A SEMINAR REPORT ON

"An Evaluation of Distributed Datastores Using The AppScale Cloud Platform"

SUBMITTED BY

Himanshu Ranjan Vaishnav

Roll No. 42065

SEMINAR GUIDE

Prof. Mrs S. S. Sonawani

Department of Computer Engineering

MAHARASHTRA INSTITUTE OF TECHNOLOGY

PUNE-411038

2012-2013

MAHARASHTRA ACADEMY OF ENGINEERING & EDUCATIONAL

RESEARCH'S

MAHARASHTRA INSTITUTE OF TECHNOLOGY

PUNE

DEPARTMENT OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that Mr HIMANSHU RANJAN VAISHNAV Roll No: 42065 of T. E. Computer successfully completed seminar in

"An Evaluation of Distributed Datastores Using the AppScale Cloud Platform"

to my satisfaction and submitted the same during the academic year 2012-2013 towards the partial fulfilment of degree of Bachelor of Engineering in Computer Engineering of Pune University under the Department of Computer Engineering , Maharashtra Institute of Technology, Pune.

Prof. Mrs S.S.Sonawani — **(Seminar guide)**

Prof. Mrs S.S. Paygude — **(Head of Computer Engineering Department)**

ACKNOWLEDGEMENT

I wish to express my heartfelt gratitude towards my seminar guide **Mrs. S.S. Sonawani**, who at each and every step in the better study and understanding of this seminar, contributed with her valuable time, guidance and provided excellent solutions for each problem arose.

I am also thankful to all the staff members who provided their kind support and encouragement during the preparation of this seminar.

I would also like to express my appreciation and thanks to all my friends who knowingly or unknowingly supported me with their interesting suggestions and comments, and feeling grateful for their assistance.

I would also like to express my gratitude to our Head of the Department **Mrs. S. S. Paygude** for her guidance and continuous support.

Once again, I want to express my true sense of gratitude to **Mrs. S.S. Sonawani** for her invaluable guidance and continuous support.

Date:

Himanshu Ranjan Vaishnav

Third Year

Computer Engineering

Roll No: 42065

Abstract

We present new cloud support that employs a single API the Datastore API from Google App Engine (GAE) to interface to different open source distributed database technologies. We employ this support to "plug in" these technologies to the API so that they can be used by web applications and services without modification. The system facilitates an empirical evaluation and comparison of these disparate systems by web software developers, and reduces the barrier to entry for their use by automating their configuration and deployment.

INDEX

Index of Figures

CHAPTER 1

INTRODUCTION

"**Cloud computing**" is the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet). The name comes from the use of a cloud-shaped symbol as an abstraction for the complex infrastructure it contains in system diagrams. Cloud computing entrusts remote services with a user's data, software and computation.

Technological advances have significantly driven down the cost of off-the-shelf compute power, storage, and network bandwidth. As a result, two key shifts have emerged. The cloud computing, is a service oriented computing model that uses operating systems, network, and storage virtualization to allow users and applications to control their resource usage dynamically through a simple programmatic interface, and for users to reason about the capability they will receive across the interface through well-defined Service Level Agreements (SLAs). Using this model, the high-technology sector has been able to make its proprietary computing and storage infrastructure available to the public (or internally via private clouds) at extreme scales. These systems provide per-user and per-application isolation and customization via a service interface that is typically implemented using high-level language technologies, well-defined APIs, and web services. A key component of most cloud platforms (e.g. Google App Engine (GAE), Microsoft Azure, Force.com), and infrastructures (e.g. Amazon Web Services (AWS) and Eucalyptus), is scalable and fault tolerant structured data management.

Cloud computing providers offer their services according to several fundamental models: Infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) where IaaS is the most basic and each higher model abstracts from the details of the lower models.

Cloud fabrics typically employ proprietary database (DB) technologies internally to manage system-wide state and to support the web services and applications that they make available to the public. In addition, in some cases (e.g. for AWS SimpleDB, GAE BigTable, Azure Table Storage), the cloud fabrics export these technologies via programmatic interfaces and overlays for use by user applications.

Distributed key-value datastores have become popular in recent years due to their simplicity, ability to scale within web applications and services usage models, and their ability to grow and shrink in response to demand. As a result of their success in nontrivial and highly visible cloud systems for web services, specifically BigTable within Google, Dynamo within Amazon, and Cassandra within Facebook, a wide variety of open-source variations of distributed key-value stores have emerged and are gaining widespread use.

AppScale is an open-source implementation of the Google App Engine cloud platform. It employs the Google Datastore API as a unifying API through which any datastore can be "plugged in". AppScale automates deployment and simplifies configuration of datastores that implement the API and facilitates their comparison and evaluation on end-to-end performance using real programs (Google App Engine applications). App-Scale currently supports HBase, Hypertable, Cassandra, Voldemort, MongoDB, MemcacheDB, Scalaris, and MySQL Cluster datastores. One limitation of this system is that only applications written in the languages that Google App Engine supports (currently Python and Java) execute over AppScale and thus are able to make use of the AppScale datastores.

To address these challenges, we investigate the efficacy of using a single well-defined API, the Google App Engine (GAE) Datastore API, as a universal interface to different cloud DBs. To enable this, we have developed the "glue" layer that links this API and different DBs so that we can "plug in" disparate systems easily. We also couple this component with tools that automate configuration and the distributed deployment of each DB. We implement this support via the AppScale cloud platform – an open-source cloud-platform implementation of the GAE APIs that executes using private and public virtualized cluster resources (such as Eucalyptus and Amazon EC2).

CHAPTER 2
GOOGLE APP ENGINE

Google App Engine (GAE) is a software development platform that facilitates the implementation of scalable Python and Java web applications. These applications respond to user requests on a web page using libraries and GAE services, access structured data in a non-relational, key-value datastore, and execute tasks in the background. The set of libraries and functionality that developers can integrate within the applications is restricted by Google, i.e. they are those "white-listed" as activities that Google is able to support securely and at scale. Google provides well defined APIs for each of the GAE services. When a user uploads her GAE application to Google resources (made available via "MyApp".appspot.com) the APIs connect to proprietary, scalable, and highly available implementations of each service.

Google offers this platform-as-a-service (PaaS) free of charge. However, applications must consume resources below a set of fixed quotas and limits (API calls per minute and per day, bandwidth and CPU used, disk space, request response and task duration, mail sent) or the request and/or application is terminated. Users can pay for additional bandwidth, CPU hours, disk, and mail. The key mechanism for facilitating scale in GAE applications is the GAE Datastore. The GAE Datastore API provides the following primitives:

• Put (k, v): Add key k and value v to table; creating a table if needed

• Get (k): Return value associated with key k

• Delete (k): Remove key k and its value

• Query (q): Perform query q using the Google Query Language (GQL) on a single table, returning a list of values

• Count (t): For a given query, returns the size of the list of values returned

The Google cloud implements this API via BigTable, and adds support for management of transactional indexed records via MegaStore. BigTable is a strongly consistent key-value datastore, the data format and layout of which can be dynamically controlled by the application. BigTable is optimized for reads and indexing, can be accessed using arbitrary strings, and is optimized for key-level synchronization and transaction support. BigTable employs the distributed and proprietary Google File System (GFS) for transparent data replication for fault tolerance. It internally uses a master/slave relationship with the master node containing only metadata and coordinating read/write requests to slave nodes, who

contain the replicated data in the system. Slave node failures can be tolerated, but the master node is a single point of failure in the system. Users query data using GQL and the platform serializes data for communication between the front-end and BigTable using Google Protocol Buffers. Google provides other forms of data management via the Memcache API for caching of non-persistent data using an interface similar to the Datastore API, and the Blobstore API which enables users to store/retrieve large files (up to 50MB). To better understand the functionality, behaviour, and performance of the GAE cloud and its applications, to enable research in the next-generation of PaaS systems, and to provide a pathway toward GAE without concern for "lock-in" and privacy of code/data, we developed AppScale. AppScale is a robust, open source implementation of the GAE APIs that executes over private virtualized cluster resources and cloud infrastructures including AWS and Eucalyptus. Existing GAE applications execute over AppScale without modification. We detail the AppScale system in and overview its components in the subsections that follow. That work established support across HBase and Hypertable, and this work expands that to include support for MySQL, Cassandra, Voldemort, MongoDB, and MemcacheDB. We focus on a novel implementation of the Google Datastore API within AppScale, support for integrating and automating deployment of the open source databases, and an empirical evaluation of these disparate DBs.

CHAPTER 3
APPSCALE

3.1 AppScale: A Cloud Platform

AppScale is a robust, open source implementation of the Google App Engine APIs that executes over private virtualized cluster resources and cloud infrastructures including Amazon Web Services and Eucalyptus. Users can execute their existing Google App Engine applications over AppScale without modification. Describes the design and implementation of AppScale. We summarize the key components of AppScale that impact our description and implementation of Active Cloud DB. AppScale is an extension of the non-scalable software development kit that Google makes available for testing and debugging applications. This component is called the Active Cloud DB 3 AppServer within AppScale. AppScale integrates open-source datastore systems as well as new software components that facilitate configuration, one-button deployment, and distributed system support for scalable, multi-application, multi-user cloud operation. The AppServer decouples the APIs from their non-scalable SDK implementations and replaces them distributed and scalable versions – for either Python or Java. This allows Google App Engine applications to be written in either language. AppScale implements front-ends for both languages in this way. Google App Engine applications write to a key-value datastore using the following Google Datastore API functions:

– Put (k, v): Add key k and value v to table; creating a table if needed

– Get (k): Return value associated with key k

– Delete (k): Remove key k and its value

– Query (q): Perform query q using the Google query language (GQL) on a single table, returning a list of values

– Count (t): For a given query, returns the size of the list of values returned

Google App Engine applications employ this API to save and retrieve data. The AppServer (and Google App Engine SDK) encodes each request using a Google Protocol Buffer [11]. Protocol Buffers facilitate fast encoding and decoding times and provide a highly compact encoding. As a result, they are much more efficient for data serialization than other approaches. The AppServer sends and receives Protocol Buffers to a Protocol Buffer Server

in AppScale over encrypted sockets. All details about the use of Protocol Buffers and the back-end datastore (in Google or in AppScale) is abstracted away from Google App Engine applications using this API. The AppScale Protocol Buffer Server implements all of the libraries necessary to interact with each of the datastores that are plugged into AppScale. Since this server interacts with all front-ends, it must be very fast and scalable so as to not negatively impact end-to-end application performance. AppScale places a Protocol Buffer Server on each node to which datastore reads and writes can be sent (i.e. the datastore entry points) by applications. For master-slave datastores, the server runs on the master node. For peer-to-peer datastores, the server runs on all nodes.

AppScale currently implements eight popular open-source datastore technologies. They are Cassandra, HBase, Hypertable, MemcacheDB, MongoDB, Voldemort, Scalaris, and MySQL Cluster (with a key-value data layout). Each of these technologies vary in their maturity, eventual/strong consistency, performance, fault tolerance support, implementation and interface languages, topologies, and data layout, among other characteristics. Presents the details on each of these datastores. Active Cloud DB and our caching support are datastore-agnostic. These extensions thus work for all of the AppScale datastores (current and future) and for Google App Engine applications deployed to Google's resources. In our evaluation, we choose two representative datastores to collect empirical results for: Cassandra and MemcacheDB.

We provide a brief overview of each. *Cassandra.* Developed and released as open source by Facebook in 2008, Cassandra is a hybrid between Google's BigTable and Amazon's Dynamo. It incorporates the flexible column layout from the former and the peer-to-peer node topology from the 4 Chris Bunch et al. latter. Its peer-to-peer layout allows the system to avoid having a single point of failure as well as be resilient to network partitions. Users specify the level of consistency required on each read or write operation. This allows read and write requests to be sent to any node in the system, allowing for greater throughput. However, this results in data being "eventually-consistent." Specifically, this means that a write to one node will take some time to propagate throughout the system and that application designers need to keep this in mind. Cassandra exposes a Thrift API through which applications can interact with in many popular programming languages. *MemcacheDB.* Developed and released as open source by Steve Chu in 2007, MemcacheDB is a modification of the popular open source distributed caching service memcached. Building upon the memcached API, MemcacheDB adds replication and persistence using Berkeley DB

as a backing store. MemcacheDB provides a key-value datastore with a master-slave node layout. Reads can be directed to any node in the system, while writes can only be directed to the master node. This allows for strong data consistency but also multiple points of access for read requests. As MemcacheDB is a master-slave datastore, the master node in the system is the single point of failure. MemcacheDB can use any library available for memcached, allowing for native access via many programming languages.

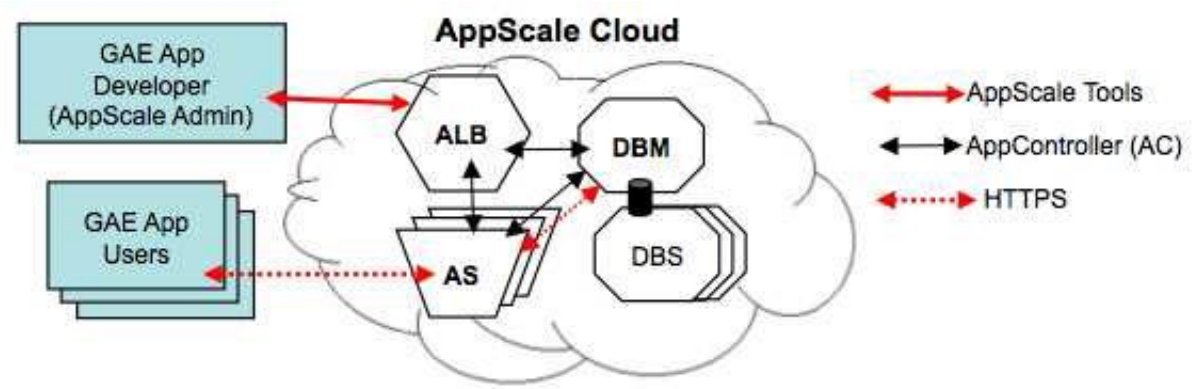

Fig 1: Depiction of an AppScale Deployment

Figure 1 depicts a typical AppScale deployment. System administrators configure and deploy an AppScale cloud using a set of command-line tools. The tools enable users to configure/start/stop a cloud, upload/remove a GAE application, and retrieve statistics on resource availability and use. AppScale also implements a web-based interface to the tools for manipulating an extant cloud and viewing the status of the cloud. For this study, we consider a static cloud configuration – the size is specified at cloud instantiation and remains the same until the cloud is destroyed. The primary components of AppScale are the AppController, the AppServer, the AppLoadBalancer, and the contribution of this paper, pluggable database support.

AppController

The AppController is a SOAP server that runs on every node in the system. An AppController is aware of the layout of the cloud and starts up the necessary services in the required order. The AppController is able to customize a node with any component and

service available in AppScale. If the node instantiates a database that requires configuration files, the AppController writes the files in the correct locations. The AppController also uses **iptables** to allow only AppScale nodes to access component ports. The first AppController in the system plays a special role in the cloud, sending a heartbeat message every ten seconds to all nodes in the system and recording whether or not the node is alive. It also profiles the other nodes, recording metrics such as CPU and memory usage to employ for dynamically scaling the cloud.

AppServer

The AppServer component is the open source GAE SDK that users employ to test, debug, and generate DB indices for their application prior to uploading it to the Google cloud. We replace the nonscalable API implementations in the SDK with efficient, distributed, and open source systems. We use available Python and Java libraries for many of the services (e.g. Memcache, Images), the local system for others (Mail), and hand-built tools for others (Tasks). For the Datastore API, we modify the SDK to open a socket and forward the protocol buffer to a remote server (the SDK simply puts/gets protocol buffers to/from a file). We remove all quota and programming restrictions that Google places in GAE applications that execute within its cloud and have added new APIs, including one that provides applications with access to Hadoop Streaming for user-defined MapReduce computation.

AppLoadBalancer

To partition the access by users to a GAE application web page across multiple servers, we employ the AppLoadBalancer. This component is a Ruby on Rails application that selectively chooses which AppServer should receive incoming traffic, and uses the **nginx** web server to serve static content.

It also authenticates the user (providing support for the Google Users API). Multiple copies of this service run and are load balanced by HAProxy. The current implementation does not use the AppLoadBalancer as a reverse proxy, as is commonly done, to prevent it from becoming a single point of failure. Thus, users will see the URL change in their address bar. If the AppLoadBalancer fails, AppServer execution and user-AppServer interaction is unaffected.

3.1.1 The AppScale Internals

Figure 2 shows the layout of AppScale. At the highest level there is a load balancer, which takes incoming requests from users and routes them to an application server, copies of which may be on a number of remote hosts. The application layer is supported by a wide range of services that eases application development by precluding the need for reimplementation of common tasks. At the lowest level there is a system-wide database service which provides persistent storage onto the disk.

AppScale automatically provisions services and does so in a fault tolerant and scalable manner. A key automated process is the setting up of a datastore. AppScale supports many databases that are implemented in many different languages and have different designs. These databases are: Cassandra, HBase, Hypertable, MongoDB, MemcacheDB, Scalaris, SimpleDB, MySQL Cluster, and Voldemort.

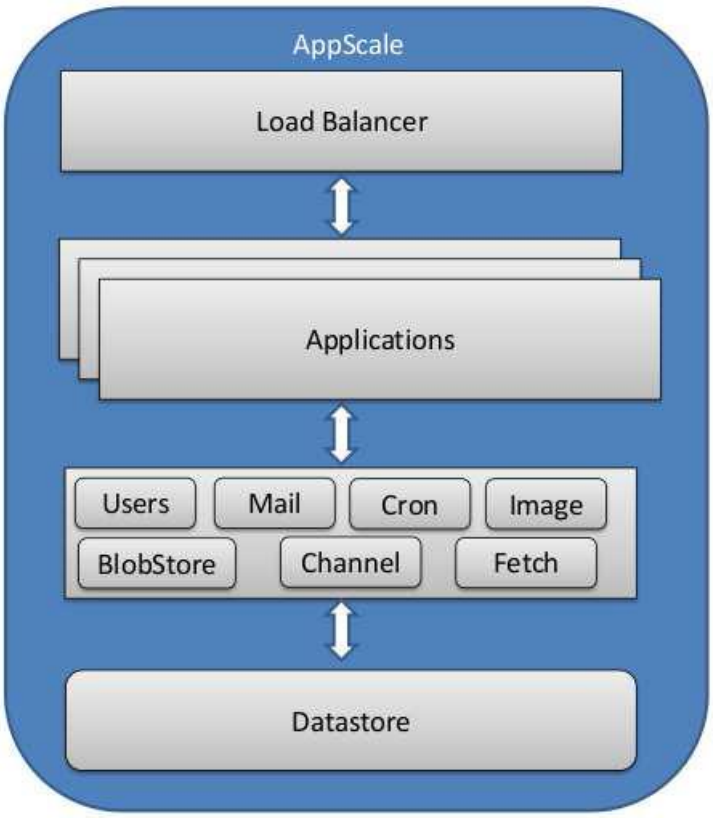

Figure 2: The multi-tiered approach within AppScale consists of a load balancer, multiple application servers, several services to support the different APIs, and a datastore for persistent data and state (not all APIs and services are shown here).

3.1.2 Google App Engine APIs

Foremost, AppScale provides implementations for the Google App Engine APIs. These APIs provide several scalable services which can be leveraged by a Google App Engine application. The Google App Engine APIs are Blobstore, Channel, Datastore, Images, Memcache, Namespaces, TaskQueue, Users, URL Fetch, and XMPP. Google App Engine provides an overview of the APIs and their functionality at http://code.google.com/appengine/docs/. We emulate this functionality within AppScale using open source software systems, tools, and services, as well as new components that we have written. We implement all of the APIs assuming distributed execution so that we are able to provide isolation, scalability, elasticity, and fault-tolerance for cloud platform applications. This is in sharp contrast to the Google App Engine SDKs that are provided for testing and debugging using a single machine. We describe the features of the API that AppScale currently does not support in Section 4.

Blobstore API

The Blobstore API enables users to store large entities of text or binary data. The upload is performed by an application using a form post with a file name. This large file is not constrained by the 1MB limitation of Blobs stored using the Datastore API.

Channel API

The Channel API allows applications to push messages from the application server to a client's browser. The program must register an application token with the service, which in turn is used by a JavaScript library to connect to a messaging service. The Channel API is useful for creating applications such as online chat or a real time multi-player game. The implementation in AppScale has the ability to send messages to multiple receivers who are registered using the same application token. Google App Engine imposes the restriction that there may be only one receiver per sending channel. AppScale uses Ejabberd and Strophejs in its implementation.

Datastore API

The Datastore API allows for the persistence of data. The API has many useful function calls that all ultimately map to either a PUT, GET, DELETE, QUERY. The Google Query Language (GQL) is similar to and is a subset of SQL; fundamentally, it lacks relational operations such as JOIN and MERGE. AppScale employs in-memory filters for GQL statements while Google App Engine maps them to range queries on the datastore.

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 [9]. 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-COMMIT isolation.

AppScale’s implementation of transactions are database agnostic and rely on ZooKeeper for locking entity groups. ZooKeeper stores information about which entities are currently locked and our transactional middleware employs a garbage collector for locks to make sure that locks are not held for a long periods of time. We store journal information within the database and implement rollback of failed transactions.

Developers use entity groups and transactions for applications that require atomic updates across multiple keys (in an entity group). Yet, transaction semantics introduce overhead, stress the underlying database implementation, and limit scalability (relative to non-transactional operations). To reduce this overhead, we encourage program developers to avoid creating large entity groups and to shard (and spread out) global state across many entity groups.

Images API

The Images API facilitates programmatic manipulation of images. Popular functions in this API include the ability to generate thumbnails, rotation, and composition, convert formats, and cropping images. The API in AppScale uses the Google App Engine SDK implementation.

Memcache API

The Memcache API permits applications to store their frequently used data in a distributed memory grid. 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.

Namespace API

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.

Task Queue API

The Google App Engine platform lacks the ability to do threading or computations within a request greater than 30 seconds. The TaskQueue API facilitates computation (tasks) in the background by enqueuing tasks via a web request. Task durations are restricted in Google App Engine to 30 seconds (as of GAE 1.4.0). AppScale implements the same restriction but this value can be increased with minor code modification. A potential workaround is to chain multiple task queues, where each task does up to 30 seconds of work and then queues another task to pick up where the current one left off.

Users API

The Users API provides authentication for web applications through the use of cookies. While GAE provides access to users who have accounts with Google, AppScale requires users to sign up an account through the AppScale portal URL. In addition, the API distinguishes between regular and administrator users.

URL Fetch API

The URL Fetch API enables an application the ability to do POST and GET requests on remote resources. REST APIs from third parties can be accessed using this API. The implementation in AppScale is identical to what is provided by the Google App Engine SDK.

XMPP API

The XMPP API presents the ability to receive and send messages to users with a valid XMPP account. Google App Engine leverages the Google Talk infrastructure while AppScale is able to use a scalable implementation in Ejabberd.

3.1.3 Other AppScale APIs

In addition to the Google App Engine APIs, AppScale implements other APIs which are not supported by Google App Engine. These APIs are of use to cloud platform applications developers for emerging application domains such as data analytics and high-performance computing.

MapReduce Streaming API

Within Google App Engine itself, there currently is no method by which long running, arbitrary computation can be performed by applications. AppScale supports such computation via Hadoop Streaming (http://wiki.apache.org/hadoop/HadoopStreaming). Through Hadoop Streaming, users can specify a Mapper and Reducer program that can run under the MapReduce programming paradigm. Fault-tolerance and data replication is automatically handled by the Hadoop Streaming framework, and AppScale exposes a simple API that interfaces to Hadoop Streaming.

This API is:

• *putMRInput(data, inputLoc)*: Given a string ”data” and a Hadoop file system location “inputLoc”, creates a file on the Hadoop file system named “inputLoc”

• *runMRJob(mapper, reducer, inputLoc, outputLoc, config)*: Given the relative paths to a mapper and reducer file (relative to the main Python file being run), run a HadoopMapReduce Streaming job. Input is fed to the program via the HDFS file named “inputLoc”, and output is fed to the HDFS file named “outputLoc”. If a hash is passed as ”config”, the key / value pairs are passed as configuration options to Hadoop Streaming.

• *getMROutput(outputLoc)*: Given a Hadoop file system location “outputLoc”, returns a string with the contents of the named file.

• *writeTempFile(suffix, data)*: Writes a file to /tmp on the local machine with the contents data. Is useful for passing a file with nodes to exclude from MapReduce jobs.

• *getAllIPs()*: Returns an array of all the IPs in the AppScale cloud. Is useful for excluding or including nodes based on some user-defined application logic.

• *getNumOfNodes()*: Returns an integer with the number of nodes in the AppScale cloud. Is useful for determining at MapReduce run time, how many Map tasks and Reduce tasks should be run for optimal performance (recommended value is 1 Map task per node).

• *getMRLogs(outputLoc)*: Returns a string with the MapReduce job log for the job whose output is located at outputLoc. Data is returned as a combination of XML and key / value pairs, in the standard Hadoop format.

Currently,Mappers and Reducers can be written by developers in Python, Ruby, or Perl. In addition, we and others can easily extend AppScale with support for other programming languages.

To use this API, the cloud administrator must select HBase or Hypertable as the platform datastore so that the Hadoop Distributed File System (HDFS) is automatically deployed (both of these datastores use HDFS for their implementations). A complete sample application that uses theMapReduce Streaming API is bundled with the AppScale Tools and is called *mapreduce*.

EC2 API

Google App Engine does not provide anymechanisms by which users can interactwith AmazonWeb Services natively, so to fill this void, AppScale provides an Amazon EC2 API. Users can use this API to spawn virtual machines and manage them entirely through an AppScale web service. The main functions this API provides mirror those of the EC2 command line tools:

• *ec2 run instances(options)*: Spawns virtual machines with the specified options (interpreted as command-line arguments).

• *ec2 describe instances()*: Returns the status of all machines owned by the current user.

• *ec2 terminate instances(options)*: Terminates virtualmachineswith the specified options (interpreted as commandline arguments).

• *ec2 add keypair(options)*: Creates a new SSH key-pair that can be used to log in to virtual machines that are spawned with this key-pair's name.

• *ec2 delete keypair(options)*: Deletes the named SSH key-pair from the underlying cloud infrastructure.

• *ec2 describe availability zones(options)*: In Eucalyptus, this returns information relating to the number of virtual machines available to be spawned by users.

• *ec2 describe images()*: Returns information about the virtual machine images, ramdisks, and kernels that are available for use in the system.

• *ec2 reboot instances(options)*: Reboots virtual machines with the specified options (interpreted as commandline arguments).

Other functions are available to handle the storage and retrieval of EC2 credentials: their usage can be found in the *ec2demo* application that is bundled with the AppScale Tools. Alternatively, the KOALA Cloud Manager project provides a Google App Engine application that implements several AWS APIs (as of writing, EC2, EBS, S3, and ELB are supported), and can run over Google App Engine as well as over AppScale. It can be found at: http://code.google.com/p/koalacloud/.

3.1.2 Advanced Deployment Strategies

AppScale is also capable of performing more advanced placement of components and services. This is controlled through the YAML cloud platform configuration file that we described in the previous subsection. In that previous example, the "ips.yaml" file identifies one node as the "controller". This role consists of a load balancer and Database Master service. The roles of the other nodes, called "servers", consist of Database Slaves and application servers. This is the default AppScale placement strategy and we illustrate it in Figure 3.

Users can define their own placement strategies instead of using the default. The possible roles are:

• Load balancer: The service that routes users to their Google App Engine applications. It also hosts a status page that reports on the state of all machines in the currently running AppScale deployment.

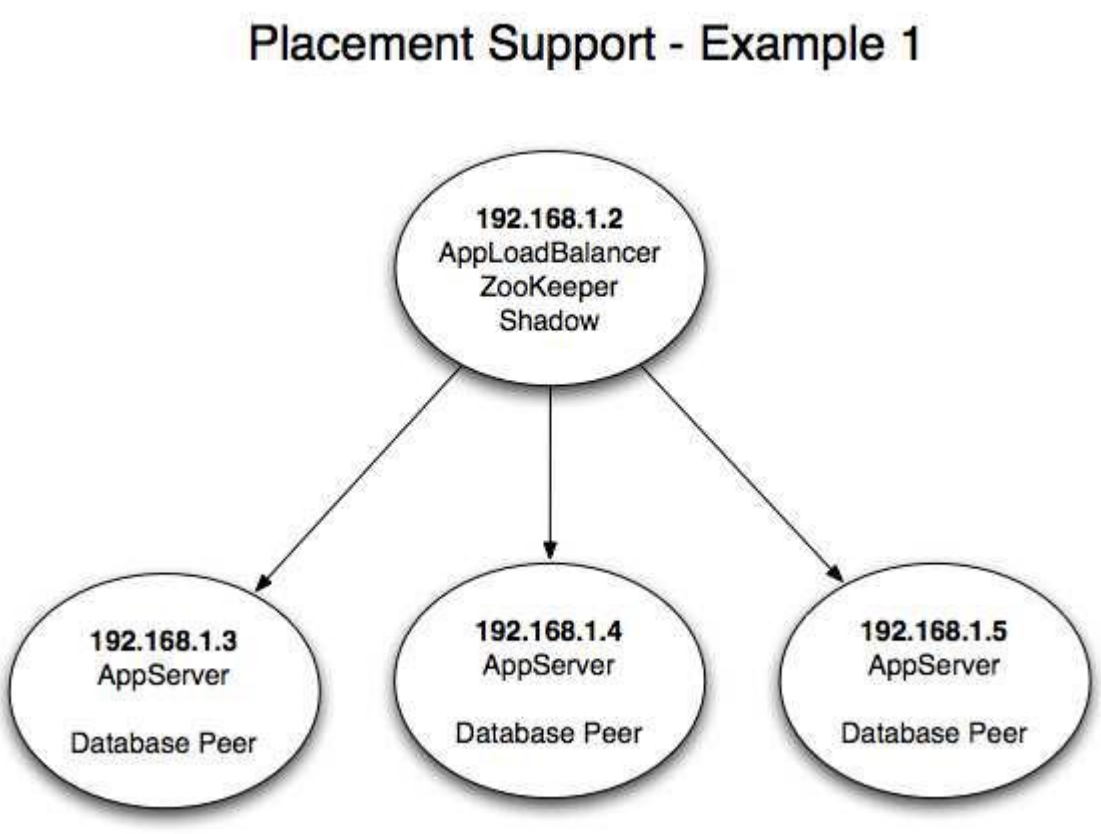

Figure 3: The default placement strategy within AppScale. Here, one node (the controller) handles load balancing and a single database server, while the other nodes (the servers) deploy application servers as well as database servers.

• App Engine: Our modified version of the non-scalable Google App Engine SDKs that adds in the ability to store data to and retrieve data from databases that support the Google Datastore API, as well as our implementations of the various other APIs that App Engine offers.

• Database: Runs all the services needed to host the chosen database.

• Login: The primary machine that is used to route users to their Google App Engine applications. Note that this differs from the load balancer - many machines can run load balancers and route users to their applications, but only one machine is reported to the cloud administrator when running “appscale-run-instances” and “appscaleupload-app”.

• ZooKeeper: Hosts metadata needed to implement database-agnostic transaction support.

• Shadow: Queries the other nodes in the system to record their state and ensure that they are still running all the services they are supposed to be running.

• Open: The machine runs nothing by default, but the Shadow machine can dynamically take over this node to use it as needed. We show how to implement a placement strategy via example. In this example, we wish to have a single machine for load balancing, for execution of the database metadata server via ZooKeeper, and for routing users to their applications.

These functions are commonly used together, so the role “master” can be used to signify all of them in a single line of configuration. On two other nodes, we wish to host applications, while on the final node, we wish to host our database. The configuration file for this deployment strategy using a virtualized private cluster (because we specify IP addresses) looks like the following:

```
:master: 192.168.1.2
:appengine:
- 192.168.1.3
- 192.168.1.4
:database:
- 192.168.1.5
```

In another example deployment, we use a cloud infrastructure with one node hosting the “master” role, two nodes hosting web applications and database instances, and a final node routing users to their applications. The configuration file for this looks like the following:

```
---
:master: node-1
:appengine:
- node-2
- node-3
:database:
- node-2
- node-3
:login:
- node-4
```

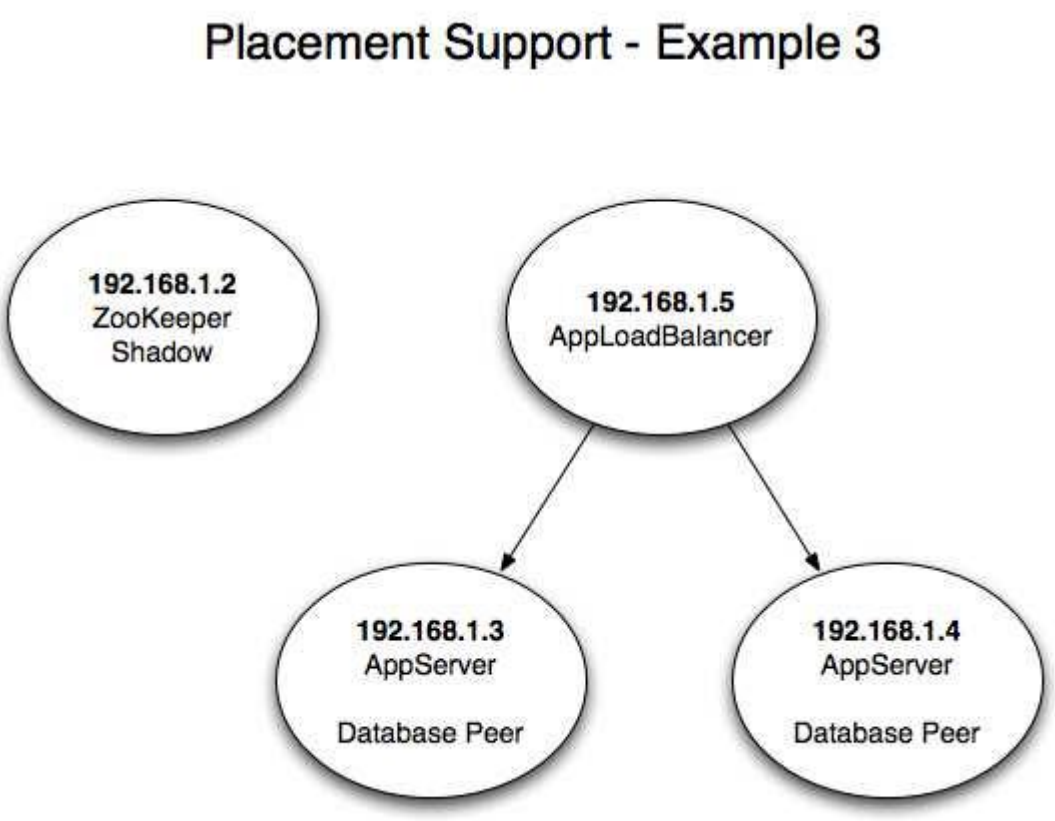

Figure 4: illustrates this placement strategy, where the machines have been assigned IP addresses beginning with
192.168.1.2.

Figure 4: A more complicated placement strategy, in which a single node handles load balancing, two nodes handle application servers and database nodes, and a final node handles transaction management and system monitoring. For databases that employ a master / slave relationship, the first machine specified in the “database” section is the master, while the others are slaves. For peer-to-peer databases, all nodes are peers, but the machine specified first in the “database” section is the first peer to be started.

3.2 Using AppScale

In this section, we discuss the critical services that make up an AppScale deployment and how they serve the applications that execute over the platform and employ the AppScale APIs.

3.2.1 Database Services

An AppScale deployment ensures that a database is started automatically and done so for a variable number of machines. Each database can be started and stopped by the AppController in a repeatable fashion, and can do so for a variety of database technologies. Each database has an implementation of the AppScale DB API of PUT, GET, DELETE, and QUERY. There are three services which use this interface.

First is a Protocol Buffer Server (PBServer), which receives Protocol Buffers from an App Engine application's database calls (Protocol Buffers are Google's internal data serialization method). The PBServer handles requests from different applications and implements the transactional middleware used to provide database agnostic transactions. Second, is the User/App Server (UAServer). The UAServer uses the AppScale DB API to access a user's and applications table and provides SOAP functions for simple RPC calls. All persistent data for the Users API is stored here. Data about an application, its ips, ports, admins, etc are stored here. Moreover, XMPP accounts and Channel

API session information is both retrieved and committed to the UAServer (all considered application data). Lastly, blobs which are uploaded using the blobstore service are stored in the database in 1MB chunks. A Blobstore Server is running on all nodes which are hosting applications. POST of blob files are forwarded to this service and once the file has completed uploading, the service redirects the user back to the application's blobstore upload success handler.

3.2.2 Monitoring Services

AppScale employs a piece of monitoring software called Monitr. It uses collectd to get standard metric information such as CPU, disk utilization, memory usage, and network access for all nodes in the deployment. A link to Monitr is present on the status page hosted by the load balancer. Furthermore, AppScale employs heartbeat checks to verify that critical components are functional. If a component is not responsive, or the process has died, AppScale will revive the component.

3.2.3 Neptune

Neptune is a domain specific language supported by AppScale. With Neptune, users can dictate workflows for certain languages, platforms, and scientific applications. These include but are not limited to X10, R, MPI, and Hadoop MapReduce. Users of Neptune also have the capability to control execution of jobs between different clouds (hybrid cloud execution).

3.3 App Engine Capability Overview

Fig 5: App Engine Capability

CHAPTER 4

Appscale: Platform-As-A-Service

As AppScale is a cloud platform, i.e. a platform-as-a-service (PaaS) cloud fabric that executes over cluster resources. The cluster resources underlying AppScale can be managed with or without virtualization (e.g. Xen, KVM) or via popular cloud infrastructures including Amazon EC2 and Eucalyptus.

The AppScale platform virtualizes, abstracts, and multiplexes cloud and system services across multiple applications, enabling *write-one, run-anywhere (WORA)* program development for the cloud. In addition to simplifying application development and deployment using cloud systems and modern distributed computing technologies, App-Scale brings popular public cloud fabrics to "on-premise", or private, clusters. To enable this, we emulate key cloud layers from the commercial sector – (i) to engender a user community, (ii) to gain access to and to study real applications, and (iii) to investigate the potential implementations of, and extensions to, public cloud systems using open-source technologies.

Since AppScale provides a software layer between applications and the implementations of cloud APIs and services, we are also able to employ existing cloud services for the implementations. As such, AppScale provides a *hybrid* cloud platform – a programming system through which an application can access services from different clouds (a mix of public and private) at different times or concurrently. Such support can be used by developers to move data between clouds, e.g., for disaster recovery, fault tolerance, data backups, to reduce public cloud costs (to use lower-cost alternatives), and to "fall out" from a private cloud with limited resources to a public cloud on-demand.

CHAPTER 5

APPSCALE: Software-as-a-Service

Distributed key-value datastores have become popular in recent years due to their simplicity, ability to scale within web applications and services usage models, and their ability to grow and shrink in response to demand. We address the problem of how to facilitate access to different Datastores via any programming language and framework using a database-agnostic interface to key-value datastores. To enable this, we present the design and implementation of a Sofware-as-a-Service (SaaS) component called Active CloudDB (in the spirit of Ruby's ActiveRecord). Active Cloud DB is a Google App Engine application that executes over Google App Engine or AppScale that provides the glue between an application and scalable cloud database systems (AppScale's datastores and Google's BigTable). Active Cloud DB is easy to integrate into language and framework front-ends and also provides the functionality to move data between distributed datastores without manual configuration, manual deployment, or knowledge about the datastore implementation.

We employ this Software-as-a-Service to implement support for various web-based language frameworks, including Ruby over the Sinatra and Rails web frameworks as well as Python over the Django and web.py web frameworks. We evaluate the performance of Active Cloud DB using the Cassandra and MemcacheDB datastores to investigate the performance differences of the primitive datastore operations. Finally, we extend Active Cloud DB with a simple data caching scheme mechanism that also can be used by any Google App Engine application, and evaluate its impact.

CHAPTER 6

APPSCALE DISTRIBUTED DATABASE SUPPORT

In order for AppServers to communicate with the backend datastore the data must be serialized via Google Protocol Buffers. Requests are sent to a Protocol Buffer Server (PBServer) which implements the Datastore API. Upon receiving a request, the PBServer extracts it and then makes the appropriate API call for the currently running datastore. This is not always a simple task, as there is not always a one-to-one correlation between Datastore API calls and what the underlying datastore supports. For example, some Datastore API calls create a table to store data in, and some of the Datastores encountered here only allow for a single table to be used in the system.

AppScale automates the deployment of distributed database technologies. To enable this, we release App-Scale as an operating system image that users can instantiate directly over virtualization technologies without any manual configuration or deployment. This provides functionality similar to that of Hadoop-on-Demand (HOD) for each datastore. AppScale generates all of the necessary configuration files, command line arguments, and environment variables automatically.

Furthermore, AppScale starts up the necessary services for each datastore in the correct order. This typically requires some amount of coordination amongst the App-Controllers in the system, as even though some datastores run in a peer-to-peer configuration, one peer in the system must always be started first and allows for the other peers to easily locate each other in the system and manage data. In this work, we employ this support to “plug in” seven open source distributed database systems that we selected because of their maturity, widespread use, documentation, and design choices for distribution, scale, and fault tolerance. We also include MySQL Cluster, which, unlike the others, is not a key-value store but instead is a relational database. We include it to show the extensibility of our framework and to compare its use as a key-value store, with the others.

6.1 Cassandra

Facebook engineers designed, implemented, and released the Cassandra datastore as open source in 2008. Cassandra offers a hybrid approach between the proprietary datastore implementations of Google BigTable and Amazon Dynamo. It takes the flexible column layout offered by the former and combines it with the peer-to-peer layout of the latter in the hopes of gaining greater scalability over other open source solutions. Cassandra is currently in use internally at Facebook, Twitter, Cisco, among other web companies.

Cassandra is eventually consistent. In this model, the system propagates data written to any node to all other nodes in the system. These multiple entry points improve read performance, response time, and facilitates high availability even in the face of network partitions. However, there is a period of time during which the state of the data is inconsistent across the nodes. Although algorithms are employed by the system to ensure that propagation is as fast as possible, two programs that access the same data may see different results. Eventual consistency cannot be tolerated by some applications; however, for many web services and applications, it is not only tolerated but is a popular trade-off for the increased scalability it enables.

Cassandra is written in the Java programming language and exposes its API through the Thrift software framework. Thrift enables different programming languages to communicate efficiently and share data through remote procedure calls. Cassandra internally does not use a query language, but instead supports range queries. Range queries allow users to batch primitive operations and simplify query programming. Cassandra requires that table configurations be specified statically. As a result, we are forced in AppScale to employ a single table for multiple GAE applications.

6.2 HBase

Developed and released by PowerSet as open source in 2007, HBase became an official Hadoop subproject with the goal of providing an open source version of Google's BigTable. HBase employs a master-slave distributed architecture. The master stores only metadata and redirects clients to a slave for access to the actual data. Clients cache the location of the key range and send all subsequent reads/writes directly to the corresponding slave. HBase also

provides flexible column support, allowing users to define new columns on-the-fly. Currently, HBase is in use by PowerSet, Streamy, and others.

HBase is written primarily in Java, with a small portion of the code base in C. HBase exposes its API using Thrift and provides a shell through which users can directly manipulate the database using the HBase Query Language (HQL). For users accessing the Thrift API, HBase exports a Scanner interface with which developers traverse the database while maintaining a pointer to their current location. This functionality is useful when multiple items are retrieved a "page" at a time.

HBase is deployed over the Hadoop Distributed File System (HDFS). HDFS is written in Java and for each node in the cluster, it runs on top of the local host's operating system file system (e.g. ext2, ext3 or ext4 for Linux). HDFS employs a master-slave architecture within which the master node runs a NameNode daemon, responsible for file access and namespace management. The slave nodes run a DataNode daemon, responsible for the management of storage on its respective node. Data is stored in blocks (the default size is 64 MB) and replicated throughout the cluster automatically. Reads are directed to the nearest replica to minimize latency and bandwidth usage. Like Google's BigTable over GFS, by running over a distributed file system, HBase achieves fault tolerance through file system replication and implements strong consistency.

6.3 Hypertable

Hypertable was developed by Zvents in 2007 and later released as open source with the same goal as HBase: to provide an open source version of Google's BigTable. Hypertable employs a master-slave architecture with metadata on the master and data on the slaves. All client requests initially go through the master and subsequent client request go directly to the slave. Currently, Hypertable's largest user is the Chinese search provider Baidu which reports running Hypertable over 120 nodes and reading in roughly 500 GB of data per day.

Hypertable is written in C++ to facilitate greater control of memory management (caching, reuse, reclamation, etc.). Hypertable exposes its API using Thrift and provides a shell with which users can interactively query the datastore directly using the Hypertext Query Language (HQL). It provides the ability to create and modify tables in a manner analogous to that of SQL as well as the ability to load data via files or standard input. Hypertable also provides a Scanner interface to Thrift clients.

Like HBase, Hypertable also runs over HDFS to leverage the automatic data replication and fault tolerance that it provides. Hypertable splits up tables into sets of contiguous row ranges and delegates each set to a RangeServer. The RangeServer communicates with a DFS Broker to enable Hypertable to run over various distributed file systems. RangeServers also share access to a small amount of metadata, which is stored in a system known as Hyperspace. Hyperspace acts similarly to Google's Chubby, a highly available locking and naming service that stores very small files.

6.4 MemcacheDB

Open source developer Steve Chu modified the popular caching framework *memcached* to add data persistence and replication. He released the resulting system as MemcacheDB in 2007. MemcacheDB employs a master-slave approach for data access, with which clients can read from any node in the system but can only write to the master node. This ensures consistency while allowing for multiple read entry points.

MemcacheDB is written in C and uses Berkeley DB for data persistence and replication. Clients access the database using any existing memcached library. Clients perform queries via the memcached get multi-function to request multiple keys at once. Since the system does not track of all the items in the cache, a query that retrieves all data is not possible: developers who require this functionality must manually add and maintain a special key that stores all of the keys in use.

MemcacheDB runs with a single master node and multiple replica nodes. User's instantiate the MemcacheDB service on the master node and then invoke replica nodes, identifying the location of the master. Since the master does not have a configuration file specifying which nodes are replicas in the system, any node can potentially join the system as a replica. This flexibility can present a security hole, as a malicious user can run their own MemcacheDB replica and have it connect to the master node in an attempt to acquire its data. Clients can employ Linux **iptables** or other firewalling mechanisms to restrict access to MemcacheDB master and slave nodes.

6.5 MongoDB

MongoDB was developed and released as open source in 2008 by 10gen. MongoDB was designed to provide both the speed and scalability of key-value datastores as well as the ability to customize queries for the specific structure of the data. MongoDB is a document-oriented database that is optimized for domains where data can be manifested like documents, e.g. template and legal documents, among others. MongoDB offers three replication styles: master-slave replication, a "replica-pair" style, and a limited form of master-master replication. We consider master-slave replication in this work. For this architecture, all clients read and write by accessing the master, ensuring consistency in the system. Commercially, MongoDB is used by SourceForge, Github, Electronic Arts, and others.

MongoDB is written in C++. Users can access MongoDB via language bindings that are available for many popular languages. MongoDB provides an interactive shell with its own query language. Queries are performed using hashtable-like access. The system exposes a cursor that clients can use to traverse the data in a similar fashion to the HBase and Hypertable Scanner interface.

MongoDB is deployed over a cluster of machines in a manner similar to that of MemcacheDB. No configuration files are used and once the master node is running, an administrator invokes the slave nodes, identifying the location of the master. MongoDB suffers from the similar security problem of unauthenticated slaves attaching to a master; administrators can use **iptables** or other measures to restrict such access to authorized machines.

6.6 Voldemort

Developed by and currently in use internally at LinkedIn, Voldemort emulates Amazon Dynamo and combines it with caching. It was released as open source in 2009. Voldemort provides eventual consistency; reads or writes can be performed at any node by clients. There is a short duration during which the view of the data across the system is inconsistent. Fetches on a key may result in Voldemort returning multiple values with their version number, as opposed to a single key (the latest version) as is done in Cassandra. It is up to the application to decide which value is valid. Voldemort persists data using BerkeleyDB (or other backends) and allows the developer to specify the replication factor for each chunk of the distributed hash table employed for distribution. This entails that the developer also partition the key space manually.

Voldemort is written in Java and exposes its API via Thrift; there are native bindings to high-level languages as well that employ serialization via Google Protocol Buffers. A shell is also provided for interactive queries.

6.7 MySQL

MySQL is a well-known relational database. We employ it within the AppScale DB framework as a key-value datastore. We store a list of columns and the value for it in the "value" column. This gives us a new key-value datastore that provides replication and fault tolerance.

There are many MySQL distribution models available; we employ MySQL Cluster in this paper. The node that performs instantiation and monitoring is referred to as the management node, while the other nodes which store the actual data are referred to as data nodes. A third type of node is the API node, which stores and retrieves data from data nodes. Application clients using MySQL Cluster can make requests to any of the API nodes. It provides concurrent access to the system while providing strong consistency using two phase commit amongst replicas. Additionally, the system is fault tolerant with the master node only required for initial configuration and cluster monitoring.

MySQL is written in C and C++. As it is a mature product, it has API drivers available in most programming languages. A shell is provided for interactive queries written in SQL, and programs using the native drivers can also use the same query language to interact with the database.

CHAPTER 7
EVALUATION

We next employ AppScale and our Datastore API extensions to evaluate how well the different databases support the API. We first overview our experimental methodology and then present our results.

7.1 Methodology

To evaluate the different datastores, we use Active Cloud DB, a Google App Engine application that exposes a REST API to the Datastore API's primitive operations. We then measure the end-to-end response time (round-trip time to/from the datastore through the AppServer). For all experiments, we fill a table in each database with 1,000 items and perform multiple put, get, delete, no-op (the AppServer responds to the client without accessing the database) and query operations in order (1,000 puts, gets, deletes, and no-ops, then 100 queries) per thread. A query operation returns the contents of a given table, which in these experiments returns all 1,000 items. We consider (i) light load: one thread; (ii) medium load: three concurrent threads; and (iii) heavy load: nine concurrent threads. We repeat each experiment five times and compute the average and standard deviation. As a point of reference, a single thread in a two-node configuration (more on this below) exercises the system at approximately 25 requests per second.

We execute this application in an AppScale cloud. We consider different static cloud configurations of size 1, 2, 4, 13, and 96 nodes. In the 1, 2, and 4 node deployments, each node is a Xen guestVM that executes with 2 virtual processors, 10GB of disk (maximum), and 4GB of memory. In the 13 and 96 node deployments, each node executes with 1 virtual processor and 1GB of memory. The 1-node configuration implements an AppLoadBalancer (ALB), AppServer (AS), and an AppDB Master (DBM). For multi-node configurations, the first node implements an ALB and DBM and the remaining nodes implement an AS and DBS. For clouds with greater than 4 nodes, we consider heavy load only. For these experiments, AppScale employs Hadoop 0.20.0, HBase 0.20.3, Hypertable 0.9.2.5, MySQL Cluster 6.3.20, Cassandra 0.5.0, Voldemort 0.51, MongoDB 1.0.0, and MemcacheDB 1.2.1-Beta.

We also vary consistency configurations using Cassandra and Voldemort, to evaluate its impact on performance. For our Cassandra deployment, a read succeeds if it accesses data from a single node, and a write succeeds automatically. Writes go to a buffer, which a separate process then writes to the datastore. If conflicts arise, the version with the newer timestamp is accepted and duplicated as needed. This therefore presents a highly inconsistent but potentially faster performing datastore.

For our Voldemort deployment, we have chosen to employ a stronger consistency configuration. Data is replicated three times, and all three nodes with each piece of data must be in agreement about any old values to read or new values to write. We therefore expect this strongly consistent datastore to perform slower than its inconsistent counterpart.

7.2 Experimental Results

We next present results for end-to-end web application response time for individual API operations using the different datastore systems. Response time includes the round-trip time between the client and database, and includes AppServer, DBM or DBS, and database access. We consider puts, gets, deletes, no-ops, and queries using the workloads described above. We present only a subset of the results, due to space constraints, but select representative datasets. In most experiments the standard deviation is very low (a few milliseconds maximum).

Figure 6 shows three graphs with the response time for put, get, and delete primitive operations, respectively, under medium load (the light load results are similar). We consider clouds of 1, 2, and 4 nodes in these graphs. We omit data for HBase, Hypertable, and MySQL Cluster for the single node configuration since they require a dedicated node for the master in AppScale, i.e., they do not suport a 1-node deployment.

The x-axis identifies the database and the y-axis shows the average operation time in seconds. The graphs show that master-slave datastores provide better response times over the peer-to-peer datastores. As the number of nodes increase in the system, response times decrease as expected. All databases perform similarly and improve as the number of nodes increases, but the peer-to-peer databases and MySQL perform the best at four nodes. This is due to the increased number of entry points to the database, allowing for non-blocking reads to be done

in parallel. The put and delete operations perform similarly and gets (reads) experience the best performance.

We next present the performance of the query operation under different loads using the 4-node configuration, in Figure 7. Each bar is the query response time in seconds for light, medium, and heavy loads respectively. Query is significantly slower than the other primitive operations since it retrieves all items from the database as opposed to working on a single item. Response time for queries degrades as load increases. The large number of threads and operations of the heavy load degrades performance significantly for some databases. MySQL Cluster scales the best for these load levels, which is believed to be due to its much greater maturity compared to the other database offerings seen here. Hypertable outperforms MySQL Cluster for light load and ranks second across databases for medium and heavy load.

We next consider heavy load for the put, get, and query operation (delete performs similarly to put) when we increase the number of nodes past 4. We present the 4-node query data from Figure 6 in the right graph and the other results in Figure 7. The x-axis is response time in seconds – note that the scale for query is different since the query operation takes significantly longer to execute than put or get. We omit data for MySQL Cluster

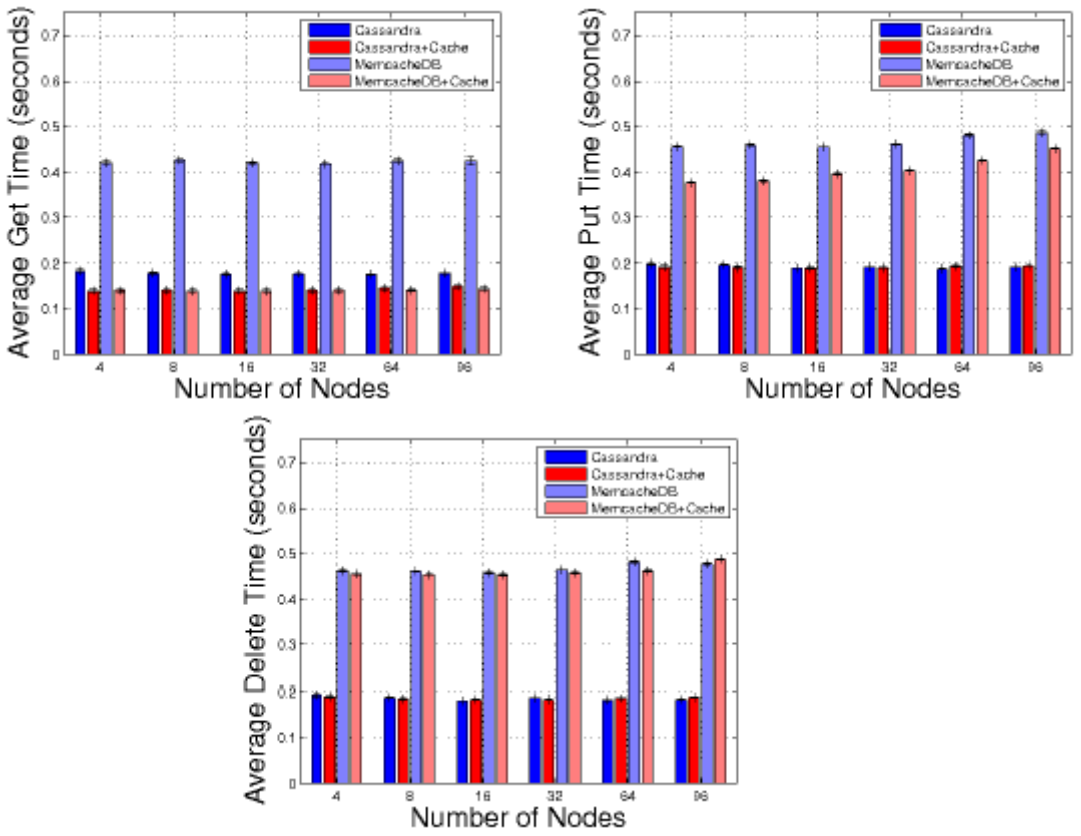

Fig. 6. Average round-trip time for get (left), put (right), and delete (bottom) operations under a load of 9 concurrent threads for a range of different AppScale cloud sizes (node count).

For 96 nodes because it does not support greater than 48 nodes (and we were unable to get it to perform with stability for larger than 13 nodes). We omit Voldemort data for 13 and 96 nodes because the database returns errors when the number of nodes exceeds 4. We are currently working with the Voldemort team on this issue.

As we increase the number of nodes under heavy load, the response time increases slightly for put and get (and delete). This is likely due to the additional overhead that is required for managing the larger cluster system in support of ensuring consistency and providing replication. For queries, increasing the number of nodes from 4 to 13 improves query response time in most cases. This improvement is significant for Cassandra and MemcacheDB. Increasing the node count past 13 for this load level provides very little if any improvement for all databases. MySQL performs significantly better than the others for this load level and the 4 and 13 node configurations for query. It performs similarly for puts, gets, and deletes as Cassandra and Voldemort, and significantly outperforms the other datastores.

These experiments also revealed a race condition in our query implementation. Specifically, dynamism restrictions for three datastores (Cassandra, Voldemort, and MemcacheDB) require that we employ a single table for data and simulate multiple tables by storing a special key containing a list of the keys currently in the given table. We use this "meta-key" whenever queries are performed, however updates to it are not atomic. In our experiments, we find that a race occurs very infrequently and does not significantly impact performance. We plan to release the fix for this issue in an upcoming release of AppScale. Furthermore, our current implementation only allows for database accesses to be done via the master node for HBase and Hypertable, whereas only the first access needs to contact the master node. We therefore are working on changing AppScale accordingly, in order to improve performance and throughput.

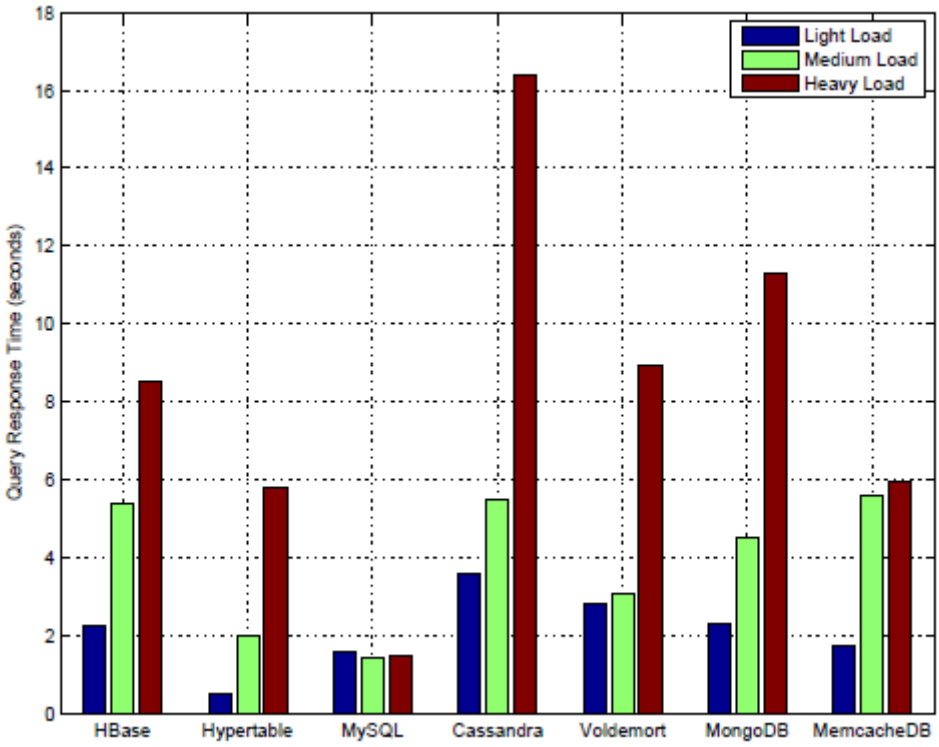

Fig. 7. Average time for the query operation under different loads
Using the four node configuration.

Finally, we investigate the performance of our application using the Google cloud with the BigTable backend. The average no-op time (round-trip time) between our cloud and Google is 240ms with a standard deviation of 4ms; using AppScale on our local cluster, average no-op time is 78ms with a standard deviation of 2ms. Private clouds enable lower round-trip times which may prove important for latency-sensitive applications. Our results for get, put, delete, and query (without the noop/ round-trip time) using the Google cloud (with an unknown number of nodes) is similar to those for App-Scale/MySQL with four or more nodes (the difference is statistically insignificant). Moreover, the Google DB access times show higher variance.

7.3 RELATED WORK

To our knowledge, this is the first project to offer a unified API together with a framework for automatically deploying and evaluating open source distributed database systems for use with real web applications

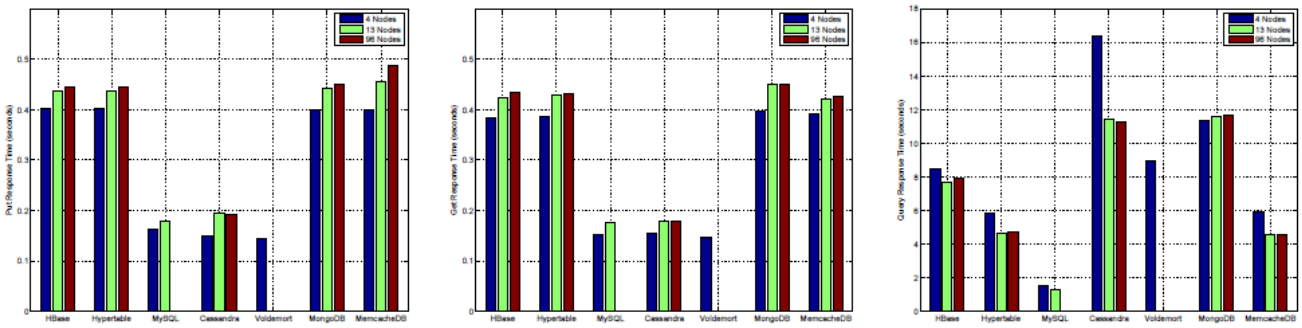

Fig. 8. Average time for the put (left), get (middle), and query (right) under heavy load as we increase the number of nodes in the cloud configuration. Note that the y-axis scale differs for query (right graph). MySQL Cluster does not support greater than 48 nodes; Voldemort returns errors when the number of nodes exceeds 4. We are currently working with the Voldemort team on the latter issue. The delete data (omitted) is very similar to that of put.

And services. In addition, we offer a description of the characteristics and deployment methodologies of seven popular DB systems. The web offers many sites that describe these systems (e.g. [5], [19]) but there is no single peer-reviewed published study that describes their characteristics and deployment process.

The one related effort is the Yahoo! Cloud Serving Benchmark (YCSB). YCSB [8]. The authors present a benchmarking system for cloud serving systems. YCSB enables different database technologies to be compared using a single system as AppScale does. In contrast to AppScale, YCSB does not automate configuration and deployment of datastores, focuses on transaction-level access as opposed to end-to-end application performance for DB accesses, and is not yet available as open source. YCSB currently supports non-replicated DB configurations, for HBase, Cassandra, PNUTS (Yahoo's internal key-value store), and Sharded MySQL.

7.4 Limitations

There are several current limitations in AppScale that developers should be aware of before deploying AppScale and their applications. The list of limitations is as follows; for each, we provide possible work-around and/or our plan for addressing them.

• Persistence: If AppScale is on a virtualized platform, terminating instances can cause a loss of data. To address this issue we are implementing a bulk uploader and downloader similar to the one included with Google App Engine.

• Blobstore Max File Size: 100MB. This value is configurable within the code and we will allow for the setting of this value in a configuration file.

• Datastore: AppScale does not index data but rather does filtering of queries in-memory. As the size of the database gets larger, users may see a decrease in performance for queries. We are working on an indexing system which can create indexes yet stay compliant with the ACID semantics needed for transactions.

• Task Queue: Tasks which are enqueued are not fault tolerant, nor does AppScale handle delayed workers. The Task Queue will be reimplemented using an open source distributed queue technology.

• Mail: Only the administrator is allowed to send mail, and reception of mail is not implemented. Extensions to this API are on our road map.

• OAuth API is not implemented in AppScale. We have this API on our road map.

• AppServers currently follow a"deploy on all nodes" method and do not scale up or down as needed. We will allow for pluggable scheduling modules which will dictate dynamic placement in the future.

• Ubuntu is the only distribution supported. This is to limit testing allowing for faster release cycles. AppScale is portable to other distributions and in the past has successfully run on Red Hat.

• The Java AppServer does not have support for the Blobstore or Channel APIs. Google has not released the source code of the Java App Engine server, making it difficult to add new services. For each new feature we must decompile, add the feature, and recompile. We are working with Google to have this source code released.

• Some datastores do not have a method of retrieving the entire table to run a query. They must use a set of special keys which track all keys in the table. These datastores therefore have the added overhead of always accessing the special keys. These datastores are: MemcacheDB, Voldemort, SimpleDB, and Scalaris. We recommend using either Cassandra, HBase, Hypertable, or MySQL Cluster because they do not suffer from this limitation.

CONCLUSIONS AND FUTURE SCOPE

We present an open source implementation of the Google App Engine (GAE) Datastore API within a cloud platform called AppScale. The implementation unifies access to a wide range of open source distributed database technologies and automates their configuration and deployment. However, each database differs in the degree to which it implements the API, which we analyse herein. We describe this implementation and use the platform to empirically evaluate each of the databases we consider. In addition, we articulate our experience using and integrating each database into AppScale. Our system (including all databases) is available as a virtual machine image at http://appscale.cs.ucsb.edu.

8.1 AppScale Feature Highlights

AppScale initially started out to provide full compatibility with Google App Engine and do it for a wide selection of datastores. Since its initial release, it has expanded to provide its own APIs as well. Below are some key features of AppScale.

8.1.1 App Engine Portability

Google App Engine was available for use in 2008. AppScale quickly followed suit and released a compatible platform in 2009. As App Engine has added new APIs, languages, and features, AppScale has provided scalable open source implementations to provide developers more choices and options to avoid code and data lock-in.

8.1.2 More Choices

Run your application on physical hardware, virtualization, or in any cloud that can run an Ubuntu image. Additionally, pick your favorite datastore as your backend for your applications. AppScale lets you compare and contrast different datastores with your application providing the benchmark.

8.1.3 Neptune Language

Neptune is a domain specific language (DSL) that automates configuration and deployment of existing HPC software via cloud computing platforms. Neptune is integrated into

AppScale and extend the platform with support for user-level and automated placement of cloud services and HPC components. Such platform integration of Neptune facilitates hybrid-cloud application execution as well as portability across different cloud providers.

8.1.4 MapReduce

Hadoop is a software framework that supports data-intensive distributed applications. It enables applications to work with thousands of nodes and petabytes of data. AppScale brings the Hadoop API to your App Engine applications for greater computational flexibility.

8.1.5 Fault Tolerance

AppScale can survive faults within a cluster. It has replication support for applications and their data. Processes are monitored and restored for cases where they have terminated or when they are misusing resources.

8.1.6 and More

AppScale has other features such as the ability to do database-as-a-service; automatically deploying a datastore of your choice at scale and then exposing it with a REST interface. Additionally, AppScale has user management access control allowing the cloud administrator to grant permissions to cloud APIs and other capabilities.

8.2 Future Directions

We began investigating AppScale as a web platform for executing Google App Engine applications without modification, in a scalable, efficient manner. We are continuously working to improve the design and implementation of AppScale to address the limitations above and to increase transparency, performance and scale. In addition, we have been focused on the automation and control of, as well as more coordinated interaction (for scheduling, automatic service-level agreement negotiation, elasticity, etc.) with cloud infrastructures (e.g. Amazon EC2 and Eucalyptus). As part of future work, we are extending AppScale with new services for large-scale data analytics as well as data and computation intensive tasks. In addition, we are investigating a wide range of hybrid cloud technologies to

facilitate application development and deployment that is cloud-agnostic. This entails new services, APIs, scheduling, placement, and optimization techniques. Finally, we are investigating new language support, performance profiling, and debugging support for cloud applications as well as the integration of mobile device use within the platform. In summary, we have presented AppScale, its design, implementation, and use. We have detailed the APIs that AppScale supports and how they relate to Google App Engine. Moreover, we have described extensions which are specific to AppScale that expand the applicability of the platform, while retaining the simplicity and automation that clouds offer. We encourage readers to try AppScale either using the readily available AMI image or the Xen Image on our hosting site http://code.google.com/p/appscale/. We appreciate all feedback and suggestions that users can provide through our community mailing list http://groups.google.com/group/appscalecommunity.

REFERENCES

[1] BerkeleyDB. http://www.oracle.com/technology/products/berkeley-db/index.html.

[2] C. Bunch, J. Kupferman, and C. Krintz. Active Cloud DB: A Database-Agnostic HTTP API to Key-Value Datastores. In *UCSB CS Technical Report 2010-07*.

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

[4] Cassandra. http://cassandra.apache.org/.

[5] R. Cattell. Datastore Survey – work in progress, Apr. 2010. http://cattell.net/datastores/Datastores.pdf.

[6] 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.

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

[8] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB (to appear), June 2010. http://research.yahoo.com/node/3202.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *ACM SOSP*, 2003.

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

[11] Hadoop on demand. http://hadoop.apache.org/common/docs/r0.17.1/hod.html.

[12] HAProxy. http://haproxy.1wt.eu.

[13] HBase. http://hadoop.apache.org/hbase/.

[14] Hypertable. http://hypertable.org.

[15] D. Judd. Hypertable Talk at NoSQL meetup in San Francisco, CA. June 2009., June 2009.

[16] MemcacheDB. http://memcachedb.org/.

[17] MongoDB. http://mongodb.org/.

[18] Nginx. http://www.nginx.net.

[19] K. North. Databases in the Cloud: Elysian Fields or Briar Patch?, Aug. 2009. http://www.drdobbs.com/java/218900502?pgno=1.

[20] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *IEEE International Symposium on Cluster Computing and the Grid*, 2009. http://open.eucalyptus.com/documents/ccgrid2009.pdf.

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

[22] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation, Apr. 2007. Facebook White Paper.

[23] Voldemort. http://project-voldemort.com/.

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

[25] Bittorrent: http://www.bittorrent.com

[26] YAML. http://www.yaml.org.

[27] Zookeeper. http://http://hadoop.apache.org/zookeeper/.