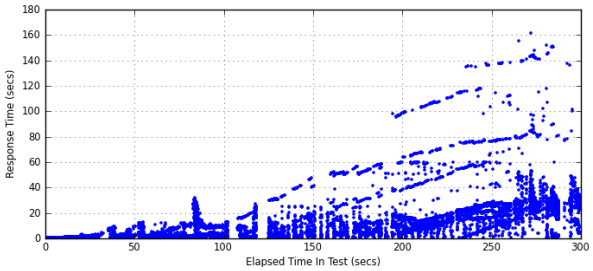


Figure 5.24: Cassandra single query response time.

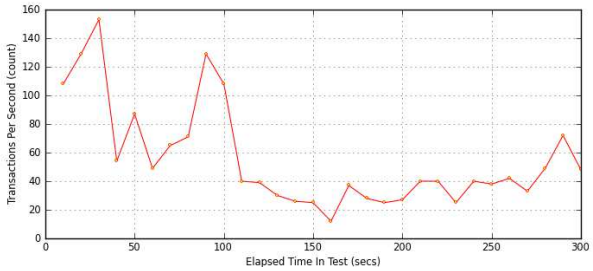


Figure 5.25: Cassandra single query throughput.

less than 5 second, between 5-10, and a steady increase towards 25 seconds, all while maintaining approximately 100 requests per second.

Hypertable gets the best performance and stability in queries with over 150 request per second in ancestor queries as seen in Figures 5.32 and 5.33. As more load is added to the system, latency per request increases and does so in a steady yet undulating manner. Single property queries see a wider spread of response times, and achieves between 75 to 95 request per second (Figures 5.34 and 5.35). Kindless queries saw similar performance to single property queries.

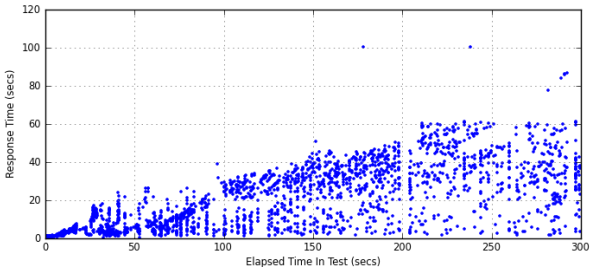


Figure 5.26: Cassandra composite query response time.

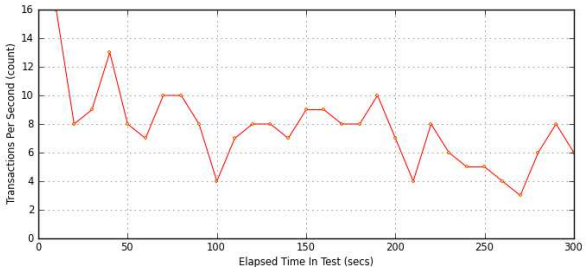


Figure 5.27: Cassandra composite query throughput.

Lastly, we look at a mixture of reads to writes, as our query support does modify our write procedure with the requirement of updating indexes. Figure 5.36 has an experiment where we ramp up users to our experimental AppScale deployment. We can achieve high throughput with over 850 request per second (Figure not shown), and then drops towards zero. Response times show that request were being retried in three second intervals, which is the standard for TCP backoff. Our head node is being overwhelmed with traffic and forces our load generating client to backoff at the transport layer. Throughput increases as the underlying TCP layer no longer is throttling traffic,

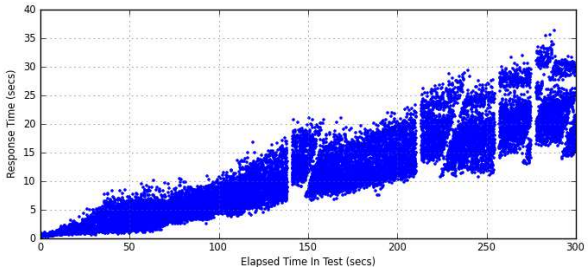


Figure 5.28: HBase ancestor query response time.

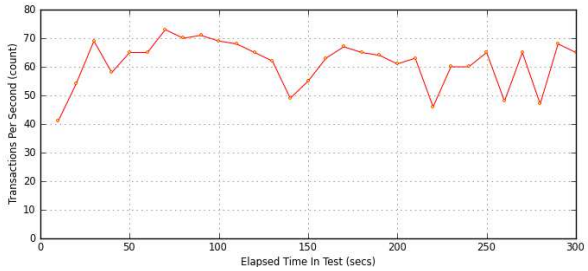


Figure 5.29: HBase ancestor query throughput.

yet the same pattern repeats. We see a similar result with Cassandra an HBase. Future work includes methods to alleviate this issue within AppScale.

5.4.2 Discussion

There are certain differences in the API which can lead to additional overhead when shoehorning to our common datastore API. Hypertable has the limitation of truncating strings that have the null terminating character, and therefore require encoding and decoding of the key. HBase does not include the last key in a range query and requires additional overhead of fetching the key if the flag for inclusivity is enabled. This over-

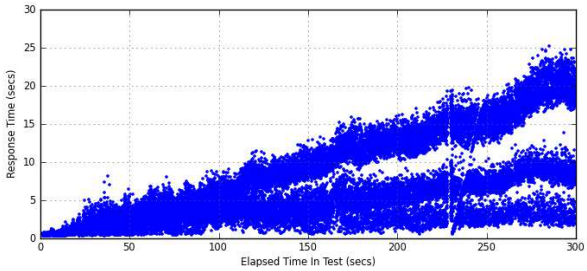


Figure 5.30: HBase kindless query response time.

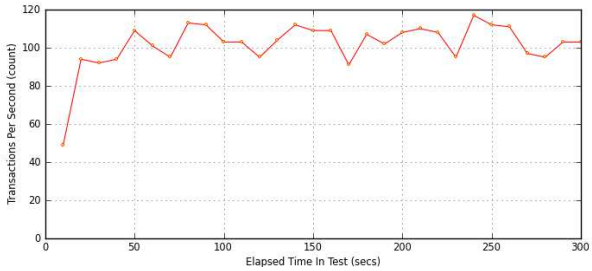


Figure 5.31: HBase kindless query throughput.

head is more if the key does not exist, as non-existent keys are much more expensive than existing keys.

5.5 Summary

In this chapter we have presented a design, implementation, and evaluation of a middleware that provides secondary index support. This support allows developers to use a SQL-like language, GQL, to query their large scale data in real-time. Our system supports four main types of queries: ancestor, kindless, single property and composite

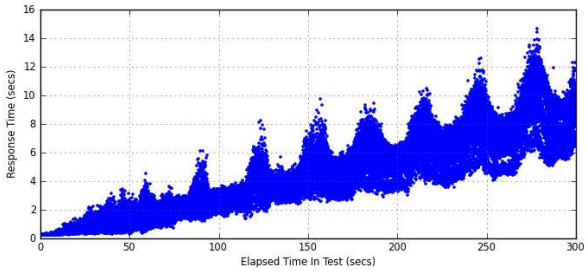


Figure 5.32: Hypertable ancestor query response time.

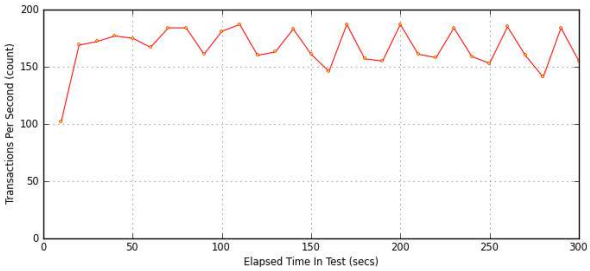


Figure 5.33: Hypertable ancestor query throughput.

queries. We provide these queries on NoSQL datastores in a datastore-agnostic way, in which we only require range query support by the underlying datastore.

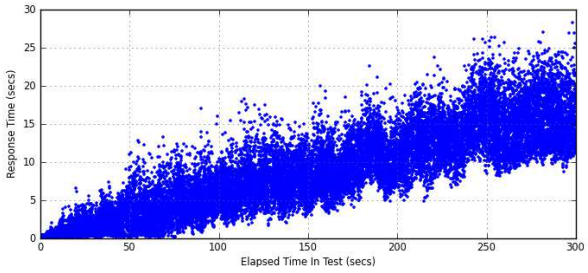


Figure 5.34: Hypertable single query response time.

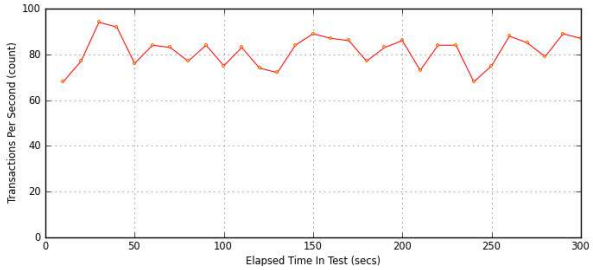


Figure 5.35: Hypertable single query throughput.

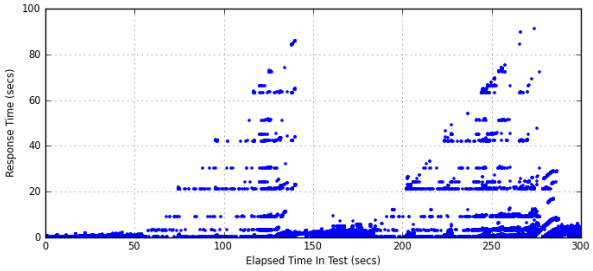


Figure 5.36: Hypertable 80:20 read to write ratio response times.

Chapter 6

Hybrid Cloud Support for Large Scale Analytics and Web Processing

Platform-as-a-service (PaaS) offerings, such as Microsoft Azure [4] and Google App Engine [37], automate configuration, deployment, monitoring, and elasticity by abstracting away the infrastructure through well-defined APIs and a higher-level programming model. PaaS providers restrict the behavior and operations (libraries, functionality, and quota-limit execution) of hosted applications, both to simplify cloud application deployment, and to facilitate scalable use of the platform by very large numbers of concurrent users and applications. Google App Engine (GAE), the system we focus on herein, currently supports over 7.5 billion page views per day across over 1,00,000 active applications [36] as a result of their platform's design. As is the case for public IaaS systems, public PaaS users pay only for the resources and services they use.

A key functionality lacking from the original design of PaaS systems is online analytics processing (OLAP). OLAP enables application developers to model, analyze, and identify patterns in their online web applications as users access them. Such analysis helps developers target specific user behavior with software enhancements (code/data optimization, improved user interfaces, bug fixes, etc.) as well as applying said analysis for commercial purposes (e.g. marketing and advertising). These improvements and adaptations are crucial to building a customer base, facilitating application longevity, and ultimately commercial success for a wide range of companies. In recognition of this need, PaaS systems are increasingly offering new services that facilitate OLAP execution models by and for applications that execute over them [38, 77, 5]. However, such support is still in its infancy and is limited in flexibility, posing questions as to what can be done within quota limits and how the service connects with the online applications they analyze.

In this chapter, we investigate the emerging support of OLAP for GAE, identify its limitations, and its impact on the cost and performance of applications in this setting. We propose an alternate approach to OLAP, in the form of a hybrid cloud consisting of a public cloud executing the live web application or service and a remote analytics cloud which shares application data. We build upon and extend AppScale to enable analytic processing for web developers and address the portion of the thesis question which concerns helping developers do so for large data.

Portability gives developers the freedom and flexibility to explore, research, and tinker with the system level details of cloud platforms [20, 22, 54]. Our hybrid OLAP solution provides multiple options for data transfer between the two clouds, facilitates deployment of the analytics cloud over Amazon's EC2 public cloud or an on-premise cluster, and integrates the popular Hive distributed data warehousing technology to enable a wide range of complex analytics applications to be performed over live GAE datasets. By using a remote AppScale cloud for analytics of live data, we are able to specialize it for this execution model and avoid the quotas and restrictions of GAE, while maintaining the ease of use and familiarity of the GAE platform.

In the sections that follow, we first provide background on GAE and AppScale. We then describe the design and implementation of our hybrid OLAP system. We follow this with an evaluation of existing solutions for analytics, our Hive processing, and an analysis of the cost and overhead of cross-cloud data synchronization. Finally, we present related work and conclude.

6.1 Background

Google App Engine was released in 2008, with the goal of allowing developers to run applications on Google's infrastructure via a fully managed and automatically scaled system. While the first release only supported the Python programming lan-

guage, the GAE team has since introduced support for the Java and Go languages. Application developers can access a variety of different services via a set of well-defined APIs. The API implementations in the GAE public cloud are optimized for scalability, shared use, and fault tolerance. The APIs that we focus on in this chapter are the Datastore (for data persistence), URL Fetch (for communication), and Task Queues (for background processing).

AppScale implements the GAE APIs using a combination of open source technologies and custom software. It provides a database-agnostic layer, which multiple disparate database/datastore technologies (e.g. Cassandra, HBase, Hypertable, MySQL cluster, and others) can plug into [11]. It implements the Task Queue API by executing a task on a background thread in the same application server as the application instance that makes the request. This support, though simple, is inherently inefficient and not scalable, because it is neither distributed nor load-balanced. Moreover, it does not share state between application servers, which leads to incorrect application behavior when more than one application server is present. We replace this API implementation as part of this work, addressing this limitation.

6.1.1 App Engine Analytics Libraries

The Task Queue API facilitates the use of multiple, independent user-defined queues, each with a rate limit of 100 tasks per second (which can be increased in some cases [37])

in GAE. A task consists of an application URL, which is called by the system upon task dequeue. A 200 HTTP response code (OK) indicates that the task completes successfully. Other HTTP codes cause re-enqueuing of the task for additional execution attempts. The number of retries, a time delay, and a task name can be optionally specified by developers as part of the task when it is enqueued. Use of task names is important to prevent the same task from being enqueued multiple times (the lack of such measures can result in a task fork bomb, in which a task is infinitely enqueued). One way to circumvent the 10 minute time limit for a task is to chain tasks, in which the initial task performs a portion of the work, and enqueuing another task to resume where it has left off. Tasks should be idempotent, or only perform side effects (e.g., updating shared, persistent data) as the final operation – since any failure of a previous statement will cause the task to be re-enqueued (potentially updating shared state incorrectly).

GAE application developers are responsible for program/task correctness when failures occur. This requires that developers make proper use of task names and chaining, and implement tasks that are idempotent. Doing so for all but the most trivial of applications can be a challenging undertaking for all but expert developers. To address this limitation, there are libraries that provide a layer of abstraction over the GAE task queue interface and implementation. These libraries are Fantasm [35], GAE Pipeline [77], and GAE MapReduce [38]. Each automates naming and failure handling by saving intermediate state via the Memcache and the Datastore APIs.

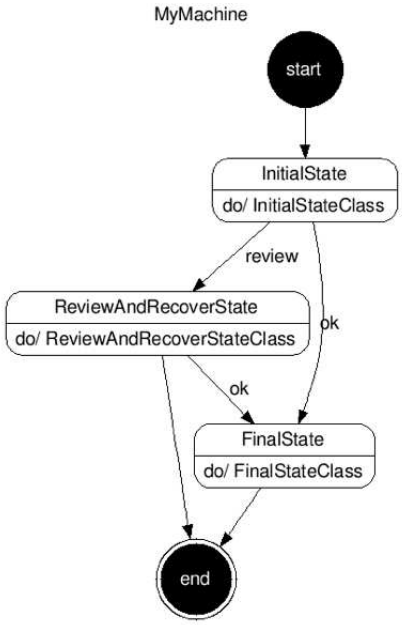


Figure 6.1: An example state machine in Fantasm.

Fantasm, based on [40], employs a programming model that is based on finite state machines (FSM). A programmer describes a state machine via the YAML markup language by identifying states, events, and actions. The initial state typically starts with a query to the datastore, to gather input data for analysis. Fantasm steps through the query and constructs a task for each entity (datastore element) that the query processes in each state. Optionally, there can be a fan-in state, which takes multiple previous states and combines them via a reduction method. Figure 6.1 shows an example FSM. A limitation of Fantasm is how it iterates through data. It does not shard datasets, but instead, pages through a query serially, leading to inefficient execution of state machines.

```
class WCUrl(pipeline.Pipeline):
  def run(self, url):
    r = urlfetch.fetch(url)
    return len(r.data.split())

class Sum(pipeline.Pipeline):
  def run(self, *values):
    return sum(values)

class MySearchEngine(pipeline.Pipeline):
  def run(self, *urls):
    results = []
    for u in urls:
      # Do word count on each URL
      results.append((yield WCUrl(u)))
    yield Sum(*results) # Barrier waits
```

Figure 6.2: Code example of Pipeline parallellizing work.

The GAE Pipeline library facilitates chaining of tasks into a workflow. Pipeline stages (tasks) yield for barrier synchronization, at which point the output is unioned and passed onto the next stage in the pipeline. Figure 6.2 shows an example of parallel processing via Pipeline that counts the number of unique words on multiple web pages. The *yield* operator spawns background tasks, whose results are combined and passed to the *Sum* operation. Implementing similar code via just the Task Queue API is possible, but is more complicated for users.

The GAE MapReduce library performs parallel processing and reductions across datasets. Mapper functions operate on a particular kind of entity and reducer functions operate on the output of mappers. Alternative input readers (e.g. for use of Blobstore files) and sharding support is also available. The GAE MapReduce library uses the Task

Queue API for its implementation, as opposed to using Google's internal MapReduce infrastructure or Hadoop, an open source implementation. Both are more flexible than GAE MapReduce, and allow for a wider range of analytics processing than this library. Currently, a key limitation of GAE MapReduce is that all entities in the Datastore are processed, even when they are not of interest to the analysis.

Each of these abstractions for background processing and data analytics in GAE introduce a new programming model with its own learning curve. Moreover, analytics processing on the dataset is intertwined with the application, (that users use to produce/access the dataset) which combines concerns, can introduce bugs, and can have adverse affects on programmer productivity, user experience, and monetary cost of public cloud use. To address these limitations, we investigate an alternate approach to performing online data analytics for applications executing within GAE that employs a combination of GAE and AppScale concurrently.

6.1.2 Related Work

OLAP and data warehousing systems have been around since the 1970s [18], yet there is no system available for GAE which is currently focused providing OLAP for executing web applications. AppScale, with its API compatibility and our extensions herein, brings OLAP capabilities (as well as its testing and debugging) to this domain.

TyphoonAE is the only other framework which is capable of running GAE applications outside of GAE. TyphoonAE however is a more efficient version of the SDK (executes the system serially) and only supports the Python language. AppScale and our work supports Python, Java, and Go languages and is distributed and scalable. TyphoonAE does not have the same facility as AppScale to run analytics, as it does not support datastores capable of Hive support. Private PaaS offerings such as Cloud Foundry [26] offer an open source alternative to many proprietary products and offer automatic deployment and scaling of applications, yet do not support GAE APIs.

There are many cloud platforms which allows for analytics to be run on large scale datasets. Amazon's Elastic MapReduce is one such service, where machines are automatically setup to run jobs, along with customized interfaces for tracking jobs [61]. The Mesos framework is another cloud platform which can run a variety of processing tools such as Hadoop and MPI, and does so with a dynamically shared set of nodes [47]. Helios is yet another framework that simplifies the application deployment process.

In [53], the authors measured data-intensive applications in multiple clouds including GAE, AWS, and Azure. Their application was a variant of the TPC-W benchmark, similar to an online bookstore. Our benchmarks, by comparison, are analytics driven rather than online processing. Furthermore, since the time of publication Google–as well as the other cloud providers–have continuously improved functionality and added

features. Our work provides a new snapshot in time of the current system, which has since come out of preview and become a fully supported service.

Data replication across datacenters is a common method for prevention of data loss and to enable disaster recovery if needed. Currently GAE implements three-plus times replication across datacenters using a variant of the Paxos algorithm [6]. Extant solutions, such as [91], however, are not applicable because of the restrictions imposed by the GAE runtime. To overcome this limitation, we provide a library wrapper around destructive datastore operations, to asynchronously update our remote AppScale analytic platform. As part of future work, we are investigating how to provide disaster recovery using our hybrid system.

6.2 Hybrid PaaS Support for Web Application Data Analysis

In this work, we investigate how to combine two PaaS systems together into a hybrid cloud platform that facilitates the simple and efficient execution of large-scale analysis of live web application data. Our hybrid model executes the web application on the GAE public cloud platform, synchronizes the data between this application/platform and a remote AppScale cloud, and facilitates analysis of the live application data using the GAE analytics libraries, as well as other popular data processing engines (e.g.

Hadoop/Hive) using AppScale. Users can deploy AppScale on a local, on-premise cluster, or over Amazon EC2. In this section, we overview the two primary components of our hybrid cloud system: the data synchronization support and the analytics processing engine. We then discuss our design decisions and how our solution works within the restrictions of the GAE platform.

6.2.1 Cross-Cloud Data Synchronization

The key to our approach to analytics of live web applications is the combined use of GAE and AppScale. Since the two cloud platforms share a common API, applications that execute on one can also do so on the other, without modification. This portability also extends to the data model. That is, given the compatibility between AppScale and GAE, we can move data between the two different platforms for the same application. We note that for vast datasets such an approach may not be feasible. However, it is feasible for a large number of GAE applications today. The cross-platform portability facilitates and simplifies our data synchronization support, and makes it easier for developers to write application and analytics code, because the runtime, APIs, and code deployment process is similar and familiar.

We consider two approaches to data synchronization: bulk and incremental data transfer. For bulk transfer, GAE currently provides tools as part of its software development kit (SDK) to upload and download data into and out of the GAE datastore en

masse. We have extended AppScale with similar functionality. Our extensions provide the necessary authentication and data ingress/egress support, as well as support for the GAE Remote API [37], which enables remote access to an application's data in the datastore. The latter must be employed by any application for which hybrid analytics will be used. Using the Remote API, a developer can specify what data can be downloaded (the default is all). Bulk download from, and upload to, is subject to GAE monetary charges for public cloud use.

There are several limitations to bulk data transfer as a mechanism for data synchronization between the two application instances. First, in its current incarnation, transfer is all or nothing (of the entities specified). As such, we are able to only perform analytics off-line or postmortem if we are to copy the dataset once (the most inexpensive approach). To perform analytics concurrently with web application execution, we are forced to download the same data repeatedly over time (as the application changes it). This can be both costly and slow. Finally, the data upload/download tools from GAE are slow and error prone, with frequent interruptions and data loss.

To address these limitations, we investigate an alternative approach to synchronizing data between GAE and AppScale: incremental data transfer. To enable this, we have developed a library for GAE applications that runs transparently in both GAE and AppScale. Our incremental data transfer library intercepts all destructive operations (writes and deletes) and communicates them to the AppScale analytics cloud. In our current

prototype, we do not support the limited form of transactions that GAE applications can perform [33]. As part of our on-going and future work, we are considering how to reflect committed transactional updates in the AppScale analytics cloud. Developers specify the location of the AppScale analytics cloud as part of their GAE application configuration file. Since the library code executes as part of the application in GAE, it must adhere to all of the GAE platform restrictions. Furthermore, communication to the AppScale analytics cloud is subject to GAE charges for public cloud use.

Our goal with this library is to avoid interruption or impact on GAE web application performance and scale, from the users' perspective. We consider two forms of synchronization with different consistency guarantees: eventual consistency (EC) and best effort (BE). EC incremental transfer uses the Task Queue API to update the AppScale analytics cloud. Using this approach, the library enqueues a background task in GAE upon each destructive datastore operation. The task then uses the URL Fetch library to synchronously transmit the updated entity. In GAE, tasks are retried until they complete without error. Thus, GAE and AppScale data replicas for the application are eventually consistent, assuming that both the GAE and AppScale platforms are available.

Our second approach, best effort (BE), for incremental transfer implements an asynchronous URL Fetch call to the AppScale analytics cloud for the application upon each destructive update. If this call fails, the GAE and AppScale replicas will be inconsistent

until the next time the same entity is updated. The BE approach can implement potentially fewer transfers since failed transfers are not retried. This may impact the cost of hybrid cloud analytics using our system. BE is useful for settings in which perfect consistency is not needed.

To maintain causal ordering across updates we employ a logical clock (a Lamport clock [58]), ensuring that only the latest value is reflected in the replicated dataset for each entity. Using this approach, it is possible that at any single point in time there may be an update missing (still in flight due to retries in EC or failed in BE) in the replicated dataset. We transmit entity updates as Protocol Buffers, the GAE transfer format of Datastore entities.

6.2.2 Analytics Processing Engine within AppScale

We next consider different implementations of the AppScale analytics processing engine. We first extend AppScale to support each of the three analytics libraries that GAE supports, described in Section 6.1.1. We start by replacing the TaskQueue API implementation in AppScale, from a simple, imbalanced approach, to a new software layer, similar to that for the Datastore API implementation and transaction support [20], that is implementation-agnostic and allows different task queue implementations to be plugged in and experimented with.

The GAE Task Queue API includes the functions:

```
AddTask(name, url, parameters)
DeleteTask(name)
PurgeQueue()
```

We emulate the GAE behavior of this API (that we infer using the GAE SDK and by observing the behavior of GAE applications) in our task queue software layer within AppScale. Each task that is added to the queue specifies a *url* that is a valid path (URL route) defined in the application, to which a POST request can be made using the *parameters*. The *name* argument ensures that a task is only enqueued once given a unique identifier. If a name is not supplied, a unique name is assigned to it. The *PurgeQueue* operation will remove all tasks from a queue, resetting it to an initial, empty state, whereas *DeleteTask* will remove a named task if it is still enqueued. Task execution code is within the application itself (a relative path), or can be a fully remote location (a full path). Successful execution of a task is indicated by a HTTP 200 response code. The task queue implementation retries failed tasks up to a configurable number of times, defaulting to ten attempts.

The AppScale Task Queue interface for plugging in new messaging systems is as follows: This API includes the functions:

```
EnqueueTask(app_name, url, parameters)
LocateTask(app_name, task_name)
AddTask(app_name, task_name)
```

```
AckTask(app_name, task_name, reenqueue)

PurgeQueue(app_name)
```

The *AddTask* function stores the given task name and state in the system-wide data-store. Possible task states are 'running', 'completed', or 'failed', and states can be retrieved via *LocateTask*). *AckTask* tells the messaging system whether the task should be re-enqueued, and if it should be, the messaging system increments the retry count associated with that task. Each function requires the application name because AppScale supports multiple applications per cloud deployment, isolating such communications.

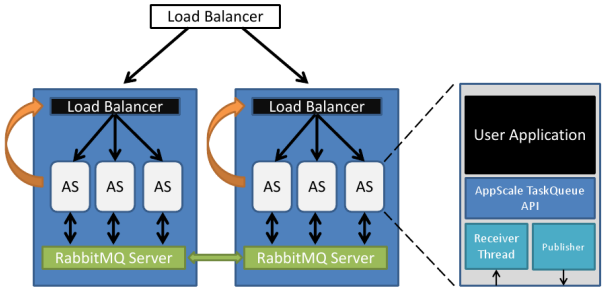


Figure 6.3: Overview of RabbitMQ implementation in AppScale.

Using the AppScale task queue software layer, we plug-in the VMWare RabbitMQ [79] technology and implement support for each of the GAE analytics libraries (GAE MapReduce, GAE Pipeline, and Fantasm) described in Section 6.1.1 on top of the Task Queue API. We have chosen to integrate RabbitMQ due to its widespread use and multiple useful features within a distributed task queue implementation, including clustering, high availability, durability, and elasticity. Figure 6.3 shows the software architecture of RabbitMQ as a task queue within AppScale (two nodes run a given application in

this figure). Each AppScale node that runs the application (load-balanced application servers) runs a RabbitMQ server. Each application server has a client that can enqueue tasks or listen for assigned tasks (a callback thread) to or from the RabbitMQ server. We store metadata about each task (name, state, etc.) in the system in the cloud datastore. A worker thread consumes tasks from the server. Upon doing so, it issues a POST request to its localhost or full path/route (if specified), which gets load-balanced across application servers running on the nodes. Tasks are distributed to workers in a round-robin basis, and are retried upon failure. RabbitMQ re-enqueues failed tasks and is fault tolerant.

In addition to the Task Queue, MapReduce, Pipeline, and Fantasm APIs, we also consider a processing engine that is popular for large-scale data analytics yet that is not available in GAE. This processing engine employs a combination of MapReduce [30] (not to be confused with GAE MapReduce, which exports different semantics and behavioral restrictions) and a query processing engine that maps SQL statements to a workflow of MapReduce operations. In this work, we employ Hadoop, an open source implementation of a fully featured MapReduce system, and Hive [85, 67, 48], an open source query processing engine, similar in spirit to Pig and Sawzall. This processing engine (Hive/Hadoop) provides users with ad-hoc data querying capabilities that are processed using Hadoop, without requiring any knowledge about how to write or chain MapReduce jobs. Moreover, using this AppScale service, users can operate on data

using the familiar syntax of SQL and perform large-scale, complex data queries using Hadoop.

AppScale integrates multiple datastore technologies, including Cassandra, Hypertable, and HBase [11, 12]. All of these datastores are distributed, scalable, fault-tolerant, and provide column-oriented storage. Each datastore provides a limited query language, with capabilities similar to the GAE Datastore access model: entities, stored as Protocol Buffers, are accessed via keys and key ranges. We focus on the currently best performing datastore in this work, Cassandra [20].

Our extensions swap out the Hadoop File System (HDFS) in AppScale and replace it with CassandraFS [10], an HDFS-compatible storage layer, that interoperates directly with Cassandra, with the added benefit of having no single points of failure within its NameNode process. Above CassandraFS, we deploy Hadoop; above Hadoop, we deploy Hive. Developers can issue Hive queries from the command line, a script issued on any AppScale DB node [54], or via their applications through a library, similar to the GAE MapReduce library implementation in AppScale.

To enable this, we modified the datastore layout of entities in the AppScale datastore. Previously, we employed a single column-family (table) for all kinds of entities in an applications dataset. We shared tables across multiple applications and we isolated datasets using namespaces prepended to the key names. In this work, we store column-families for each kind of entity. The serialization and deserialization between

Hadoop, CassandraFS, and Cassandra happens through a custom interface, which enables Hadoop mappers and reducers to read and write data from Cassandra. We extended the AppScale Datastore API with a layer that translates entities to/from Protocol Buffers. Our extensions eliminate the extract-transform-load step of query processing so that entities can be processed in place.

This support enables Hive queries to run SQL statements which are partitioned into multiple mapper and reducer phases. Hive compiles SQL statements into a series of connected map and reduce jobs. Analysts can perform queries that are automatically translated to mappers and reducers, rather than manually writing these functions and chaining them together. Take for example the task of getting the total count of entities of a certain kind. A Hive query is as simple as:

```
SELECT COUNT(*) FROM appid_kind;
```

To to the same thing in GAE, the entities are paged through and a counter incremented. Note that the Google Query Language for GAE applications limits the number of entities in a single fetch operation to 1000. If the dataset is large enough, then the developer must use a background task or manually implement task queue chaining. Another alternative approach is to use sharded counters to keep a live count; multiple counter entities are required if the increment must happen at a rate faster than once per second. Both methods are foreign to many developers and are far more complex and non-intuitive than simple SQL Hive statements.

us-east-1	Northern Virginia, USA
eu-west-1	Dublin, Ireland
ap-southeast-1	Singapore
ap-northeast-1	Tokyo, Japan
sa-east-1	Sao Paulo, Brazil
us-west-1	Oregon, USA
us-west-2	California, USA

Table 6.1: EC2 Regions for Amazon Web Services.

6.3 Evaluation

In this section, we evaluate multiple components of our hybrid web application and analytics system. We first start with an evaluation of the cross-cloud connectivity within a hybrid cloud deployment. For this, we analyze the round-trip time (RTT) between a deployed GAE application in Google datacenters and virtual machines deployed globally across multiple regions and availability zones of Amazon EC2. We next evaluate the performance of the GAE libraries for analytics using the GAE public cloud. We then evaluate the efficacy of our extensions to the AppScale TaskQueue implementation. Lastly, we show the efficiency of using the AppScale analytic solution running Hive over Cassandra.

6.3.1 Cross Cloud Data Transfer

To evaluate the performance of cross-cloud data synchronization between GAE and AppScale, we must first understand the connectivity rate between them for incremental

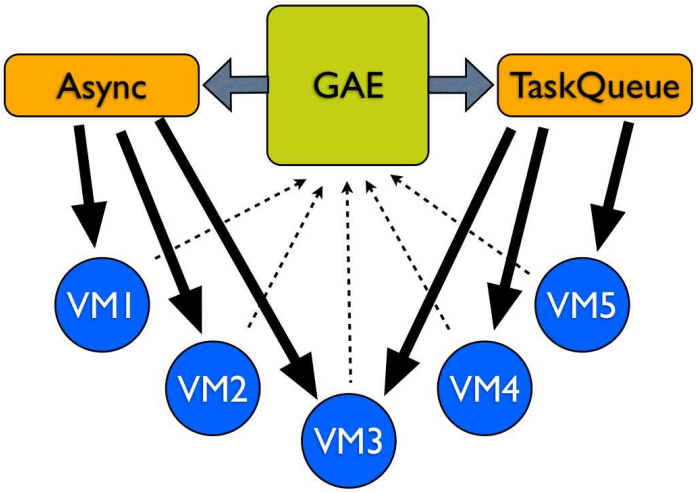


Figure 6.4: Experimental Setup for Measuring Round-trip Time and Bandwidth Between a GAE Application and VMs in Multiple EC2 Regions.

data transfer (cf Section 6.2.1). To measure this, we deploy an application in the GAE public cloud that we access remotely from multiple Amazon EC2 micro instances in 16 different availability zones, spanning seven regions. Figure 6.1 shows the regions we consider, and Figure 6.4 depicts our experimental setup.

Our experiment issues a HTTP POST request from the EC2 instances, each with a data payload of a particular size, a destination URL location, a unique identifier, and the type of hybrid data synchronization to employ: eventually consistent (EC) or best effort (BE). The sizes we consider are 1KB, 10KB, 100KB, and 1MB (the maximum allowed

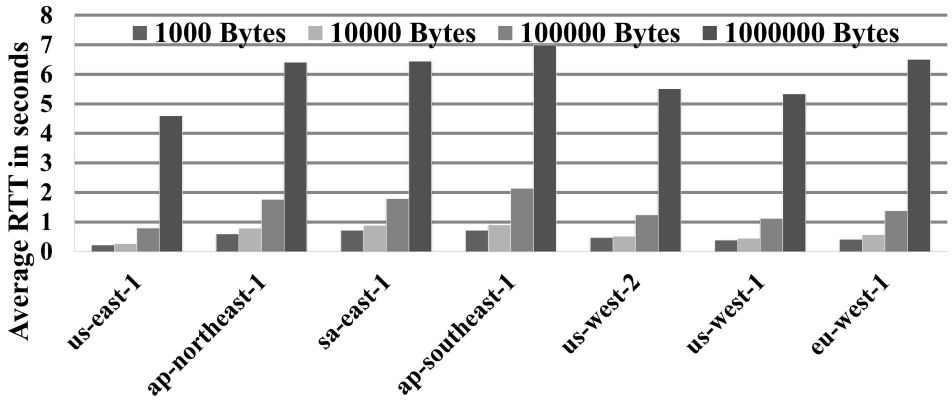


Figure 6.5: Round-trip Time Per Different Packet Size.

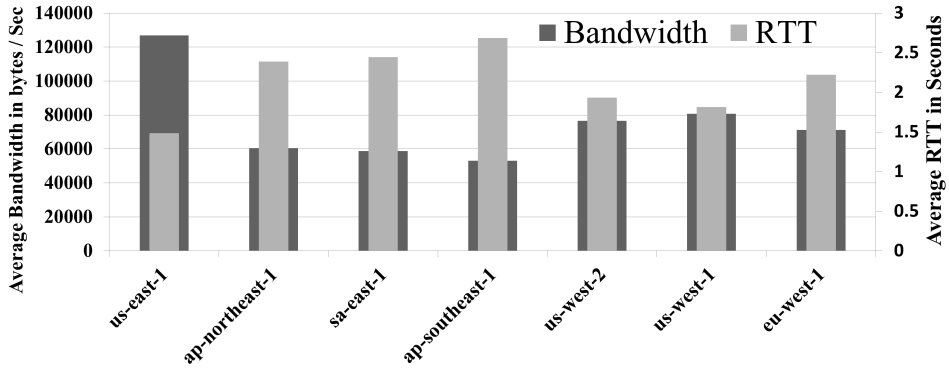


Figure 6.6: Round-trip Time and Bandwidth Between a GAE Application and Different EC2 Regions.

for GAE's Datastore API). The EC2 instances host a web server, which receives the data from the GAE application (either from a task via EC or from the application itself via BE) and records the current time and request identifier. Figure 6.5 shows the average RTT for different packet sizes, for each availability zone. The data indicates that it is advantageous to batch updates when possible since there is not a linear relationship between size and RTT, as sizes grow.

We next consider whether the geographical location of the AppScale cloud (different EC2 regions) makes a significant difference in the communication overhead on data synchronization. To evaluate this, we consider the average round-trip time (RTT) and bandwidth across payload sizes to the GAE application for the different regions (Figure 6.6). The US East region had the RTT with the highest bandwidth, by a factor of two. Both US regions have the next best performing communication behavior. This data suggests that our GAE application is hosted (geographically) in GAE in the Eastern US. Locality to the application shows more than 2x the bandwidth for the US East availability zone than other zones (130KB versus 50KB to 80KB for other zones). We investigated this further and found via traceroutes and pings that the application was located near or around New York. We also found with this experiment that bandwidth over time is generally steady, with the exception of between the hours of 16:00 and 22:00 (figure not shown). It may be possible to take advantage of such information to place the AppScale cloud to enable more efficient data synchronization.

We next investigated the task queue delay in GAE. We are interested in whether the delay changes over time or remains relatively consistent. We present this data in Figure 6.7, as points at each hour in the day (normalized to Eastern Standard Time) that we connect using lines to help visualize the trends. The left x-axis is RTT in seconds for the region, and the right x-axis is the average queue delay (in seconds) for the region.

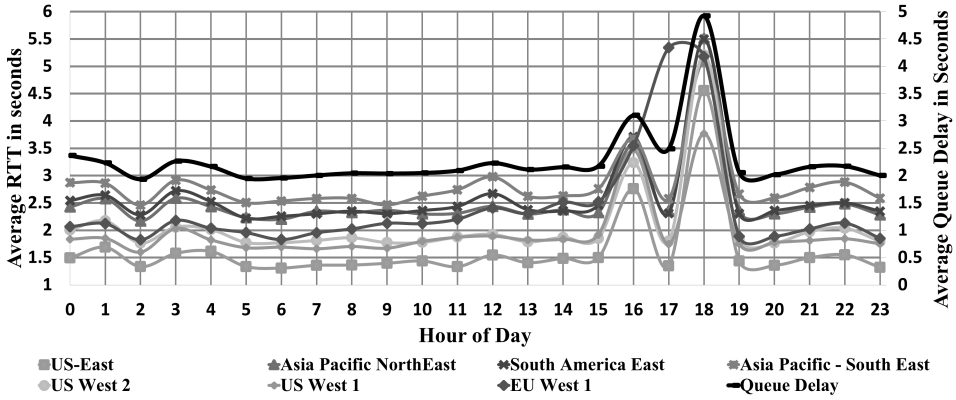


Figure 6.7: Round-trip time from multiple regions to a deployed GAE application with task queue delay.

Queue delays do vary but this variance (impact on RTT) is most perceptible during the early evening hours in all regions.

Finally, we compare our two methods for synchronization: EC and BE. EC uses a combination of the Task Queue API and synchronous URLFetch API; the use of the former ensures that all failed tasks are retried until they are successful. BE uses asynchronous URLFetch for all destructive updates and does not retry upon failures.

We ran the experiment for seven days and sent a total of 1195288 requests. Out of the 597644 packets (half of the total packets) sent via the TaskQueue option, 11679 were duplicates (unnecessary transfers). The asynchronous URLFetch experienced 10 duplicate packets suggesting the URLFetch API will retry in some cases from within the lower layers of the API implementation as needed. We experienced no update loss using EC and 5 updates lost for BE.

6.3.2 Benchmarks

We next consider the performance of five different and popular analytics benchmarks: wordcount, join, grep, aggregate, and subset aggregate. Wordcount counts the number of times a unique word appears. Join takes two separate tables and combines them based on a shared field. Grep searches for a unique string for a particular substring. Aggregate gives the summation of a field across a kind of entity, while subset aggregate does the same, but for a portion of the entire dataset (one percent for this benchmark). We implemented each benchmark using the Fantasm, Pipeline, and MapReduce GAE libraries, as well as a Hive query.

6.3.3 Google App Engine Analytics

For the experiments in this section, we execute each benchmark five times and present the average execution time and standard deviation. We use the automatic GAE scaling thresholds, and had billing enabled. We considered experiments with 100, 1000, 10000, and 100000 entities in the datastore. We attempted even higher numbers of entities, but the running time for each trial became infeasible to get complete results.

The tables in 6.2 shows the results for all of the benchmarks. The Fantasm implementation shows a large latency for a significant numbers of entities, and compared to Pipeline, is 6X to 30X slower. This is due to the fact that Fantasm's execution model

has a task for each entity, so it must do paging through the query[1]. Pipeline, by comparison, retrieves a maximum of 1000 entities at a time from the datastore, reducing the amount of time spent querying the database. Pipeline does not see much latency increases from 100 to 1000 entities, because both require only a single fetch from the datastore, and the difference lays in the summation. MapReduce also deals in batches, but the size of the batch depends on the number of shards. When the number of entities went from 100 to 1000 for MapReduce, the growth in latency was over 5X because the number of shards was one. 10000 entities, on the other hand, had 10 shards, and therefore did more work in parallel, seeing an increase in less than half the time. Pipeline has an advantage because of its ability to combine multiple entity values before doing a transactional update to the datastore, whereas both MapReduce and Fantasm are incrementing the datastore transactionally for each entity. For the implementation, the counter was sharded to ensure that there was high write throughput for increments.

Pipeline shows less overhead for Grep as compared to Aggregate (100-1000) because it uses half as many Pipeline stages. In the aggregate Pipeline implementation, there was an initial Pipeline which does the query fetches to the datastore, and another for incrementing the datastore in parallel after combining values. Grep, by comparison, does not need require combining or transactional updates, as required for the counter update in aggregate. Counter updates require reading the current value, incrementing it,

[1]The Fantasm library, since the writing of this chapter, has added the ability to do batch fetches for better performance.

and storing it back. Aggregate vs Grep MapReduce has a similar behavior to Pipeline because each mapper does not require transactional updates.

Figure 6.8: An identical benchmark run three times showing variability in run time.

The Join benchmark combines two different entity kinds to create a new table. The Join results show similar trends as Aggregate and Grep. During the experiments for Join, we experienced high variability in the performance of both the Pipeline and Fantasm libraries. Figure 6.8 shows a snapshot of three separate trials for Fantasm, in which noticeable differences in processing times occur. Multitenacy could be a primary reason for the fluctuations, yet the exact reasons are unknown and requires further study.

The Subset benchmarks queries a Subset of the entities rather than the entire dataset. Here we see that Fantasm does well, as this scenario was the primary reason for developing the library according to its developers [35]. Pipeline performs best, once again, because of its ability to batch the separate entities, and to not require separate web requests to process individual entities as Fantasm does. MapReduce suffers the most because it must map the entire dataset even though only a Subset is of interest.

For wordcount, MapReduce experiences its largest increase from 10000 to 100000 in this benchmark, which was due to several retries because of transaction collisions. The optimistic transaction support in GAE allows for transactions to rollback if a newer transaction begins before the previous one finishes. This is ideal for very large scale deployments, where failures can happen and locks could be left behind to be cleaned up after a timeout has occurred. Yet it is also possible to bring the throughput of a single entity to zero if there is too much contention. The performance of the wordcount benchmark can be improved by using sharded counters per word as opposed to the simple non-shared counter per word in our implementation. Built-in backoff mechanisms in the MapReduce library alleviates the initial contention, allowing the job to complete.

6.3.4 AppScale Library Support

We next investigate the use of the GAE analytics libraries over AppScale using the original Task Queue implementation in the GAE software development kit (SDK) and our new implementation based on the RabbitMQ (RMQ) distributed messaging system. We present only Pipeline results here for brevity (the relative differences between GAE and AppScale are similar). Table 6.3 shows the average time in seconds for the GAE applications executing over a 3 node Xen VM AppScale deployment. Each VM had 7.5GBs of RAM and 4 cores, each clocked at 2.7GHz. Note, that for the GAE num-

bers, we do not know the number of nodes/instances or the capability of the underlying physical machines employed.

Table 6.3 shows the RMQ execution time in seconds for each message size while Table 6.4 shows the SDK execution time in seconds for each message size. The SDK implementation enqueues the tasks as a thread locally rather than spreading out load between nodes. In addition, the SDK spawns a thread for each task which posts its request to the localhost. Tasks which originate from the local host will never be run on another node. RabbitMQ, on the other hand, spreads load between nodes, preventing any single node from performing all tasks. We are unable to run the 100K jobs using the SDK because the job fails each time from a lack of fault tolerance. If for any reason the node which enqueues the task fails, that task is lost and not rerun again. RabbitMQ, however, will assign a new client to handle the message, continuing on in the face of client failures. For larger sized datasets we also see a speedup because of the load distribution of tasks.

6.3.5 AppScale Hive Analytics

We next investigate the execution time of the GAE benchmarks using the Hive/Hadoop system. Figure 6.5 presents the execution time for the previous benchmarks using the Hive query language on a AppScale Cassandra deployment. There was no discernible difference between the sizes of the datasets, but rather the number of stages, where grep

only needed a single mapper phase, while the rest had both mapper and reducer phases. While slower for smaller sizes than the GAE library solutions, the Hive solution is consistently faster when dealing with larger quantities of entities (although it has the same issue as the MapReduce library when dealing with data subsets).

The Hive/Hadoop system in AppScale introduces a constant startup overhead for each phase (map or reduce) of approximately 10s. This overhead is the dominant factor in the performance. Once the startup has occurred, each benchmark completes very quickly. The numbers in the table include this overhead. Each of the benchmarks use a single mapper and reducer phase except for Grep. Our approach is significantly more efficient (enabling much larger and more complex queries) than performing analytics using GAE. Moreover, our approach significantly simplifies analytics program development. Each of our GAE benchmarks requires approximately 100 lines each to implement their functionality. Using our system, a developer can implement each of these benchmarks using a single line with fewer than 50 characters.

6.3.6 Monetary Cost

The cost of transferring data in GAE is dependent on two primary metrics: bandwidth out which is billed at .12 USD per gigabyte, and frontend instances, at .08 USD per hour. For low traffic applications, these costs can be covered by the free quota. For higher traffic, it is possible to adjust two metrics to keep cost down; the first is the max-

imum amount of time waiting before a new application server is started (where it will be billed for a minimum of 15 minutes), and the second is the number of idle instances that can exist (lowers latency to new requests in exchange for higher frontend cost).

We can compress data and work in batches to lower the bandwidth cost, seeing as how the additional latency for sending updates is between 4 and 7 seconds on average for the largest possible entity of 1MB. The compression execution time is added to frontend hour cost, and the level of compression is very dependent on the application's data (images, for example, may already be highly compressed). The average daily cost of the data transfer was 12.41 USD for frontend hours, 1.03 USD for datastore storage (went up over time), 2.55 USD for bandwidth, and 15.63 USD for datastore access. As future work, we are leveraging our findings to improve our datastore wrapper to minimize cost while still maintaining low latency overhead.

The cost for on-site analytics such as Fantasm and Pipeline is based on datastore access, both for reading the data which is needed for operation, and metadata for tracking the current progress of a job. The other cost associated is the frontend instance hours. The cost for running Pipeline for wordcount on 100000 entities was 0.34 USD (not accounting for the free quota), where 0.056 USD was frontend hours, 0.13 USD was datastore writes, and 0.154 USD on datastore reads. The cost of datastore writes is highly dependent on the number of indexed entities, and therefore if the entities have more properties, the writes can multiply quickly as would cost (each index write counts

as a datastore write). In general, it is difficult to predict the cost of GAE analytics. Our approach allows developers to perform analytics repeatedly without being charged at the cost of data transfer.

Our other option for downloading the data is via bulk transfer using tools provided by the SDK. We investigated the use of such tools but we ran into difficulties where exceptions arose and the connection would drop. Multiple attempts were needed, driving cost up as much to 5 to 6 times the cost of a daily experimental run (from 15 USD to 86 USD) before being able to complete a full download of the data. It took 9520 seconds on average for the three successful downloads of a dataset of 202MB. This option is clearly not acceptable for hybrid analytic clouds.

6.4 Summary

Cloud computing has seen tremendous growth and wide spread use recently. With such growth comes the need to innovate new methods and techniques for which extant solutions do not exist. Online analytics processing systems are such an offering for Google App Engine, where current technology has focused on web application execution at scale and with isolation, and existing solutions have operated within the restrictions imposed.

In this chapter we have described, implemented, and evaluated two systems for running analytics on GAE application, running current libraries in AppScale through the implementation of a distributed task queue, and the ability to run SQL statements on cross-cloud replicated data. Future work will carry forward our findings to optimize cross-cloud data synchronization as well apply our system to another use case: disaster recovery.

	100	1000	10000	100000
Fantasm	13.80 ± 1.61	110.29 ± 5.00	1148.24 ± 86.20	11334.59 ± 1047.57
Pipeline	2.46 ± 0.86	3.05 ± 0.32	11.08 ± 0.50	98.34 ± 3.82
MapReduce	9.34 ± 0.35	57.36 ± 8.96	104.56 ± 17.83	377.70 ± 63.35

Aggregate

	100	1000	10000	100000
Fantasm	10.85 ± 0.77	121.21 ± 21.07	1819.86 ± 1175.19	10360.40± 396.56
Pipeline	2.40 ± 1.26	2.663 ± 0.51	9.77 ± 0.72	98.89 ± 13.76
MapReduce	2.73 ± 0.30	4.56 ± 0.09	24.05 ± 0.30	227.57 ± 20.76

Grep

	100	1000	10000	100000
Fantasm	10.71 ± 1.22	109.83 ± 4.90	977.23 ± 80.34	10147.75 ± 1106.15
Pipeline	4.54 ± 2.34	14.48 ± 5.22	44.11 ± 12.57	159.96 ± 73.30
MapReduce	6.28 ± 1.43	40.18 ± 1.66	66.76 ± 10.92	256.40 ± 11.16

Join

	100	1000	10000	100000
Fantasm	0.58 ± 0.30	3.54 ± 0.28	16.95 ± 1.34	78.28 ± 10.62
Pipeline	1.97 ± 0.05	2.04 ± 0.20	2.01 ± 0.09	3.81 ± 1.60
MapReduce	2.67 ± 0.24	5.42 ± 0.45	27.66 ± 1.74	237.75 ± 12.00

Subset

	100	1000	10000	100000
Fantasm	12.22 ± 3.20	105.82 ± 8.45	1022.96 ± 72.85	10977.50 ± 1258.76
Pipeline	3.63 ± 0.74	4.97 ± 0.92	25.89 ± 8.92	222.14 ± 9.02
MapReduce	6.40 ± 0.96	42.70 ± 0.72	134.88 ± 9.59	840.71 ± 125.15

Wordcount

Table 6.2: Execution time in seconds for the benchmarks in GAE.

	100 RMQ	1000 RMQ	10000 RMQ	100000 RMQ
Aggregate	3.02	5.72	183.93	610.12
Grep	5.37	16.90	205.53	862.36
Join	2.72	5.16	165.03	455.31
Subset	2.45	3.12	12.61	786.53
Wordcount	7.41	11.43	311.52	635.28

Table 6.3: Execution time in seconds for benchmarks using the Pipeline library on AppScale with RabbitMQ (RMQ).

	100 SDK	1000 SDK	10000 SDK
Aggregate	3.77	6.14	N/A
Grep	6.11	28.88	260.03
Join	3.78	5.90	305.82
Subset	2.55	3.20	12.11
Wordcount	8.38	17.40	411.12

Table 6.4: Execution time in seconds for benchmarks using the Pipeline library with the AppScale SDK implementation.

	100	1000	10000	100000
Aggregate	20.59 ± 1.41	21.14 ± 0.55	20.30 ± 0.88	20.94 ± 0.59
Grep	11.90 ± 1.32	11.00 ± 0.58	11.17 ± 1.30	10.69 ± 0.44
Join	20.52 ± 1.01	20.71 ± 0.84	20.43 ± 0.57	23.41 ± 0.64
Subset	19.93 ± 0.54	20.07 ± 1.34	20.26 ± 0.86	20.66 ± 0.45
Wordcount	21.73 ± 1.50	22.13 ± 1.51	22.19 ± 0.96	21.54 ± 0.95

Table 6.5: Execution time in seconds for benchmarks using Hive.

Chapter 7

Spot Instances for MapReduce Workflows

MapReduce is a general computational model that originated from the functional programming paradigm for processing very large data sets in parallel. A scalable, fault tolerant approach of MapReduce has been popularized and recently patented by Google [30, 39]. This implementation operates on data in the form of key/value pairs and simplifies how large-scale data reductions are expressed by programmers. The system automatically partitions the input data, distributes computations across large compute clusters, and handles hardware and software faults throughout the process. Since the emergence, use, and popularity of MapReduce for a wide range of problems, many other implementations of the process have emerged. The most popular of which is Hadoop [43], an open-source implementation of Google MapReduce. Hadoop is currently in use by Yahoo!, Facebook, and Amazon, among other companies.

Given its ease of use and amenability to parallel processing, MapReduce is employed in many different ways within cloud computing frameworks. Google employs its MapReduce system for data manipulation within its private compute cloud and AppScale, the open-source implementation of the Google App Engine (GAE) cloud platform, exports Hadoop Streaming support to GAE applications [22]. The Amazon Web Services cloud infrastructure makes Hadoop and Hadoop Streaming available as a web service called Elastic Map Reduce [61].

In December of 2009, Amazon announced a new pricing model for AWS called Spot Instances (SIs). SIs are ephemeral virtual machine instances for which users pay for each completed runtime hour. A user defines a maximum bid price, which is the maximum the user is willing to pay for a given hour. The market price is determined by Amazon, which they claim is based on VM demand within their infrastructure.

If a VM is terminated by Amazon because the market price became higher or equal to the maximum bid price, the user does not pay for any partial hour. However, if the user terminates the VM, she will have to pay for the full hour. Furthermore, a user pays the market price at the time the VM was created, given that it survives the next hour. The cost of the hours that follow may differ depending on the market price at the start of each consecutive hour.

SIs are an alternative to on-demand and reserve VM instances in Amazon. On-demand instances have a set price for each hour that does not change. Reserve instances

have a cheaper per-hour price than both on-demand instances and SIs, but the user must lease the VMs for long periods of time (1 or 3 year terms). SIs therefore provide inexpensive computational power at the cost of reliability (variable and unknown VM lifetime). The reliability is a function of the market price and the users maximum bid (limited by their hourly budget).

In this work, to address the analytics portion of the thesis question, we investigate the use of SIs for MapReduce tasks. SIs fit well into the MapReduce paradigm due to its fault tolerant features. We use SIs as accelerators of the MapReduce process and find that by doing so we can significantly speed up overall MapReduce time. We find that this speedup can exceed 200% for some workloads with an additional monetary cost of 42%.

However, since SIs are less reliable and prone to termination, faults can significantly impact overall completion time negatively depending on when the fault occurs. Our experiments experience a slow down of up to 27% compared to the non-SI case, and 50% compared to an accelerated system in which the fault does not occur.

Since the likelihood of termination is dependent on the market price of the VM and the user defined maximum bid price, we investigate the potential benefit and degradation (cost of termination) of using SIs for MapReduce given different prices. We also use the pricing history of Amazon SIs to determine how much to bid as well as how many machines to bid for. By using this characterization for a given bid and market

price, we compute expected VM lifetimes for users. Such a tool enables users to best determine when to employ SIs for MapReduce jobs.

7.1 Background

We first briefly overview the Hadoop MapReduce process. Using Hadoop, users write two functions, a mapper and a reducer. The map phase takes as input a file from a distributed file system, called the Hadoop Distributed File System (HDFS), and assigns blocks (splits) of the file as key-value pairs to mappers throughout the cluster. HDFS employs replication of data for both fault tolerance and data locality for the mappers. Mappers (map tasks) consume splits and produce intermediate key-value pairs which the system sorts and makes available to the reducers. Reducers (reduce tasks) consume all pairs for a particular key and perform the reduction. Reducers then store the resulting output to HDFS. The result may be a final computation or may itself be an intermediate set of values for another MapReduce tuple.

Each machine is configured with a maximum number of mapper and reducer tasks slots. The number of slots depends upon the resources available (i.e. number of CPU cores and memory) as well as the type of job being run (CPU-bound versus IO-bound). The master runs a Job-Tracker process which assigns work to available worker slots. Slave nodes run Task-Trackers which have their task slots assigned work as it becomes

available by the Job-Tracker. Each Task-Tracker can run a custom configuration. It can be designated to run only mappers, only reducers, or, as is typical, some combination of the two.

Hadoop tolerates failures of a machine through the use of replication. Output data can be regenerated given there are live replicas of the input splits. The replication policy for Hadoop is rack-aware and will place one copy on the same physical rack and the second off-rack. Hadoop also tolerates bad records. Records which cause exceptions are retried a default of three times and then skipped to ensure the entire job is not halted due to a single bad record. This issue can come about when buggy third-party software is used.

Hadoop uses heart-beat messages to detect when a machine is no longer operable. Data which was lost due to a failure is replicated to ensure that the configured number of replicas exist.

The time for a MapReduce job in Hadoop is dependent on the longest running task. Tasks that are few in number and those that continue execution once most others have completed are called stragglers. The system can speculatively execute stragglers in parallel using idle task slots in an attempt to reduce time to completion. The authors in [93] provide details on the impact of stragglers in virtualized environments.

7.2 Data Analytic Cloud

In this section we describe what we envision to be a data analytic cloud which uses MapReduce for analyzing data and the cost associated with running such a service. The scenario which this chapter focuses on relies on a provider to host their large data sets in a public cloud. The data is stored in a distributed file system running on a subset of leased VMs in the cloud. In addition, the provider may provision the data analytic engine required for processing or querying the data. In this chapter we consider the MapReduce framework as that engine, although this work also carries to using higher level query languages such as Pig [74] and Hive [67]. Users submit MapReduce jobs, and the provider charges the user an hourly rate, along with the option to speedup their job at an additional cost. In order to maximize profit, the provider uses the cheapest source of computation available. Amazon's EC2 SI pricing is competitive in this area, being as low as 29% of the cost of an EC2 on-demand instance.

Amazon's Elastic MapReduce is another option available, giving users an easy and cost effective way to analyze large data sets. Data, at the time of writing this chapter, is free to upload into their Simple Storage Service (S3) and free to transfer within EC2, but transferring out is $0.15 per GB and storage per month per GB is $0.15. A 1TB set of data cost $150 per month to store, and $150 per transfer out. There is also an additional cost for PUT and GET request for S3 at $0.01 per 1000 requests. Elastic

MapReduce is spawned with a user defined number of instances. The user only pays for the number of VM hours used at an additional 17.6% charge of the on-demand VM instance price.

The minimum cost of hosting a 1TB data set in a Hadoop cluster (with a 3 year term) using just local instance disk space, with three times replication, costs $194 dollars a month (20 small instance VMs with 160GB each for a total of 3.2TB of distributed storage). The Elastic MapReduce service with S3 is more affordable if the total amount of cost for VM instances is less than $44 dollars a month, which affords 440 VM hours a month or 22 hours of processing for 20 small VM instances. The disadvantages of storing the data in S3 is that the MapReduce cluster loses the data locality a local HDFS cluster provides.

7.3 Analysis

We next investigate how best to employ Hadoop within a cloud infrastructure for which virtual machine instances are transient. Our goal is to investigate how best to do so given the Spot Instance (SI) option offered by Amazon Web Services (AWS). SIs offer a cost effective alternative to on-demand instances since the cost of their use is dependent on market-based supply and demand of instances. We find that SIs can be as low as 29% of the cost of on-demand instances. SIs trade off termination control for

such cost savings. SIs are good for short running jobs that can tolerate termination, i.e. faults in the execution process. MapReduce is an ideal candidate for SIs since we can use additional nodes to accelerate the computation.

However, since the time to complete a MapReduce process is dependent upon how many faults it encounters, we must also consider SI termination. Since SI termination is dependent upon market price and maximum bid price, we are interested in using this information to estimate the likelihood of termination.

To enable this, we consider bid prices independent of market prices since there is very limited information available from Amazon as to how they determine the market price. Amazon does not reveal bids by users or the amount of demand. Table 7.1 shows the pricing of different instance types in the western US region. The SI pricing is an average of prices since they were first introduced in December of 2009 till March of 2010. The small instance type uses a 32-bit platform, while the rest are 64-bit. An EC2 compute unit is equivalent to a 1.0 to 1.2 GHz 2007 Xeon or Opteron processor [2].

7.3.1 Spot Instance Characterization

We model the SI lifetimes by building a Markov Chain with edges being the probability of price transitions for each hour interval. Given the transitional probabilities we

Instance Type	Average Price	StDev	On-Demand Price	EC2 Compute Units	Memory (GB)	Storage (GB)
m1.small	$0.0399	0.001327	$0.095	1	1.7	160
c1.medium	$0.0798	0.002551	$0.190	5	1.7	350
m1.large	$0.1673	0.04163	$0.380	4	7.5	850
m2.xlarge	$0.2397	0.007489	$0.570	6.5	17.1	420
m1.xlarge	$0.3197	0.009045	$0.760	8	15	1690
c1.xlarge	$0.3233	0.02469	$0.760	20	7	1690
m2.2xlarge	$0.5593	0.01756	$1.340	13	34.2	850
m2.4xlarge	$1.1164	0.03288	$5.08	26	68.4	1690

Table 7.1: Prices of different VM instances from the US west region. Instances labeled with ”m1” are standard instances, ”m2” are high-memory instances, and ”c1” are high-CPU instances. EC2 compute units are based on CPU cores and hardware threads. All instances here are for the Linux operating system. Costs are obtained from [2].

can calculate n-step probability using a variant of the Chapman-Kolmogorov equation:

$$P(i, b, n) = \sum_{j \notin B} M_{ij} P(j, b, n-1) \tag{7.1}$$

where

$$p(i, b, 0) = \begin{cases} 0 & \text{if } i \in B \\ 1 & \text{if } i \notin B \end{cases} \tag{7.2}$$

The starting market price at the time of VM creation is i and n is the number of time unit steps. The set of prices which are over the bid price, b, are in set B. M_{ij} is the probability matrix of a price point from i to j. Pricing history was collected over time using Amazon’s EC2 tools and can be attained from [27]. $P(i, b, n)$ is solved

recursively, where each step depends on the previous one. The base case is a binary function of whether or not the bid price is greater than the market price.

Figure 7.1 shows the probability of a VM running for n hours. The figure has different maximum bid prices given the market price being $0.035 at the time of starting the instance. As the maximum bid decreases, the probability of the SI staying up decreases as well. Some small increments in the bid price can give much larger returns in probabilities as can be seen when incrementing the bid price from $0.041 to $0.043, whereas other increments give very little return ($0.037 to $0.039). A SI in this case has more than an 80% chance of making it past the first hour given the market price was less than the bid price at the start of the VM. Bids that are less than or equal to the market price at the start of the VM would stay at 0% probability ($0.035 for example which is not viewable because it is directly overlaid on the x-axis).

Figure 7.2 has two sets of data for comparison. The data set labeled ”A” is from mid-January 2010 to mid-March 2010, while the data labeled ”B” is from mid-March 2010 to the end of May 2010. A comparison of the two models shows that the past pricing model is a good indicator of future pricing. It should be noted that data prior to mid-January was not used in building the model as shown in Figure 7.1 because as reported in [3] there was a bug in the pricing algorithm prior to this date which has since been fixed. The bug’s impact can be seen in the pricing visualized in [27] where

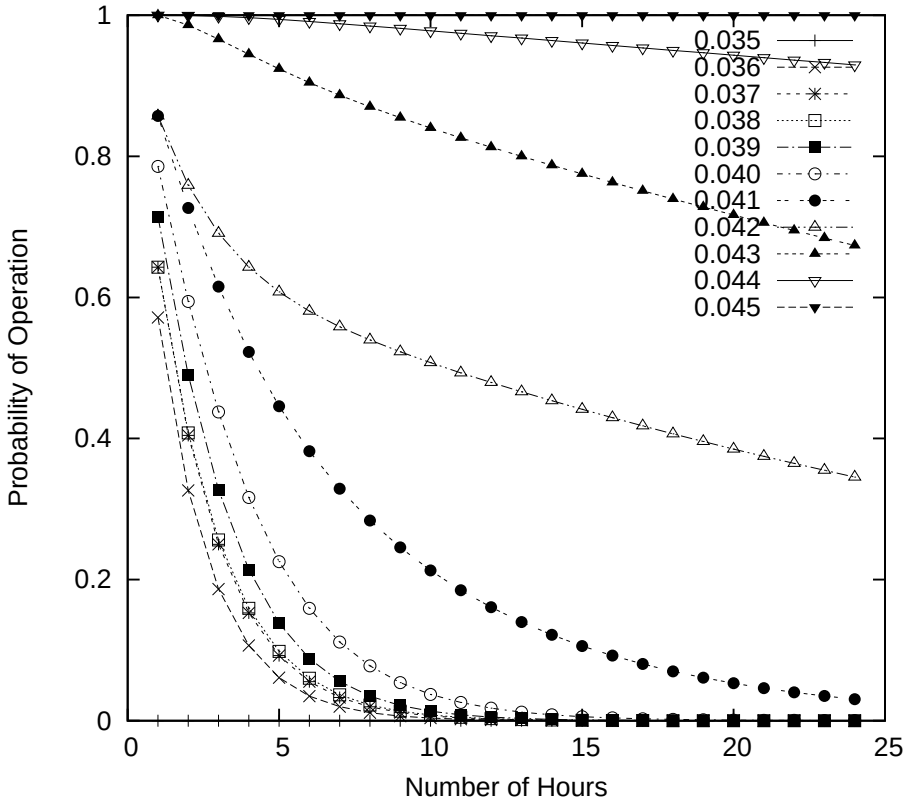


Figure 7.1: The probability of a small VM instance staying operational over time given a starting price of $0.035 with varying bids.

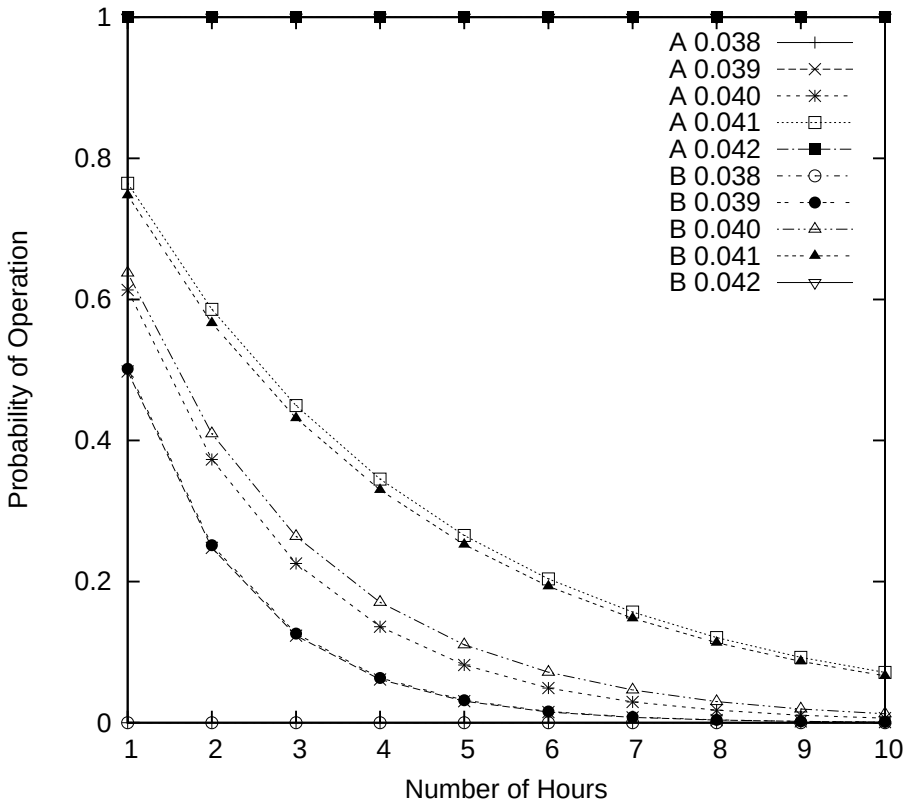


Figure 7.2: A comparison for verifying the pricing and lifetime model of a small VM instance given a starting price of $0.038 and varying bids.

prices stabilized post January 15th. This explains the smaller range of prices between Figure 7.1 and Figure 7.2.

Using Equation 7.1, we can calculate the expected lifetime, l, of a VM given a starting market price, i, a given bid, b, and a max run time of τ time units:

$$E(l) = \sum_{n=1}^{\tau} nP(i, b, n) \tag{7.3}$$

We can determine the amount of expected work a VM should achieve given the lifetime of a VM. This value can be used in the planning of backing up data and hence reduce the impact of failure. Moreover, it can be used in bidding strategies to ensure the greatest amount of SIs can be requested without fear of going over your maximum allocated budget.

7.3.2 Cost of Termination

We define the cost of a termination as the amount of time lost compared to having the set of machines stay up until completion of the job. The minimum cost is

$$\delta + (fM/s)/(s - f) \tag{7.4}$$

where the total time taken to complete the mappers is M. The total number of mappers is a function of block size and the size of the input file. The total number of slaves is s, the number of machines terminated is f, and the time spent waiting for a heart-beat timeout to occur while useful work could be done is δ. Early termination of a machine into the map phase allows for an overlap of when the termination is detected and the rest of the cluster is doing useful map work (i.e. no map slot goes idle). Work is equally divided given the machines are homogeneous.

Termination also results in the loss of reducer slots if that machine was configured so. This may or may not be an additional cost of failure depending on the job configuration which can specify the number of total reducers. This potential cost is not reflected in Equation 7.4 due to its application specific and configuration specific nature. Termination after all the mappers have finished, sees the most expense of the fault detection, forcing a re-execution of all mappers completed on the terminated machine, even those which have been consumed by reducers and will not be consumed again.

7.3.3 Evaluation

Our initial experiment consists of five small-sized on-demand instances on EC2 with one node as the master, and four as slaves. The slaves were configured with two map slots and one reduce slot. Additional EC2 SIs which are added for speedup also have the same configuration. Each data point is an average of five trials. The applica-

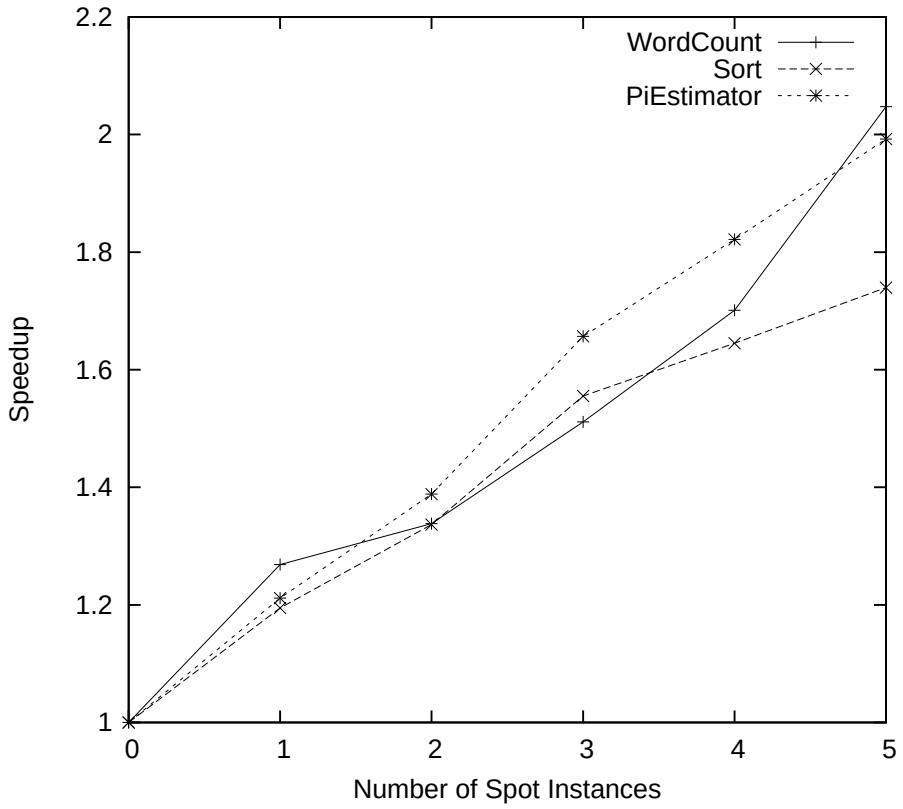


Figure 7.3: This graph shows the speedup of three different applications. The x-axis shows the number of SIs used in addition to the original HDFS cluster. Each data point represents the average of 5 trial runs.

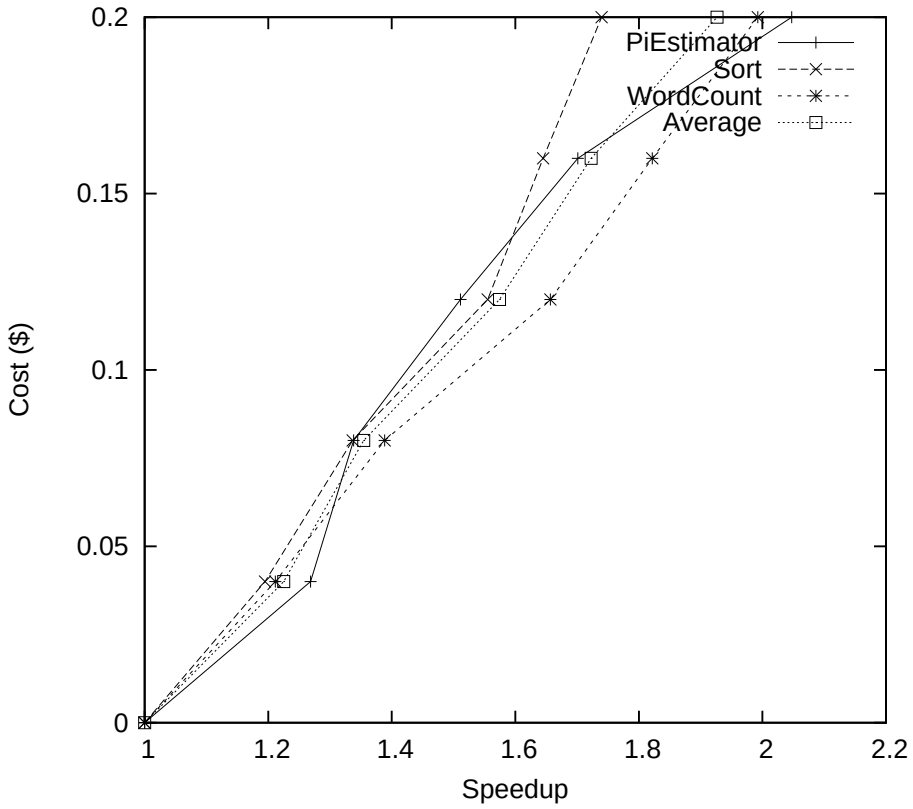


Figure 7.4: This graph shows the cost versus the speedup. The baseline hourly cost for the HDFS cluster is $0.475 per hour. Each additional SI instance cost $0.040.

tions are wordcount, pi estimation, and sort. Wordcount counts the occurrence of each word of an input file. Pi estimation uses a quasi-Monte Carlo procedure by generating points for a square with a circle superimposed within. The ratio of points inside versus outside the circle is used to calculate the estimation. Sort uses the MapReduce framework's identity functions to sort an input file.

Figure 7.3.3 has the speedup of each job with the number of SIs varied. The speedup is normalized to the original HDFS cluster configuration. Speedup is linear for all three applications. The price for speedup is in Figure 7.3.3 where each additional SI cost $0.04 per hour. Each job ran for less than one hour, therefore had the job been running for n hours, the y-axis would be a multiple of n.

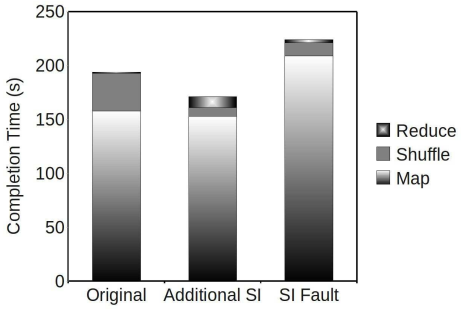
Our second experiment was using five machines as the HDFS cluster, and one machine as an accelerator. Figure 7.5 has the speedup breakdown of adding an accelerator as well as the relative slowdown when the SI is terminated halfway through execution. The detection of machine faults was set to 30 seconds to minimize δ for these experiments, where the default is 10 minutes. The default delay is set sufficiently large for the purpose of distinguishing between node failure and temporary network partitioning and had our experiments used the default value δ would have grown accordingly.

For Figure 7.5 the mapper portion is from when the first mapper begins and the last mapper ends. The shuffle period is where map output is fetched by reducers. This phase runs in parallel with mappers until the last mapper output is fetched. Reducers

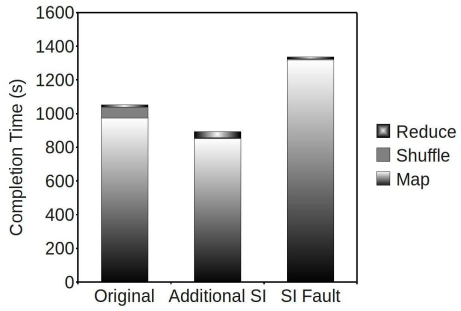
fetch and merge-sort the data as map output becomes available. They finish merge sorting the remaining intermediate data and proceed to run the reduce procedure once the shuffle phase is complete. The reduce procedure does not start until all map output is accounted for.

As expected, we see speedup for all applications with the addition of an SI. Yet the cost of losing the accelerator actually slows down the application sufficiently, to the point where it was faster with the original setup. If the SI ran longer than an hour, it would have cost the user money with no work to show for it. On the other hand, if the SI was terminated before the first complete hour, no money is lost. The completion time is hampered in both cases. Section 7.4 presents the solution we are pursuing to alleviate this problem.

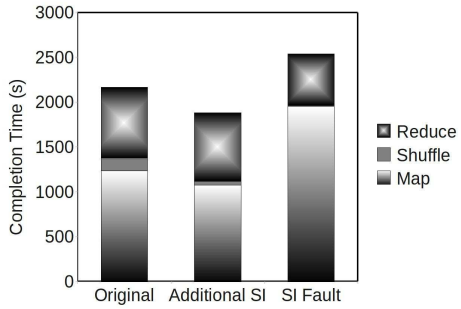
The addition of SIs improves the completion time of the mappers, but may not improve the completion time of the reduce phase. Many applications have a sole reducer at the end of the map phase because it requires a holistic view of the map output. Additionally, the runtime of the reducer is dependent on the amount of intermediate data generated. The amount of intermediate data is subject to the MapReduce application, and the input data. The use of a combiner also reduces the amount of intermediate data, which is invoked at the end of a mapper performing an aggregation of mapper output. Our wordcount benchmark uses a combiner which essentially does the same



(a) Pi Estimator



(b) Word Count



(c) Sort

Figure 7.5: Completion time for three different applications showing runtime for MapReduce on the original on-demand HDFS cluster, with one SI, and with an SI termination halfway through completion (85, 450, 940 seconds for Pi Estimation, Word Count, and Sort, respectively).

job as the reducer at a local level. The aggregated map output helps to decrease the reducer workload in both the amount of data which must be fetched and processed.

7.4 Discussion

Had we kept adding SIs to the system in our first experiment, we would expect to get a diminishing return in the amount of speedup an application sees. For each SI, data must be streamed to it from the HDFS cluster which is hosting the input file. Moreover, there may be a tipping point in which the HDFS cluster is overburdened with too many out going data transfers, and the addition of an SI would result in a slowdown. We are pursuing discovering where this breaking point is, and what the ratio of HDFS VMs to SI VMs are for different applications.

We also ran experiments with accelerator nodes only running mappers. Our first notion was that mapper output which was consumed by a reducer would not be re-executed in case of a failure. This assumption was wrong. All mappers are re-executed on a machine regardless of whether it will be consumed again. Reducer output is already stored in HDFS with default replication of three. Check-pointing the map functions can be done by replicating the intermediate data as done in [51]. Other methods include saving the intermediate data to Amazon's S3 or EBS. The current Hadoop

framework can also be modified to use tracking data on which mappers have been consumed to prevent re-execution during a fault.

Future work includes analyzing the cost-to-work ratio of different VM instance types. SIs can be used as probes for determining the best configuration. But this is only after fixing the availability of mapper output after termination, since we want to be able to restart the Task-Trackers with the optimal discovered configuration. The optimal configuration consists of having the most amount of mappers and reducers without them hitting performance bottlenecks due to sharing CPU, disk, network, and memory resources. Without the ability to save their intermediate data, the probes would become liabilities for wasted computation. Furthermore, we plan on investigating the use of heterogeneous configurations and instance types where a portion of the VMs only run reducers or mappers.

Additional future work includes analyzing the effects of staggering max bid prices across a set of SI VMs. In such a case it would be possible to only lose portions of accelerators at a time, essentially giving some VMs priority.

The nature of SI billing also leads to an interesting discussion on how to maximize utilization. An SI will be billed for the entire hour if terminated by the user even though it was only used for a partial hour, while no billing results if the VM ran for a partial hour and Amazon terminates the instance due to a rise in the current market price. Users may want to terminate an instance after an hours time in order to only pay for a full hour

usage rather than pay for a partial hour, but this is only wise when the framework can recover from failure without significant adverse affects on the completion time.

7.5 Summary

We have presented SIs as a means of attaining performance gains at low monetary cost. We have characterized the EC2 SI pricing for informed decisions on making bids given the current market price. Our work has shown that due to the nature of spot instances and their reliability being a function of the bid price and market price, MapReduce jobs may suffer a slowdown if intermediate data is not stored in a fault tolerant manner. Moreover, a fault can cause a job's completion time to be longer than having not used additional SIs while potentially costing more.

Chapter 8

Infrastructure Agnostic and Datastore Agnostic Live Migration of Private Cloud Platforms

Companies with on-premise Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) systems employ private cloud technology, which provides the flexibility and power of the public cloud, yet allows for the utilization of on-premise resources and infrastructure.

As more and more companies go towards private PaaS offerings, there is a critical concern for providing high reliability and availability while also enabling the ability to perform updates on the underlying hardware and software resources. At the OS level, within individual VMs, security patches must be installed that may require the system to be rebooted. At the PaaS level, user applications rely on a multitude of software subsystems that may be frequently updated (e.g., load balancers, application servers,

and databases). Moreover, hardware updates can occur when moving to higher end servers, or moving to higher performing storage options (such as solid state drives).

Open source PaaS technologies rely on a multitude of components, which themselves are comprised of open source solutions that are rapidly changing. These changes come in response to getting community uptake or decline in popularity, for reasons such as performance and reliability. The communities following NoSQL datastore technologies, where there are well over 150 different options [72], are a prime example where there are constant shifts between selections as technologies improve with better performance and newer feature sets. Yet, while the capability to swap out a datastore should be possible, developers of such technologies are not incentivized to create portable systems.

We address the thesis question and the requirement that real PaaS systems face for frequent upgrades and the desire to swap out technologies with minimal downtime by using a technique called *live migration*. With live migration, PaaS users can be transplanted from one underlying technology to another, whether that technology is the virtualization layer, the IaaS, or some component technology of the PaaS, with minimal service disruption.

We do so by extending the AppScale PaaS framework. AppScale can Google App Engine (GAE) applications and do so scalably while supporting multiple infrastructures

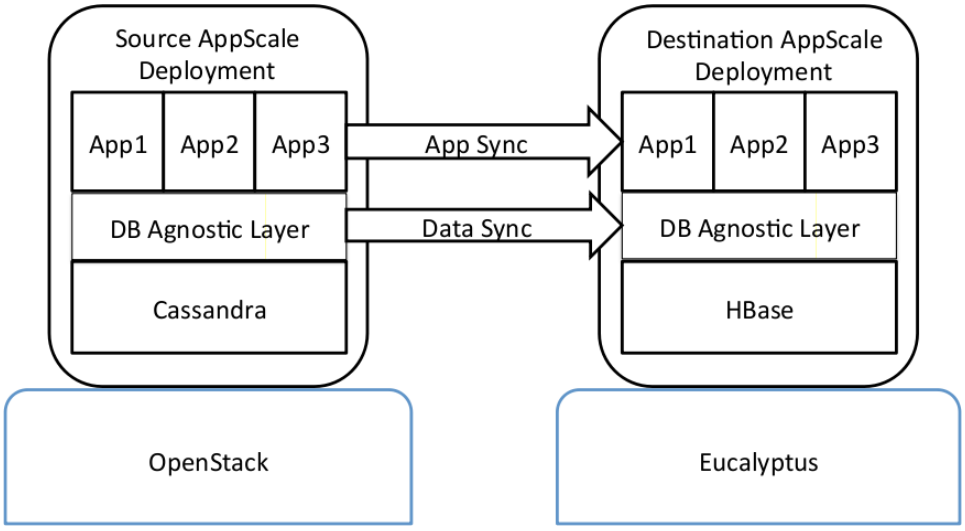


Figure 8.1: Live migration in AppScale.

and datastores. AppScale has plug-in capability for datastores, supporting datastores such as Cassandra, HBase, Hypertable, and MySQL Cluster as detailed in Chapter 4.

Figure 8.1 shows an example of a live migration of two different AppScale deployments, where the underlying IaaS system and datastore used are being updated. In this chapter we address the need to be able to move applications and tenants from one PaaS deployment to another, and to leverage the elasticity of private cloud infrastructures to perform live migrations.

In the sections that follow, we first provide required background on AppScale and its data model. We then describe the requirements, design, and implementation of our live migration system and show an evaluation of our live migration support between two deployments of AppScale, where we transition the datastore used from Cassandra

to Hypertable. Our evaluation looks at several components of the system, including the synchronization of our distributed transaction manager, datastore performance, and switchover time.

8.1 Background

AppScale provides GAE application portability as well as infrastructure and datastore agnosticism. It provides this portability by implementing the GAE APIs, doing so scalably and with fault tolerance. While there are many APIs supported by AppScale for GAE compatibility, the only system state that requires migration is the datastore, as the other APIs are stateless or have no impact on correctness if transferred to a secondary deployment. Yet, for performance reasons we also address the preloading of memcache, a distributed memory caching system meant to alleviate load on the datastore, as to prevent having a cold cache upon the traffic handover.

Infrastructure agnosticism comes by the way of how AppScale is packaged as a virtual machine image. Any virtualization technology capable of running a Ubuntu virtual machine image can run AppScale (e.g., Xen, KVM), and any IaaS that is EC2 compatible (e.g., Eucalyptus, OpenStack) allows for AppScale to be automatically deployed over a varying number of nodes at initialization.

AppScale employs an abstraction layer above the datastore, allowing for the plugging-in of a variety of NoSQL technologies, which are automatically deployed at initializa-

tion. We contribute a unifying data migration layer that now allows for the ability to do rolling upgrades to new versions of the existing datastore or an entirely different datastore, a feature that many NoSQL datastores do not currently support.

The datastore layer within AppScale was extended to provide ACID transaction support, regardless of the underlying datastore [20], via a distributed coordinator. Lock granularity for transactions is at an "entity group" level, where entities that share a common root entity are within the same group. These groups are detailed by the developer within their application, and cannot be changed thereafter without deleting the entities.

Moreover, the query support in GAE, and thus AppScale, is limited to only scalable operations. There is no support for JOINs, MERGEs, or queries which can do INSERTs, and hence all queries perform read-only operations. Since queries which can be performed in GAE are derived from the ability to do range queries on the datastore, certain queries are not allowed, such as inequality filters on multiple properties.

Related work includes Albatross [29], a migration technique for moving tenants in a cloud system between deployments. While we can also provide per-tenant movement between deployments, our data model allows for the capability to update the software stack at multiple levels, all while maintaining backwards compatibility with running applications. Furthermore, while much research has been done in VM migration [60], it does not address the problem of performing software stack upgrades above the IaaS layer or allow for per-tenant migration.

8.2 Design and Implementation

Live migration of data must adhere to certain requirements, such as high availability, backward compatibility, a minimal number of failed transactions, and minimal performance degradation. We have designed and implemented our PaaS migration techniques within AppScale with these requirements and metrics in mind. To do so, we leverage existing components, including the datastore-agnostic transaction support. Migration requires multiple phases, in which state is synchronized between two separate deployments. Figure 8.2 shows the different stages required to make a full transition from the current deployment to the next.

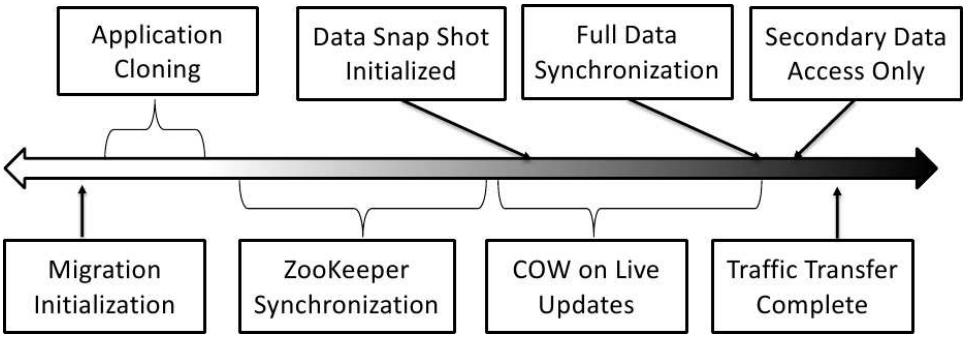


Figure 8.2: Timeline of the migration process.

8.2.1 Migration Initialization

The first steps in our migration process require the configuration and deployment of a secondary AppScale instance (N_2), initiated by the primary AppScale instance (N_1).

N_2's firewall is opened up to allow access by N_1 to control N_2's network channels (such as SOAP servers).

Once N_2 has been successfully initialized, N_1 utilizes the AppScale command-line tools (a toolset which cloud administrators can use to interact with AppScale deployments) to upload copies of the applications running in N_1 to N_2. At this point, no data has transferred and the applications themselves, while running, are not being accessed by users. We currently do not support the uploading of new applications to N_1 while the migration is taking place.

8.2.2 Metadata Synchronization

ZooKeeper is a distributed coordination system that AppScale employs to manage state between different services within a deployment, as well as for locking to provide transactional semantics, as explained in [20].

After the N_2 ZooKeeper instances are up, nodes are automatically synchronized with N_1 for new updates, as a consensus is required via the Paxos algorithm once they have joined the cluster. Existing data is then made available to the new ZooKeeper nodes by doing the synchronization functionality in a depth-first search. ZooKeeper nodes are decommissioned at N_1 after the full migration is complete.

8.2.3 Memcache Warm-up

Our objective for memcache is to have a warm cache in N_2 by the time the handover takes place. For this we do not require full synchronization but a best effort to keep all relevant and most recently used data in the cache. We achieve this by employing copy-on-write (COW) and also copy-on-read (COR) for memcache updates to N_2. The local read or write happens in parallel to the remote write to minimize overhead. We do not do asynchronous updates as to adhere to cache coherency when entries are invalidated. This step is initiated as soon as N_2's memcache system is operational (not shown in Figure 8.2).

8.2.4 Data Synchronization

After synchronizing the metadata in N_1, we can now synchronize application data. The data access layer at each node has a REST interface that signals the stages of migration the process should be in. Each datastore process on each node is sent a message containing the IP address of N_2. Upon receiving this message, any writes or deletes are forwarded to N_2 in a copy-on-write manner.

It should be noted that because of the GAE Datastore API's transaction semantics, writes and deletes are always part of a transaction, even if the transaction is only a single operation. Therefore, each operation requires that a lock be acquired and held through ZooKeeper (which is shared state between deployments). Furthermore, COW

updates are done in parallel with local writes to the transaction journal and datastore, to minimize latency.

Transactions must always verify that if it started during normal operation that it did not transition into COW mode mid-transaction upon being committed. If so, the transaction must be retried to ensure that its state is successfully synchronized with the secondary deployment via COW. By default, failed transactions will retry up to three times, before they permanently fail.

Once all datastore access layers acknowledge they are in COW mode, then the datastore snapshot process can begin. COW updates start before the snapshot is started and proceeds during and after, as to make sure no new updates are lost. The updates themselves are SOAP calls to a migration service running on N_2 which uses the datastore agnostic API.

A full snapshot of the datastore consists of serializing each table into a set of flat files which are then compressed. Each independent file can be loaded into N_2 in parallel as an optimization, yet we currently do it serially for simplicity.

The completed snapshot is then copied over to N_2 where it is loaded into the datastore via the datastore agnostic transaction layer [20]. Updates are done transactionally, where the key is first checked to make sure no live updates were done to the entry before updating it. This is only possible because ZooKeeper state is currently shared between deployments. If an entry has been updated during a live datastore write, the

snapshot version is simply ignored, as it is stale data (a journaled version will still be available if a rollback is required). Furthermore, it is not possible for an entry to be loaded into N_2's datastore while an ongoing transaction is in place at N_1. N_2 will fail to get the lock on the given entity group and will exponentially backoff until the lock can be attained. After the lock has been acquired, it will then check to see if the given entry already exists, where it will find an entry due to the aforementioned transaction, and thus move onto the next entity to load.

8.2.5 Traffic Handover

Once full data synchronization as been achieved we then switch N_1 as a full proxy for data access to N_2, making it the primary replica for data access. This step is required as we make the transition onto N_2 for the traffic handoff.

We have two stages of traffic switching. The first stage does permanent redirects at the proxy routing layer (nginx), but because we cannot guarantee that all proxies on all nodes force redirection at the same time we require the full data-proxy stage to make sure there is no case where a user who has not been routed over does not see updates made by a user at the secondary deployment (independent updates at N_2 are not synchronized back to N_1).

Second, we use DNS updates to make sure that the secondary deployment has subsequent traffic from new users. DNS updates alone do not suffice, as many clients cache the DNS entry and it may take ample time before it refreshes its entry. Amazon's

Route 53 was the DNS service we used because of its high availability and scalability. Modifications to the DNS was done using their RESTful API which allows for dynamic updates. Our updates consisted of updating the resource record field to point from N_1's IP to N_2's IP.

8.2.6 Fault Tolerance

In a distributed setting we are able to leverage AppScale's current fault tolerant capabilities for live migration. If transactions fail during a live migration the transaction handler identifier is recorded into ZooKeeper which is shared state between deployments. Any reads of an entity that has a blacklisted transaction identifier is ignored, and the correct version identifier, which is saved in ZooKeeper, is fetched from the transaction journal. While data is currently checked with md5 hashes when transferred across nodes to prevent data corruption, we do not handle Byzantine faults.

8.3 Evaluation

In this section we measure the overhead associated with live migration between one AppScale deployment to a secondary. We do so with two single node deployments of VMs with 7.5GB of RAM and 4 CPU cores. The initial deployment had Cassandra 1.0.7 as its storage layer, while the secondary deployment had Hypertable 0.9.5.5. The testing application was a GAE application with a RESTful interface. Reads and writes were done based on parameters passed to this application per request.

Lock Count	Min	Mean	Stdev	Max
1000	4.44	4.49	$\pm$0.03	4.53
5000	6.57	6.58	$\pm$0.04	6.62
10000	8.35	8.47	$\pm$0.07	8.53
50000	23.6	23.8	$\pm$0.13	24.0
100000	42.8	43.0	$\pm$0.22	43.4

Table 8.1: Time in milliseconds required for lock synchronization on a new ZooKeeper node with a varying number of lock entries.

We first measure the time to synchronize our locking system with ZooKeeper. Next we empirically evaluate the time taken to upload different sized entities from a snapshot. Furthermore, we look at the overhead of updates to both the datastore and memcache which occurred during live migration. Lastly, we quantify the latency associated with a switch over using Amazon's Route 53.

8.3.1 ZooKeeper Synchronization

Table 8.1 shows the time taken for synchronizing a node given a different amount of ZooKeeper nodes in which transactional lock states are stored. The number of root entities signifies the number of locks required, and thus need to be synchronized. We see that the time taken on average has sub-linear growth as the number of entries grow while maintaining a relatively low standard deviation.

8.3.2 Memcache

We measured the time taken for migration of reads and writes to memcache and measured the overhead compared to normal operation. For entity sizes of 5KB, we

State	Read %	Min	Mean	Stdev	Max
N	20	46.4	107.2	±23.6	240.2
N	50	44.5	100.2	±24.5	223.9
N	80	44.3	94.0	±24.1	348.1
M	20	43.7	115.8	±32.4	536.5
M	50	45.7	103.3	±27.1	274.5
M	80	47.0	94.2	±23.0	233.1

Table 8.2: A comparison of time taken for request in milliseconds between normal operation (N) and live migration (M) for different workload percentages of reads versus writes.

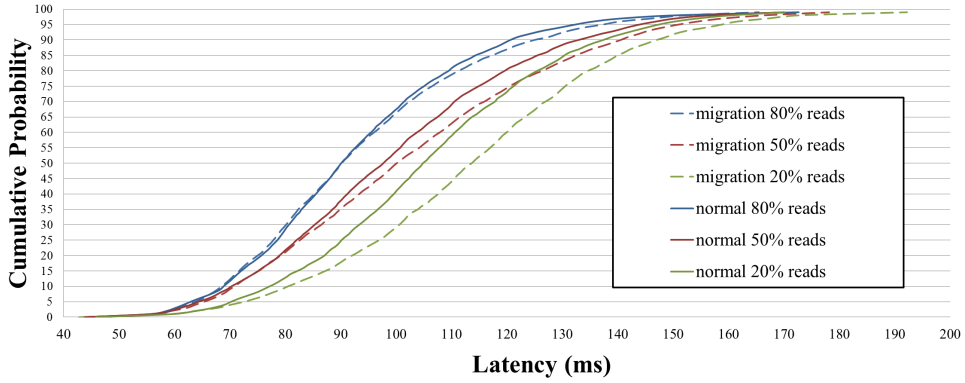


Figure 8.3: CDF of latency of different work loads comparing normal operation to live migration. The x-axis is latency in milliseconds.

found that COW added 0.17ms of overhead, while COR added 0.85ms, both adding less than one percent overall overhead per user request when doing both local and remote updates in parallel. COR added slightly more over overhead because writes are 10.3 times longer compared to a local read.

8.3.3 Datastore Performance

Table 8.2 has a comparison of the average latency with different workloads, from a 20/80 read-to-write ratio, to an 80/20 ratio. We compare the initial state (pre-migration)

Size	Min	Mean	Stdev	Max
100B	1.72	2.45	±0.92	20.53
500B	1.71	2.29	±0.80	19.08
1KB	1.70	2.43	±0.97	17.03
5KB	1.78	2.62	±0.75	12.53
10KB	1.79	2.71	±1.09	18.26
50KB	1.82	2.88	±1.03	20.02
100KB	2.17	3.18	±1.00	24.45

Table 8.3: Time for transactionally loading entities of different sizes into Hypertable through the datastore agnostic transactional layer. Times are in milliseconds.

and during migration with 100,000 updates. Load is generated using the Apache Benchmark Tool with a concurrent setting of 10 requests which maxed out all the CPU cores. Read heavy operations see the least amount of overhead as it does not require copy-on-write operations with the secondary deployment. As Figure 8.3 shows that there is more overhead associated with write heavy workloads, yet because updates to the remote deployment are done in parallel with the local writes to the datastore, we minimize the additional required latency. Overall we see the overhead at an average of 7.4% with write heavy workloads, while being negligible for read heavy workloads at 0.2%. The most write heavy workload also sees a longer tail past the 95th percentile, from 170ms to 190ms. For both scenarios no failed requests were reported.

Table 8.3 presents the time taken for different entity sizes when loaded from a snapshot. For this experiment 10,000 updates of each size were loaded and measured. These times include the time to acquire the lock, to check if the current key had an existing

value, and to do the write. There were insertion times over 20ms as the max times show, but these were well into the 95th percentile (CDF not shown).

8.3.4 Traffic Handover

We use the AWS REST-based API to dynamically update the resource record names in Route 53. We measure the switchover time with the Apache Benchmark Tool which continuously sends HTTP request to the initial deployment. The average time to switch over was 46.4 seconds with a standard deviation of 0.97 with a total of 10 trials. The time measured is the difference between when the first HTTP request appears in the access logs of the secondary deployment to the the initial time the API request was sent.

8.4 Summary

In this chapter we have designed, implemented, and evaluated a PaaS live migration technique that provides minimal performance degradation and little to no service disruption. As part of future work, we will evaluate different combinations of rolling upgrades throughout the cloud stack, as well as migrations across WANs.

Chapter 9

Conclusion

This dissertation has laid the groundwork for a large scale hybrid cloud platform-as-a-service and has leveraged it to investigate multiple useful and high impact facets of cloud computing. First, it addresses the lack of tools to compare and contract different NoSQL offerings, in which there are over 120 at the time of writing this thesis. Second, it addresses the missing features of NoSQL datastores such as ACID transactions, and secondary index support. Third, this dissertation has provided a valid and novel use case for hybrid computing for emerging PaaS platforms, utilizing independent, disparate cloud systems for offline data analytics. Moreover, we extended this research by considering how certain cost models of cloud computing can be exploited to achieve low cost analytics. Lastly, we have pursued platform support of live migration for applications, with the capability to swap out the underlying hardware and software by leveraging the abstractions that cloud computing exposes.

In Chapter 3 we have presented AppScale, the first open source PaaS available to the research community, filling a need which was previously unfulfilled. In order to engender a vibrant developer community, we have emulated the Google App Engine APIs, allowing any of 1 million-plus applications to be also run on our private cloud platform. AppScale does not compete with App Engine, but it provides scale and flexibility for application developers and cloud researchers alike given the cluster resources it has been allocated. AppScale also addresses the problem of GAE lock-in since it enables applications to move to different public IaaS providers and to and between private clouds, without code modification. Finally, because the platform is open, developers can turn off the restrictions that GAE imposes on applications (e.g. limited libraries allowed, time restrictions on service use and request handling, sandboxed execution), albeit at the potential cost of scalability and system stability.

Our work in Chapter 4 has considered the benefits and trade offs of the cloud platform from the perspective of large scale data management. Our advances facilitate automatic deployment of distributed datastore technologies while providing application portability across them via a common datastore API. The portability layer of our implementation has allowed for the plugging in of datastores and distributed load of datastore requests across a distributed system. Applications can now be written using a common datastore API and easily migrated to another datastore without any modifications to their applications.

Because this layer provides a level of indirection between the application and the datastore, we have been able to show that we can use it to implement datastore-agnostic extensions to the underlying technologies. We prototype a limited form of distributed ACID transaction semantics (Chapter 4) and secondary index support (Chapter 5) by exploiting this layer. By doing so, we show that we can implement such support once and benefit numerous technologies for which it is absent – as opposed to reimplementing such support for each individually. For our limited transaction semantics, we require only that the underlying datastore technologies provide atomic row access and strong consistency across replicas. For secondary index support, we require only range query support. We show that we can efficiently add such support independent of the datastore technologies in this way by leveraging the extensible cloud platform that AppScale provides.

Given efficient and portable support for large scale, key-value data management, we next turn our attention to making use of such technology for analytics. Analytics is critical to both large and small businesses to make smart decisions about how best to satisfy customers and grow. In Chapter 6 we have described the first hybrid use case for PaaS systems. We have contributed two methods for synchronization across cloud platforms and have evaluated them using a range of metrics. Moreover, we have provided a high level query language for developers to run analytics jobs on top of their existing appli-

cations data without any requiring them to perform extraction-transformation-loading (ETL) manually.

In Chapter 7, we have considered the monetary aspects of using an emerging cost model exported by public cloud provider for large scale data analytics: market-based pricing. We have focused on and modeled the Amazon EC2 implementation of such pricing because of its mature and available implementation – but such a model can also be used on-premise for any organization with auditing requirements. Specifically, we have modeled a dataset of pricing of Amazon EC2 and have used it to make intelligent bidding decisions for resource prices for MapReduce jobs. Furthermore, we have found that this programming model, as it is implemented by the open source Hadoop system, suffers performance issues upon node failures due to intermediate data that must be regenerated.

We next investigated a key building block for hybrid cloud that an open and extensible platform like AppScale can provide. In particular, we have considered support for moving (migrating) an application between datastore, IaaS, and PaaS technologies while it is executing, without modifying the application code. Our results have shown that we are able to do so with little overhead, no lost updates, and with our limited transaction semantics support, for read heavy workloads.

Our contributions address how to facilitate portability of data-intensive applications across different cloud fabrics and internal storage systems. Our extensions to App-

Scale facilitate comparison of disparate storage systems without the need to learn the idiosyncrasies of any one datastore, while providing transaction support and secondary indexes. Moreover, our advances enable cross cloud synchronization of data that facilitates offline analytics and live application migration.

9.0.1 Impact

Our research artifacts include multiple publications in top conference venues including USENIX HotCloud, USENIX WebApps, and IEEE CLOUD. Additionally, we have combined and extended two of our papers [11, 21] for a journal publication in the Journal of GRID Computing [19]. We have also published a book chapter on the usage of AppScale in [55], which has been downloaded and referenced by users over 5,000 times.

We have released all of our code as open source under the a modified BSD license. Our releases began in April 2009 and we have released the system seven times since then. Our system has had over 10,000 downloads and over 4,000 starts in the last year, by users all over the world and by researchers at organizations including NASA and IBM. AppScale is the platform that Google representatives reference to address vendor lock-in concerns by potential and extant App Engine customers. Our user community (mailing list) which has over 320 members who consider the use of AppScale for both research and production deployments.

9.0.2 Future Work

The keys to the success and continued high impact of this dissertation work requires that we maintain and continue high-fidelity and compatibility with Google App Engine and that we support and integrate the updates made to the internal technologies that AppScale integrates (e.g. the NoSQL datastores) that are under development by third parties. To keep up with the release cycle of App Engine (monthly) we communicate with the Google App Engine team. We prioritize advances and changes to APIs based on application use. As part of future work we will continue to pursue such evolutions so that AppScale is as similar as possible to App Engine.

NoSQL datastores also have a fairly rapid release cycle, and updating the datastores requires a one-time effort. Most efforts are simple as the start up procedures and interfaces have not changed, but we've experienced underlying APIs changing causing porting time to increase. Yet, with developers leveraging AppScale, this porting effort is abstracted away because the GAE datastore layer is consistent.

Additional work is required to extend our index support to include scalable support for transactions. Our current limitation with pessimistic locking is simple yet restricts scalability and performance for batch updates of entities from different entity groups. Also as part of future work, we are investigating the trade-offs of different implementations of indexing in order to provide the high scale of NoSQL and with limited transaction semantic support.

Next, our application migration support currently works only for local area networks. As part of future work, we are extending this to the wide area. Since we rely on the Zookeeper open source technology for distributed coordination, and it has been used in the wide area, we are working on extending our contributions to provide limited forms of migration functionality in the wide area. We will evaluate both the limitations necessary and the trade offs required to scale and performance that such support across disparate clouds requires. We would also like to leverage our dissertation technologies to mirror data to facilitate such migration as well as to enable disaster recovery for applications.

With our cloud platform there are many longer term advances that are also possible. We find the switch over in the disaster recovery scenario can be trivial, but the fail-over mechanisms required are very application-specific. Additional advances can provide a more general solution for such scenarios.

Currently, our platform is ideal for a single application which can use the entire set of resources allocated to it. While AppScale can also support multiple applications, the resource utilization of individual applications can be improved. Different applications have different workloads (CPU versus datastore access), which leads to difficult scheduling and scaling problems. However, since AppScale can monitor and profile all of the activities of the cloud through API call interception, this feedback can be used to provide intelligent placement and elasticity support.

Finally, we have found that it is possible to combine our research on using spot instances with the automatic scaling of applications – since our application servers are stateless. The scaling out of our datastore is more challenging however, as different datastores behave differently when new nodes are added and removed. Moreover, different NoSQL technologies implement replication differently. Another potential long term extension of our work is support for automatic elasticity of the AppScale NoSQL plug-ins – in a way that is datastore agnostic (similar in spirit to our support for indexing and limited transaction semantics in this work).

Bibliography

[1] Accumulo. "http://accumulo.apache.org".

[2] Amazon Web Services. "http://aws.amazon.com/".

[3] AWS Disscussion Forum. http://tinyurl.com/2dzp734.

[4] Microsoft Azure Service Platform. "http://www.microsoft.com/azure/".

[5] M. Azure. Business Analytics, 2011. `http://www.windowsazure.com/en-us/home/tour/business-analytics/`.

[6] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *5th Biennial Conference for Innovative Data Systems Research*, 2011.

[7] J. Baker, C. Bond, J. Corbett, J. Furman, A. K. J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, January 2011.

[8] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[10] Brisk Datastax. "http://www.datastax.com".

[11] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura. An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *IEEE International Conference on Cloud Computing*, Jul. 2010.

[12] C. Bunch, J. Kupferman, and C. Krintz. Active Cloud DB: A RESTful Software-as-a-Service for Language Agnostic Access to Dis tributed Datastores. In *ICST International Conference on Cloud Computing*, 2010.

[13] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

[14] Cassandra. "http://cassandra.apache.org/".

[15] T. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live - An Engineering Perspective. In *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*, 2007.

[16] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. *Proceedings of 7th Symposium on Operating System Design and Implementation(OSDI)*, page 205218, 2006.

[17] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Symposium on Operating System Design and Implementation*, 2006.

[18] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26:65–74, March 1997.

[19] N. Chohan, C. Bunch, C. Krintz, and N. Canumalla. Cloud platform datastore support. *Journal of Grid Computing*, pages 1–19, 2012.

[20] N. Chohan, C. Bunch, C. Krintz, and Y. Nomura. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing*, July 2011.

[21] N. Chohan, C. Bunch, Y. Nomura, and C. Krintz. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing*, 2011.

[22] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open App Engine Application Development and Deployment. In *International Conference on Cloud Computing*, Oct. 2009.

[23] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See spot run: using spot instances for mapreduce workflows. In *USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[24] N. Chohan, A. Gupta, C. Bunch, K. Prakasam, and C. Krintz. Hybrid cloud support for large scale analytics and web processing. In *WebApps USENIX Conference*. USENIX Association, June 2012.

[25] N. Chohan, A. Gupta, C. Bunch, S. Sundaram, and C. Krintz. Infrastructure agnostic and datastore agnostic live migration of private cloud platforms. In *HotCloud USENIX Workshop*. USENIX Association, June 2012.

[26] Cloud Foundry. "http://cloudfoundry.com/".

[27] CloudExchange. http://cloudexchange.org/.

[28] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[29] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4:494–505, May 2011.

[30] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating System Design and Implementation(OSDI)*, pages 137–150, 2004.

[31] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Symposium on Operating System Principles*, 2007.

[32] ejabberd. "http://ejabberd.im".

[33] G. A. Engine. App Engine Transaction Semantics, 2010. `http://code.google.com/appengine/docs/python/datastore/transactions.html`.

[34] Eucalyptus home page. `http://eucalyptus.cs.ucsb.edu/`.

[35] Fantasm. "http://code.google.com/p/fantasm/".

[36] Google app engine blog. "http://googleappengine.blogspot.com/2011/05/year-ahead-for-google-app-engine.html".

[37] Google App Engine. `http://code.google.com/appengine/`.

[38] Google App Engine MapReduce. "http://code.google.com/p/appengine-mapreduce/".

[39] Google MapReduce Patent. http://tinyurl.com/yezbynq.

[40] J. V. Gurp and J. Bosch. On the implementation of finite state machines. In *in Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, IASTED/Acta*, pages 172–178. Press, 1999.

[41] Hadoop. `http://hadoop.apache.org/core/`.

[42] Hadoop Distributed File System. "http://hadoop.apache.org".

[43] Hadoop MapReduce. "http://hadoop.apache.org/".

[44] HAProxy. "http://haproxy.1wt.eu".

[45] HBase. `http://hadoop.apache.org/hbase`.

[46] HBase. "http://hadoop.apache.org/hbase/".

[47] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Networked Systems Design and Implementation*, 2011.

[48] Hive. Hive Query Processing Engine, 2010. `https://cwiki.apache.org/confluence/display/Hive/Home`.

[49] HyperTable. `http://www.hypertable.org/`.

[50] Hypertable. "http://hypertable.org".

[51] S. Ko, I. Hoque, B. Cho, and I. Gupta. On Availability of Intermediate Data in Cloud Computations. In *HotOS*, 2009.

[52] G. Kola, T. Kosar, and M. Livny. Faults in large distributed systems and what we can do about them. *Lecture Notes in Computer Science*, 3648:442–453, 2005.

[53] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *International Conference on Management of Data*, pages 579–590, 2010.

[54] C. Krintz, C. Bunch, and N. Chohan. AppScale: Open-Source Platform-A s-A-Service. Technical Report 2011-01, University of California, Santa Barbara, Jan. 2011.

[55] C. Krintz, C. Bunch, and N. Chohan. Appscale: Open-source platform-as-a-service. In L. Vaquero, J. Ciceres, and J. Hierro, editors, *Open Source Cloud Computing Systems: Practices and Paradigms*, chapter 9. IGI Global, Jan. 2012.

[56] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[57] Kernel based virtual machine. "http://www.linux-kvm.org/".

[58] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 1978.

[59] L. Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems*, 1998.

[60] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 101–110, New York, NY, USA, 2009. ACM.

[61] A. E. MapReduce. Amazon Elastic MapReduce. "http://aws.amazon.com/elasticmapreduce".

[62] E. Meijer and G. Bierman. A co-relational model of data for large shared data banks. *Commun. ACM*, 54(4):49–58, Apr. 2011.

[63] "memcached". "http://memcached.org".

[64] MemcacheDB. "http://memcachedb.org/".

[65] MongoDB. "http://mongodb.org/".

[66] Mongrel. "http://mongrel.rubyforge.org".

[67] R. Murthy and N. Jain. Talk at ICDE 2010. Hive–A Petabyte Scale Data Warehouse Using Hadoop., Mar. 2010.

[68] MySQL. "http://www.mysql.com".

[69] MySQL Cluster. "http://www.mysql.com/cluster".

[70] National Institute of Standards and Technology. "http://www.nist.gov/".

[71] Nginx. "http://www.nginx.net".

[72] Nosql databases. "http://nosql-databases.org".

[73] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *"9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)"*, May 2009.

[74] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *ACM SIGMOD*, 2008.

[75] OpenStack. "http://openstack.org".

[76] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Symposium on Operating System Design and Implementation*, 2010.

[77] Google App Engine Pipeline. "http://code.google.com/p/appengine-pipeline/".

[78] Protocol Buffers. Google's Data Interchange Format. "http://code.google.com/p/protobuf".

[79] RabbitMQ. "http://www.rabbitmq.com".

[80] Rackspace Inc. `http://www.rackspace.com/`.

[81] Redis. "http://redis.io".

[82] Ruby on Rails. "http://www.rubyonrails.org".

[83] Salesforce Customer Relationships Management (CRM) System. "http://www.salesforce.com/".

[84] SimpleDB. "http://aws.amazon.com/simpledb/".

[85] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, pages 1626–1629, 2009.

[86] TyphoonAE. "http://code.google.com/typhoonae".

[87] UnQL Query Language Unveiled by Counchbase and SQLite. "http://www.couchbase.com/press-releases/unql-query-language".

[88] Voldemort. "http://project-voldemort.com/".

[89] Z. Wei, G. Pierre, and C.-H. Chi. Scalable transactions for web applications in the cloud. In *Proceedings of the Euro-Par Conference*, Delft, The Netherlands, Aug. 2009. `http://www.globule.org/publi/STWAC_europar2009.html`.

[90] What is Google App Engine? "http://code.google.com/appengine/docs/whatisgoogleappengine.html

[91] T. Wood, H. A. Lagar-Cavilla, K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. Pipecloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 17:1–17:13, New York, NY, USA, 2011. ACM.

[92] XenSource. "http://www.xensource.com/".

[93] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

[94] ZooKeeper. "http://hadoop.apache.org/zookeeper".