An Evaluation of Distributed Datastores Using the AppScale Cloud Platform

Chris Bunch Navraj Chohan Chandra Krintz Jovan Chohan Yoshihide Nomura
Jonathan Kupferman Puneet Lakhina Yiming Li
Computer Science Department
University of California, Santa Barbara
{cgb, nchohan, ckrintz, jovan, jkupferman, puneet, yiming_li}@cs.ucsb.edu

Software Innovation Laboratory
Fujitsu Labs Ltd., Japan
nomura@pobox.com

***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 Terms*—database concurrency operations; cloud computing; platform-as-a-service (PaaS);**

I. Introduction

Cloud computing is an attractive utility-computing paradigm based on Service Level Agreements (SLAs) that is experiencing rapid uptake in the commercial sector. Cloud systems offer low cost public access to vast proprietary compute, storage, and network resources. 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 [20]), is scalable and fault tolerant structured data management. 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.

Open source DBs have emerged which emulate the functionality of popular cloud DBs including HBase, Hypertable, Cassandra, and others. Many of which have been customized and are in use internally by successful web service providers, including Facebook, Baidu, SourceForge, and LinkedIn. The appearance of these new (proprietary and open source) data management systems have spawned a heated debate over the use of traditional relational data models (e.g. those offered by SQL-based systems such as Oracle, Sybase, and MySQL) versus these "NoSQL" (non-relational) systems for cloud-based applications. Such discussions focus on the differences in scale, dynamism, complexity, and ease of use.

In addition to data model, these offerings (both relational and non-relational) vary widely and can differ in query language, topology (master/slave vs peer-to-peer), data consistency policy, replication policy, programming interfaces, and implementations in different programming languages. In addition, each system has a unique methodology for configuring and deploying the system in a distributed environment. As a result, the use and deployment of any of these systems imposes a significant learning curve on web application developers, making it challenging for them to compare and evaluate these systems for different applications. Furthermore, the sheer number of datastores that have emerged recently make it extremely difficult to perform a meaningful comparison across these very different datastores in a repeatable, consistent fashion.

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). We use this system to empirically evaluate and compare how well seven different popular DB technologies map to the GAE Datastore API (without modification or customization). We find that the DB

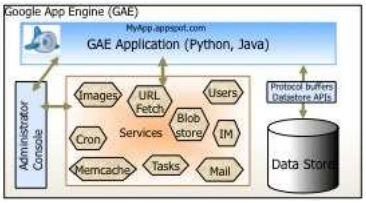

Fig. 1. Depiction of Google App Engine (GAE).

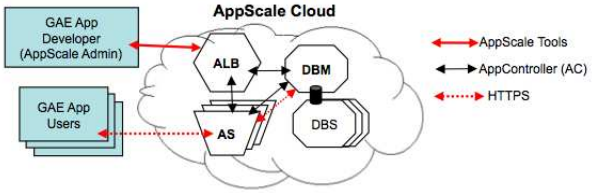

Fig. 2. Depiction of an AppScale Deployment.

technologies employed vary greatly in their performance characteristics, typically in favor of those with more entry points for reading and writing data.

II. Google App Engine

Google App Engine (GAE) is a software development platform depicted in Figure 1 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 [6], 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 [9] (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 [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/retrieve large files (up to 50MB).

To better understand the functionality, behavior, 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 [7] 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.

III. AppScale

Figure 2 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 non-scalable 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 [18] 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 [12]. 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.

IV. 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 AppScale 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) [11] 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 AppControllers 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.

A. Cassandra

Facebook engineers designed, implemented, and released the Cassandra datastore as open source [4] 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 [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. 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.

B. HBase

Developed and released by PowerSet as open source in 2007, HBase [13] 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) [10]. 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.

C. 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 [15].

Hypertable is written in C++ to facilitate greater control of memory management (caching, reuse, reclamation, etc.) [14]. 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 [3], a highly available locking and naming service that stores very small files.

D. 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 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. Users 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.

E. 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 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.

F. Voldemort

Developed by and currently in use internally at LinkedIn, Voldemort emulates Amazon Dynamo and combines it with caching [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 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 [1] (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 [21]. A shell is also provided for interactive queries.

G. 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.

V. 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.

A. Methodology

To evaluate the different datastores, we use Active Cloud DB [2], 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.

B. 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 3 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 4. 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 4 in the right graph and the other results in Figure 5. 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

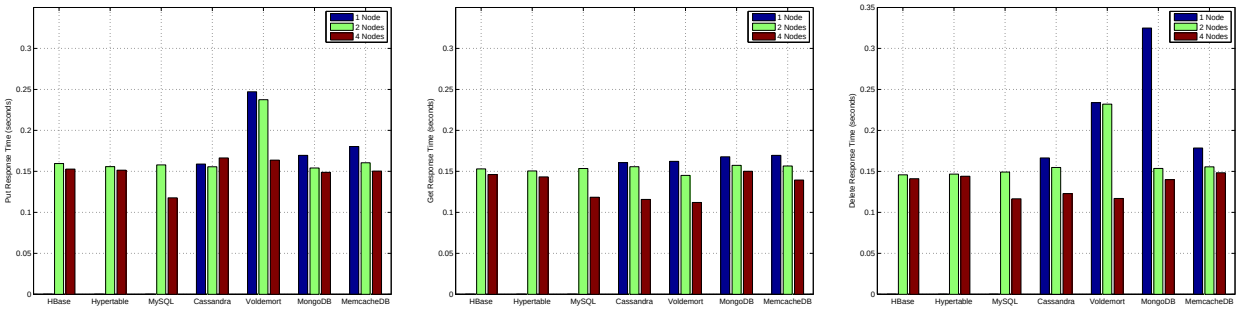

Fig. 3. Average time for put (left), get (middle), and delete (right) operations for the different databases using AppScale. We show results for medium load (three threads). The data includes round-trip (no-op) time. For each graph, we show three bars for different numbers of cloud nodes (1, 2, and 4). We were unable to run HBase, Hypertable, and MySQL in a single node configuration, so no data is present for those scenarios.

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 significanly 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.

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 no-op/round-trip time) using the Google cloud (with an unknown number of nodes) is similar to those for AppScale/MySQL with four or more nodes (the difference is statistically insignificant). Moreover, the Google DB access times show higher variance.

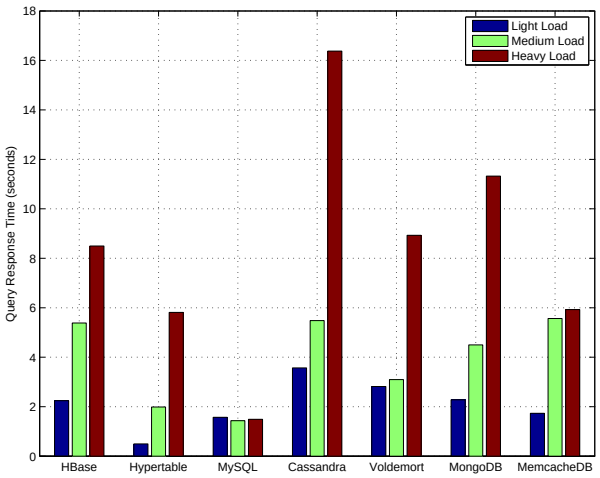

Fig. 4. Average time for the query operation under different loads using the four node configuration.

VI. 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

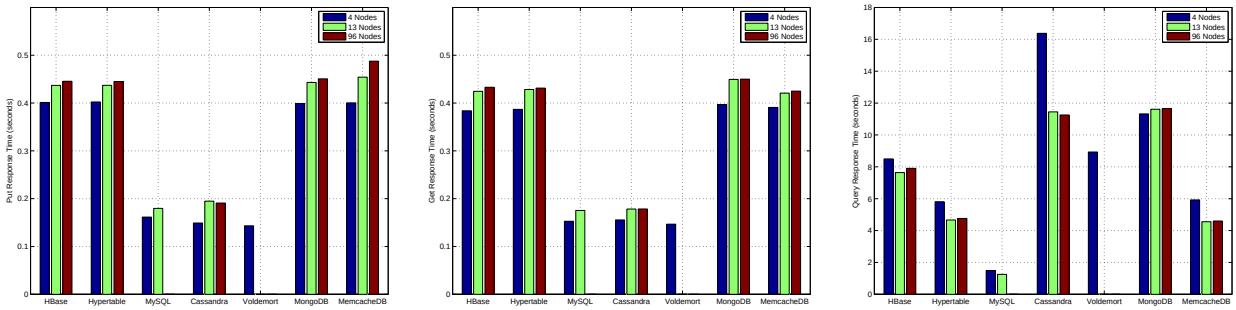

Fig. 5. Average time for the put (left), get (middle), and query (right) under heavy load as we increase the number of nodes in the cloud configuration. Note that the y-axis scale differs for query (right graph). MySQL Cluster does not support greater than 48 nodes; Voldemort returns errors when the 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 configuation 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.

VII. 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.

VIII. Acknowledgements

We would like to thank the developers and communities of the datastores in this paper for their help and support. This work was funded in part by Google, IBM, and the National Science Foundation (CNS/CAREER-0546737, CNS-0905273, and CNS-0627183).

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/.