Key-Value Datastores Comparison in AppScale

Chris Bunch Navraj Chohan Chandra Krintz Jovan Chohan
Jonathan Kupferman Puneet Lakhina Yiming Li Yoshihide Nomura[†]
Computer Science Department [†]Software Innovation Laboratory
University of California, Santa Barbara Fujitsu Labs Ltd., Japan
February 17, 2010 UCSB Tech Report 2010-03

Abstract

We present a simple framework that employs a single API – the Datastore API from the Google App Engine cloud computing platform – to interface to different open source distributed database technologies in use today. We use the framework to "plug in" different databases to the API so that they can be used by web applications and services without modification. The system facilitates empirical evaluation and comparison of these disparate systems by web software developers, and reduces the barrier to entry for the use of such systems by automating their configuration and deployment.

1 Introduction

Highly available, scalable, relational database management systems (RDBMS), such as those offered by MySQL, Oracle, PostgreSQL, Sybase, and others, have traditionally been employed by the commercial sector for mission-critical, enterprise business applications. These applications commonly require fault tolerant, high-throughput transaction processing and implement complex queries that access data across multiple tables. Administrators of these RDBMSs configure fixed schemas for the tables prior to deployment and use by applications. Static schemas enable query optimization and static error checking of structured query language (SQL) code, among other benefits. RDBMs vary in cost, storage model, distribution and replication strategy, scale, and performance. Although highly effective for the enterprise application domain, recent advances in distributed database technologies have shown that for web applications, simpler datastores may be sufficient and potentially more scalable.

One reason behind this trend is that web applications and services exhibit significantly different data access behavior than that of enterprise software. For example, web-based software (e.g. distributed systems based on the model-view-controller pattern written in high-level languages) is typically read-heavy and string-oriented, employs finer-grain data accesses (e.g. individual keys rather than entire tables and individual tables rather than multiple tables, per query), use a small subset of the SQL language, and rarely make use of the full feature sets that relational systems offer. As a result, many new database technologies have emerged in support of web applications.

For example, Google and Amazon both employ distributed databases for use by web services that implement simple key-value datastores, the data format and layout of which can be dynamically controlled by the application. These systems are optimized for reads and indexing, can be accessed using arbitrary strings, and are optimized for key-level synchronization and transaction support. Such systems are highly scalable and very efficient for the application domain that they target. Google's offering, BigTable [6], provides strong consistency and high availability, but does not replicate the actual data. It employs the distributed and proprietary Google File System [8] (GFS) for transparent replication beneath Bigtable. Amazon implements Dynamo [2] which replicates data, is eventually consistent, and highly available. Both Google (via Google

App Engine (GAE)) and Amazon (via Amazon Web Services (AWS)) make their proprietary implementations of these datastores available through high-level abstractions within their cloud computing platform (GAE) and infrastructure (AWS). Cloud computing in the commercial sector as offered by GAE, AWS, and others, provides pay-per-use rental of remote resources (CPU, network, disk, etc.) based on service-level agreements, and is currently employed primarily for web-based applications.

When the descriptions of BigTable and Dynamo were published, a number of open source offerings emerged that emulated the functionality of these systems. Such offerings include HBase [13], Hypertable [14], and Cassandra [5]. These distributed database technologies aim to provide scalable storage along with fast indexing, a variety of data models, query support that is simple and specific to the data model, and fault-tolerance. In addition, like BigTable and Dynamo, they offer different consistency and fault tolerance policies. In addition, they vary in the programming language employed for their implementation, the distributed computing topology (master-slave and peer-to-peer), and the programming interfaces they offer. As a result, the use and deployment of any of these systems imposes a significant learning curve on web application developers, making it challenging to compare and evaluate these systems for their applications.

To address this challenge, we present a simple framework with which different database technologies can be employed by web applications easily and automatically. We enable this through an open-source, distributed implementation of the Google App Engine (GAE) cloud computing platform. This system, called AppScale, facilitates "plugging in" different databases for the implementation of the Google Datastore API, used by GAE for its BigTable implementation. That is, we employ this API as a universal substrate through which web applications access data stored in a wide variety of different database back-ends.

In this paper, we describe the design and implementation of this support and the ways in which AppScale eases the installation, configuration, and deployment of these disparate software systems. We use the terms datastore and database interchangeably. We consider seven popular open source distributed database systems including MySQL cluster. We provide an overview of each of these technologies, Google App Engine and its Datastore API, and the AppScale platform. We detail our implementation of this API and our experience with using it as a unifying abstraction for each of these database systems. We employ AppScale with these extensions to empirically compare and evaluate each in the context of activities typical of web services. The systems vary in terms of ease of use and integration, as well as in performance and scalability. Our results measure of how well each implements the Google Datastore API out of the box (i.e. without optimization). We begin with an introduction to the database systems that we consider.

2 Open Source Distributed Database Technologies

In this section, we overview seven distributed database technologies. We selected these systems based on their maturity, widespread use, documentation, and different design choices made for distribution, scale, and fault tolerance. We also include MySQL Cluster, which unlike the others, is a relational database management system. We include it to show the extensibility of our Datastore API implementation and so that we can compare it to the others using the AppScale framework. The technologies we consider are Cassandra, HBase, Hypertable, MemcacheDB [16], MongoDB [17], Voldemort [23], and MySQL Cluster [18].

2.1 Cassandra

Facebook engineers designed, implemented, and released the Cassandra datastore as open source [5] 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. The Apache Incubator Project page states that Cassandra has been employed on a production cluster (> 150 machines) to manage over 100TB of data [5].

Cassandra is eventually consistent. In this model, the system propogates data written on any node to all other nodes in the system. These multiple entry points improves read performance, response time, and faciliates 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 users that access the same web site may see different results. Eventual consistency cannot be tolerated by some applications; however, for many web services and application, it is not only tolerated but a popular trade-off for the increased scalabilty it enables.

Cassandra is written in the Java programming language and exposes its API through the Thrift software framework [22]. 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.

While Cassandra allows users to statically specify the number and layout of tables used in the system, we were unable utilize this system since it failed when specifying more than one table. This seems to be an issue with the version used for testing. Regardless, tables cannot be created dynamically, as a result users must store all of their data within a single table. Facebook and other commercial entities use a proprietary extension of Cassandra internally. We find that there is a significant gap between the open source offering and the proprietary versions, in terms of reported performance, scalability, and usability.

2.2 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 [13]. 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 send all reads / writes to the master. This model ensures data consistency across the system. HBase also provides flexible column support, allowing users to define new columns on-the-fly. Currently, HBase is in use by PowerSet, Streamy, and StumbleUpon, amongst other commercial entities.

HBase is written in the Java programming language (although some of the code base, e.g. for data compression, are written 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 scanner functionality is useful when multiple items are retrieved a "page" at a time. Although any database can perform scanning, HBase provides developers with direct control over the scanning process which can be used to improve performance.

HBase is deployed over the Hadoop Distributed File System (HDFS) [9]. 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.

2.3 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, data on the slaves, and all client requests going though the single entry point of the master to ensure data consistency. 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 [15].

In contrast to Cassandra and HBase, Hypertable is written in C++ in an attempt to enable better performance. The designers of Hypertable claim that using C++ gives them greater control of memory management (caching, reuse, reclamation, etc.) [14]. Hypertable exposes its API using Thrift and provides a shell with which users can access the datastore directly using the Hypertext Query Language (HQL). Hypertable also provides a Scanner interface to 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 [4], a highly available locking and naming service that stores very small files.

2.4 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 [16]. MemcacheDB employs a master-slave approach for accessing data, with which clients can read data from any node in the system but can only write to the master node. This keeps the data consistent while allowing for multiple read entry points. MemcacheDB is currently in use at the link-sharing site Reddit.

MemcacheDB uses a combination of memcached and Berkeley DB for data persistence. Both are written in the C programming language. Clients access the database using any existing memcached library. Using such libraries, clients can create a shell to access the database directly using any programming language that implements the libraries. Clients perform queries on the database via the memcached `get_multi` function which request multiple keys at once. Since the system does not track of all the items in the cache, a true query that retrieves all the 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 slave nodes. Therefore, users instantiate the MemcacheDB service on the master node and then invoke replica nodes with a command line argument that identifies 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 slave. 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.

2.5 MongoDB

MongoDB was developed and released as open source in 2008 by 10gen [17]. 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, like CouchDB, since clients can specialize their queries based on document type, 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. Therefore, the data is consistent across the system. Commercially, MongoDB is used by SourceForge, github, EA, and others.

Like Hypertable, MongoDB is written in C++ so that memory management of the system can be controlled more precisely by developers. Users can access MongoDB via language bindings which have been implemented for many popular languages. MongoDB provides an interactive shell with its own unnamed query language. Queries are performed in a manner similar to JSON, using a hashtable-like format. The

system exposes a cursor that identifies the current point in the database that is being queried. Clients can use this support 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 using a command line that identifies the location of the server. MongoDB suffers from the similar security problem of unauthenticated slaves attaching to a master; administratrators can use iptables or other measures to restrict such access to authorized machines.

2.6 Voldemort

Developed and currently in use internally at LinkedIn, Voldemort emulates Amazon Dynamo and combines it with the caching framework of memcached [23]. 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 Cassandra which only returns the newest version. It is up to the application to decide which value is valid. Voldemort uses memcached to cache data and improve access times. It persists data using BerkeleyDB [3] (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 the Java programming language and exposes its API via Thrift; there are native bindings to high-level languages as well that employ serialization via Google protocol buffers [21]. A shell is also provided for interactive queries. In older version of Voldemort, users cannot obtain all the data for a table in Voldemort; if they wish to do so, they must keep a special key whose value is a list of all the keys in the given table and maintain it accordingly. This functionality has been added in subsequent versions through the get_all function.

2.7 MySQL

MySQL is a well-known relational database that we employ in this work 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 distibution models available; we employ MySQL cluster for this work. This version precludes manual partitioning of the key space and complex client forwarding (sharding). MySQL cluster employs a much simpler distribution model than sharding by using a coordinator to handle writes and replication. The node that performs this function is referred to as the master node, while the other nodes store the actual data are referred to as API nodes. Unlike HBase and Hypertable where clients make requests only to the master node, clients using MySQL cluster can make requests to any of the API nodes. The system can survive the failure of an API node but not the master node. Additionally, the management node is only required for initial configuration and cluster monitoring.

MySQL is written in C and C++. As it is a mature product, it has drivers available in most programming languages that allow programs to access its API. A shell is provided for interactive queries written in SQL, and programs using the native drivers provided can also use the same query language to interact with the database. Bindings allow for query strings to be passed to the database, giving application designers full access to SQL's capabilities.

Setup for MySQL Cluster begins with running the management server (ndb_mgmd). The master node setup uses configuration file specifying the number of replicas and specifying the role of each slave: as a data node, API node, or both. In our configuration, each data node is also an API node. The master node runs ndb_mgmd, which allows the slave nodes to connect to it after each slave is started.

3 Google App Engine and AppScale

As the success and wide spread use of web-based software and services for commercial, social, and personal endeavors continues to grow, so do the offerings available in support of development of such software. Many recent frameworks provide implementation, library, cross-language interoperability, and deployment support and automation for a number of different languages (e.g., Ruby on Rails, Django for Python, Trax for PHP, Struts and Spring MVC for Java). Concurrently, cloud computing is experiencing rapid uptake in the commercial sector, offering an attractive utility-computing paradigm based on Service-Level Agreements (SLAs). Cloud systems offer public access at very low cost to vast proprietary compute, storage, and network resources, along with per-user and per-application isolation and customization via a service interface that is typically implemented using high-level language technologies, APIs, and web services.

Google combines these two offerings within a single platform called App Engine. Google App Engine (GAE) is a software development framework for implementation of 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 key-value datastore, and execute tasks in the background. Figure 1 depicts GAE. The set of available libraries is restricted by Google, i.e. they are those "white-listed" as activities that Google is able to support scalably and safely (in isolation). 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.

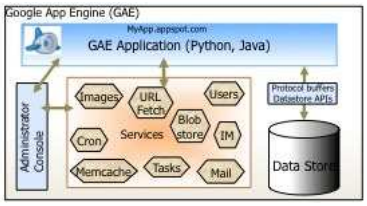

Figure 1: Google App Engine.

Today, Google offers this platform-as-a-service (PaaS) cloud 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). Users can pay for additional bandwidth, CPU hours, disk space, and mail.

The GAE Datastore API is implemented via BigTable [6], and provides the following primitives:

- Put(k, v): Add key k with value v into the database, creating the table for the entity if needed
- Get(k): Acquire the value associated with key k
- Delete(k): Remove key k and its associated value from the database
- Query(q): Perform query q on a single entity and return a list of entities
- Count(t): Determine the number of entities in table t
- Begin/End Transaction: Indicates when transactions begin or end

Google provides a simple query language that is encoded within the Query interface. Data is serialized using Google Protocol Buffers [21]. 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/retreive large files (up to 50MB).

To test their GAE applications and to build the datastore indexes, users execute their application using an open source software development kit (SDK) provided by Google. This SDK implements the GAE APIs using simple, sequential, non-scalable implementations or stubs.

We extend this SDK to implement AppScale. We decouple the implementations from the API and replace them with more scalable and distributed versions. To enable this, we build upon and extend a number of different open technologies (including language implementations, communication and language interoperability frameworks, and databases). AppScale provides a robust, cluster-based implementation of GAE that scales with the size of the cluster available; it executes over virtualized cluster resources and over cloud infrastructures such as AWS and Eucalyptus [20]. We detail the AppScale system in [7] and overview in the subsections that follow. In this paper, we focus on the implementation of the Google Datastore API and

AppScale support for integrating and automating deployment of the open source database implementations for use by web applications and services.

3.1 AppScale Components

Figure 2 depicts an example AppScale deployment. System administrators configure and deploy and 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 resources availability/use. AppScale also implments a web-based interface to the tools for manipulating a extant cloud and viewing the cloud status. For the configuration that we employ for this study, the cloud is static – the size is specified at cloud instantiation and remains the same until the cloud is destroyed.

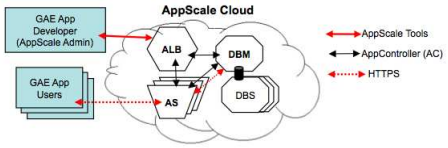

Figure 2: AppScale Deployment.

The primary components of AppScale are the AppController, the AppServer, the AppLoadBalancer, and the AppDB. We briefly overview each. We discuss the AppDB component in the next section as it is the key component for integrating the variety of database systems that we discuss in the previous sections.

AppController. The AppController is a SOAP server written in Ruby that runs on every node in the system. It is spawned by either the AppScale tools if it is the first node or by the first AppController if it is not. 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 a known locations. The AppController also uses iptables to close all ports that are not used by the cloud. Once the system has been set up, all AppControllers sleep waiting for instructions from the first AppController. This AppController sends a heartbeat message every ten seconds to all nodes in the system, 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 (starting/stopping components). The AppController uses tcpdump to estimate the network traffic.

AppServer. The AppServer component is the Google open source GAE SDK with the non-scalable API implementations replaced with distributed and open source versions, that we describe above. We use available Python and Java libraries for many of the services (e.g. Memcache, Instant Messaging (IM), Images), the local system for others (Mail, Cron), and hand-built tools for others (Tasks, URL Fetch). 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 have removed the quota and programming restrictions that Google places in GAE applications that execute within its cloud and have added new APIs including one that provides GAE applications with direct access to a back-end map-reduce implementation if available.

AppLoadBalancer. To partition the access by users to a GAE application web page across multiple servers, we employ the AppLoadBalancer. This component uses the nginx web server [19] to serve static content and haproxy [12] to selectively choose which AppServer should receive incoming traffic. 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. Users instead access a Ruby on Rails service that authenticates the user (providing support for the Google Users API), identifies a target AppServer for the GAE application and redirects the request to that AppServer. Users see the URL change in their address bar. If the AppLoadBalancer fails, the AppServers are not affected.

4 AppScale Distributed Database Support

In this section, we describe how AppScale integrates different databases. Such support is key for easy use and investigation of the vast open source offerings available to web service developers today. AppScale employs the Google Datastore API as a universal API to these different offerings. In this section, we discuss this design and implementation and our experiences coupling this API with extant database systems.

The AppServers send data requests to the datastore serialized as Google Protocol Buffers. In front of the datastores is a server to which we refer to as the Protocol Buffer Server (PBServer). This service is a database-agnostic, multi-threaded, Python HTTP server that consumes requests from an AppServer. The PBServer extracts the request (the Datastore API call) from the Protocol Buffer and calls the API function for the instantiated datastore, passing the Protocol Buffer. Most of the databases that we have considered use a combination of the application's name, class name, and key name as the true key name in the database, with the class contents as the value for the object.

4.1 Deploying a Database

AppScale automates the deployment of distributed database technologies, significantly reducing the learning curve and barrier to entry in using these typically complex distributed systems. We release AppScale as an operating system image by which users can instantiate directly over virtualized cluster resources or cloud infrastructures without any manual configuration or deployment. This essentially provides functionality similar to that of Hadoop-on-Demand (HOD) but for all seven implemented databases [11]. AppScale generates configuration files, command line arguments, and environment variables automatically.

Databases in AppScale currently implement either a master-slave or peer-to-peer (P2P) topology. For master-slave, the AppController designates one node as the Database Master (DBM) and the others as Database Slaves (DBS). For P2P, the AppController designates all nodes as Database Peers. DBM/DBS comprise the AppDB component in AppScale. The distribution dictates the number of entry points for each database, which are the locations to which reads and writes can be directed. A PBServer executes in front of each entry point. For databases with a master-slave configuration, this is the master node. The AppController starts the first AppDB node (master or peer). The AppController then starts all other AppDBs as appropriate, each contacting the first node to join the system.

4.2 Adding a Database to AppScale

To add a new database to AppScale, we modify the PBServer and the AppController. In the PBServer, we add a single function to each of the API calls and include all libraries employed in the function implementations. If the database implements all of the API directly then this addition is simple. However, if it only implements a subset of the API, the developer must write the code that emulates the functionality. For example, if the database does not provide support for adding a new table at runtime or count support, then this functionality must be written by hand. Once the database interface has been implemented it must pass a series of regression tests to verify that database nodes are functioning properly in terms of communication, performance, consistency, and API implementation.

The AppController must also be updated to configure and deploy a new database automatically. It must be able to deploy the various database nodes in the necessary order. For example, in all our implementations, we always deploy the master node (or one peer node) first, and then deploy the slave nodes (or remaining peer nodes). The AppController must also identify the entry points to the database (specifically, where to run the PBServer). A last series of top level regression tests are performed to verify correctness for Google App Engine applications that will run over this new database.

4.3 Our Experience With Integrating Extant Database Technologies

The operating system distribution that we support is Ubuntu Linux v. 9.04. Thus, our experience with the installation and integration of the open source databases we consider is specific to it. However, it is similar to a wide range of other operating systems. We have experimented with others internally and have found that minor modifications are needed to build the AppScale image.

Cassandra. As in the BigTable model, Cassandra implements range queries as opposed to a query language. Range queries allow a developer to access a group of entities that have their key fall in between a certain lexigraphical range. That is, a range query allows a user to use a single access to the database to retrieve multiple data elements. Since Cassandra doesn't allow for dynamic table creation, we simulate multiple tables in a single table. A special meta-table key stores a list of all the tables that exist and is updated accordingly whenever a table is created or deleted. Similarly, a special key is used for each table to store a list of entities currently stored in the table (updated similarly). This is necessary for the query operation, which returns the entire contents of a table. In AppScale, the AppController is configured to run a PBServer on every Database Peer.

HBase. One discernable difference with HBase, compared to the other databases, is the amount of time it takes to create a table (approximately five seconds). This is noticable on the first usage of an application because we create new tables when the first put request is received. Moreover, we found an issue with the jar libraries which HBase was using. An exception would be thrown concerning the use of IP's instead of fully qualified domain names. This problem was fixed with a patch to the HDFS code [10]. A frequent problem we faced came when starting HBase right after HDFS, where all the nodes have not come online in HDFS. This caused a failure for HBase to start because it could not successfully write to HDFS. Our solution for this problem was to poll the number of slave connections made by the master node using the linux command "lsof". Lastly, when doing clean initializations of the datastore, one had to make sure that all the cluster nodes removed the previous data files. Failure to do so led to HDFS throwing an exception about incorrect versions of itself running.

Hypertable. Hypertable's Thrift interface includes the ability to do HQL string queries. Our interface uses both of Thrift's HQL calls and API function calls. The HQL queries require us to i build the string to execute based on the table name, keys, and column names. Test ran at ZEvent have shown that thrift is less than 2bindings [1]. As with HBase, we also use a script to see if HDFS has come up with all slave nodes before starting Hypertable. Keys must be alphanumeric, and require us to use base 64 encoding if application names, or keys require a non-alphanumeric character. Failure to do so causes HQL parsing exceptions. Moreover, we had to patch two files in the code in order to successfully compile it.

MemcacheDB. MemcacheDB is accessed via the standard memcached libraries, and like memcached, provides neither a query langauge nor range queries. It does provide the ability to do bulk reads and writes via the "get_multi" and "set_multi" API calls, respectively. Since table creation is not provided by MemcacheDB, we emulate it via the method described for Cassandra. In order to perform query operations efficiently, we perform one "get" call to the special table key containing the list of keys in the given table and then a "get_multi" call to do a bulk read of all the keys in the table. This allows us to use only two API calls to acquire a table's contents as opposed to having to make a "get" call for each item listed in the special table key. Since MemcacheDB exhibits a master-slave relationship, the AppController starts the Protocol Buffer Server only on the master node.

MongoDB. The API is translated using their native python interface. It is a document oriented database, but used as a key-value store for our analysis. Configuration is simple; start the master and have slaves point to it via the command line argument. No configuration files are needed.

Voldemort. Voldemort exposes its API over the Thrift software framework, so we have constructed the necessary library to interface with it from the Protocol Buffer Server. Voldemort does not expose either a query language or the ability to do range queries. Since Voldemort does not allow for dynamic table creation, we use a single table with the table layout used for Cassandra and MemcacheDB. To implement the query operation, we make a single request to the database to acquire a list of the keys for a given table, and then requests to the database to fetch each key. The number of database requests made for a query operation is therefore dependent on the size of the table in question. Newer versions of Voldemort alleviate this problem by exposing a "get_all" API call that allows for bulk reads, which we suspect will improve query performance. In Voldemort, any node is an entry point to the database, so the AppController runs a Protocol Buffer Server on all nodes.

MySQL. Our initial version of AppScale ran into multiple issues with Mysql Cluster 5.0. Fixed width columns caused for wasted space, and limitations on the size of our entities and applications. The current version in AppScale, MySQL 5.1 NDB 6.3.20, features variable width columns, alleviating the problem. Another limitation is the number of data nodes must be evenly divisiable by the configured number of replicas. Moreover, tables must begin with a letter, and thus we prepend a character to each table name as to act in accordance with the AppScale Datastore API. On startup, we have a script which polls the management console on the head node to verify that all slave nodes have come online before creating our initial tables. The default maximum packet size was also increase in this newer version from 1MB to 16MB. In version 5.0 we would get packet-too-large errors if the default size was not changed.

Database Error Handling. Each database reports errors and exceptions up the chain of procedure calls. How we handle each error and exception depends on the top level Datastore API interface specification. For example, if the AppServer requests a key that does not exist, the error should be passed all the way up to the application level. Conversely, if a put operation causes a TableNotFound exception (e.g., this is the first write operation for the given table), the implementation must catch this error, create the table, and re-insert the given entity. All database interfaces return an array containing the results of a given database operation, reserving the first field for potential errors that may arise.

5 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.

5.1 Methodology

To evaluate the different databases we construct a GAE application that exercises the Datastore API's primitive operations. We fill a table in each database with 1000 items and perform the put, get, delete, and query operations. We repeatedly execute (1000 times for puts/gets/deletes and 100 times for queries) the different operations in order. For each experiment, we access the web page using a machine on the same network. Our measurements therefore consist of the round-trip time to/from the AppServer as well as all database activity. We measure this time using a primitive no-op operation. We include this operation (executing it 1000 times) as part of the scenario. We vary the number of threads that execute the scenario to

consider the impact of load on the system. We consider (i) light load: one thread; (ii) medium load: three concurrent threads; and (iii) heavy load: nine concurrent threads. To lend some insight into the duration of the scenario, we find that a scenario executed by a single thread on a 2-node configuration (more on this below) exercises the system at approximately 25 requests per second.

We execute the application in an AppScale cloud. We consider three static cloud configurations: 1 node, 2 nodes, and 4 nodes. Each node is a Xen guestVM that executes with 2 virtual processors, 10GB of disk (maximum), and 4GB of memory, on a quad-core 2.83GHz machine with 8GB of RAM and a Gb/s network. Only one guestVM executes per physical machine. The 1-node configuration implements an AppLoadBalancer (ALB), AppServer (AS), AppDB Master (DBM), and AppDB Slave (DBS) (or two peers). For the 2- and 4-node configurations, the first node implements an ALB and DBM and the remaining nodes implement an AS and DBS. For heavy load, we employ the 4-node configuration. Although the creators of these datastores use many more nodes in their production servers, we believe there is merit in using four nodes since the datastores still perform well at this level and show variations amongst each other even at this level.

	Light Load	Medium Load
put	0.30	0.28
get	0.27	0.24
delete	0.29	0.28
query	2.04	2.62
no-op	0.24	0.18

Figure 3: Average time (secs) for each operation using Google App Engine (Google's resources).

5.2 Experimental Results

The average time (measured in seconds) for each operation is in Figure 3, showing the GAE application's performing using Google's resources. Note that the number of nodes is unknown here and Google spawns new front-ends in response to load. We consider light and medium load (GAE returned server errors under heavy load). The round-trip time to the application at Google is 240ms with a standard deviation of 4ms, as shown by the no-op data; using AppScale on our local cluster average no-op times range from 77-79ms with a standard deviation of 1-3ms (not shown).

We next present data for the primitive operations. Figure 4 shows three graphs, showing the performance of the put, get, and delete primitive operations, respectively. The medium load case consists of 3 threads accessing the application. Each graph contains three bars for the different AppScale configurations we consider. The x-axis identifies the database used, while the y-axis shows the average query time in seconds. From the left graph, we can see that the master-slave datastores perform the fastest, while the peer-to-peer datastores are slower. As the number of nodes increase in the system, we also see an improvement in put performance across all databases except for Cassandra. The reason behind Cassandra's slow-down is unknown, and is work we are actively investigating. We see similar results for the get scenario. Here, all databases perform similarly and improve as the number of nodes increases, but the peer-to-peer databases and MySQL

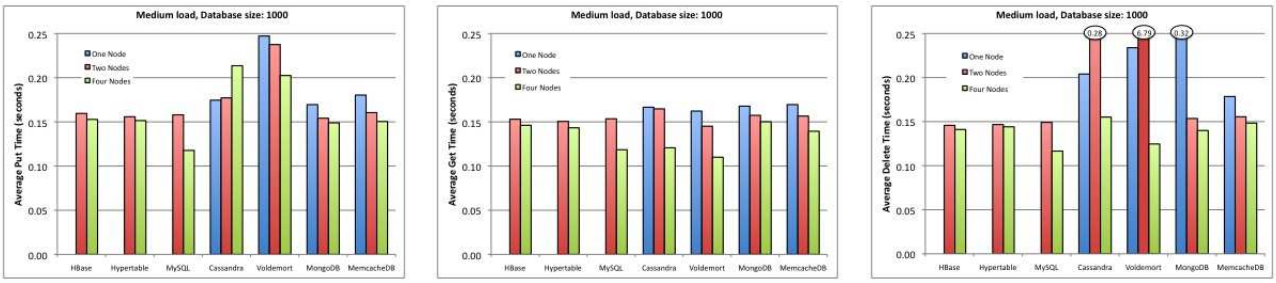

Figure 4: Average time for put (left), get (middle), and delete (right) operations for the different databases using AppScale. We show results for medium load (three threads). The data includes round-trip (no-op) time. For each graph, we show three bars for different numbers of cloud nodes (1, 2, and 4).

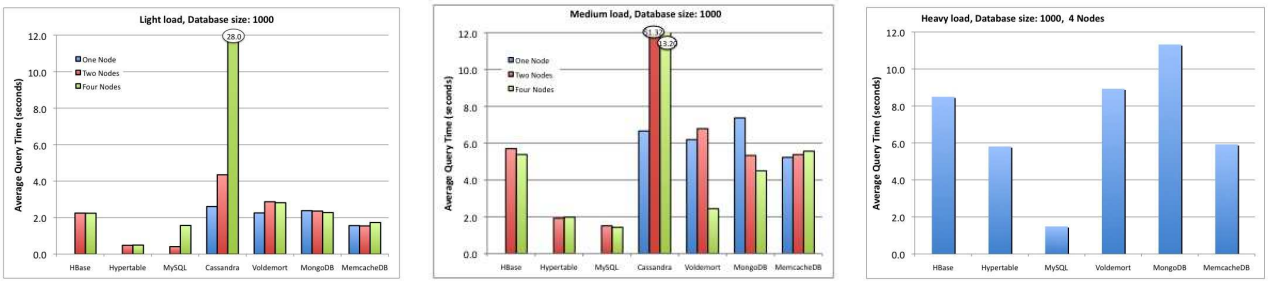

Figure 5: Average time for the query operation under different loads: light (left), medium (middle), and heavy (right).

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. Finally, the delete scenario shows a significant slow-down across several databases at the two node case. This is suspected to be due to the high amounts of database contention on a single Protocol Buffer Server. We suspect that splitting delete requests between the Protocol Buffer Servers on the two nodes would have improved performance due to the increased number of entry points. We omit the data for the light and heavy loads due to space constraints.

Figure 5 shows the performance of the query operation. The left graph shows the system under a light load, while the middle graph shows the system under a medium load, and the right graph shows the system under a heavy load. For light and medium loads, we show the impact of AppScale configurations and database performance. The x-axis identifies the database used, while the y-axis shows the average query time in seconds. Compared to the other primitive operations in the system, the query operation is the slowest. There also are no clear trends across the databases as was the case for the other primitives. We do note that Cassandra performs the worst here at the four node scenario. We believe this is because the default key-partitioning scheme is inefficient for any operation that seeks to acquire all the keys in the system, but are unable to account for why it perform significantly worse at the four node case versus the two node case. This key-partitioning scheme may also explain why Voldemort performs better in the query operation. The medium load case shows that all the databases perform slower than in the light load case as expected, but it is still hard to draw trends from it. The same general rankings amongst the databases are preserved here, and Cassandra still performs slower than the others. Voldemort also experiences an increase in performance at four nodes as compared to two and one node, which is believed to be due to the increase in entry points.

In the right graph, we exclude Cassandra (due to its substantially slower query performance) and put the system under a heavier load of nine accessing threads in a single, four-node AppScale configuration. Once more, the databases tend to perform slower and preserve their general rankings compared to the light and medium load scenarios. HBase, Voldemort, and MongoDB experience a higher degree of slow-down than in the other scenarios, while MySQL and MemcacheDB are able to accomodate the increased load.

6 Conclusions

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 analyze 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`.

References

[1] Personal Communication, 2009.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

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

[4] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Seventh Symposium on Operating System Design and Implementation*, 2003.

[5] Cassandra. `http://incubator.apache.org/cassandra/`.

[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] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles*, 2003.

[9] Hadoop. `http://hadoop.apache.org/`.

[10] Hadoop issues. `https://issues.apache.org/jira/browse/HADOOP-5191`.

[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.

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

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

[18] MySQL. `http://www.mysql.com`.

[19] Nginx. `http://www.nginx.net`.

[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/`.