Active Cloud DB: A RESTful Software-as-a-Service for Language Agnostic Access to Distributed Datastores

Chris Bunch   Jonathan Kupferman   Chandra Krintz

Computer Science Department
University of California, Santa Barbara

Abstract. In this paper, we present Active Cloud DB, an open source Software-as-a-Service (SaaS) application that allows for RESTful access to cloud-based distributed datastore technologies that implement the Google Datastore API. We implement Active Cloud DB as a Google App Engine application that we employ to expose the Google App Engine Datastore API to developers – for use with any language and framework. We evaluate this SaaS on both Google App Engine and over AppScale, the open-source implementation of Google App Engine that enables Google App Engine applications to execute on cloud infrastructures without modification. As part of this work, we extend Active Cloud DB with simple caching support to improve the performance of datastore access and evaluate our technique with and without this support. We also make use of this support within multiple client-facing prototypes (e.g. Ruby on Rails, Python through Django) to show the ease-of-use and applicability of our contribution to other web development environments.

Key words: Cloud Computing, SaaS, Open-Source, REST, Distributed Systems

1 Introduction

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 non-trivial and highly visible cloud systems for web services, specifically BigTable [5] within Google, Dynamo [1] within Amazon, and Cassandra [4] within Facebook, a wide variety of open-source variations of distributed key-value stores have emerged and are gaining widespread use.

However, these datastores implement a wide variety of features that make them difficult for prospective users to compare. For example, there are differences in query languages, topology (master/slave vs peer-to-peer), consistency policies, and end-user library interfaces. As a result, we and others have investigated a single framework with which such systems can be compared [6, 3, 7].

The Yahoo! Cloud Serving Benchmark (YCSB) provides a database interface and a synthetic workload executor for exercising the DBs that the authors attach to the interface. The system measures the response time of primitive operations in a workload between a thread on one machine and the datastore on another. The authors support four datastores: HBase, Cassandra, PNUTS (Yahoo's proprietary key-value store), and Sharded MySQL.

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 configuation of datastores that implement the API and faciliates their comparison and evaluation on end-to-end performance using real programs (Google App Engine applications). AppScale 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. YCSB does not support applications at all, but instead spawns requests between the server and the datastore for the sole purpose of measuring datastore response time and throughput.

In this work, we address the problem of how to facilitate access to these 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 Cloud DB (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.

In the sections that follow, we overview AppScale, the implementation and evaluation framework we use to investigate the performance of our approach. We then describe Active Cloud DB and the caching support with which we extend it. Next, we describe four proof of concepts we have developed that make use of Active Cloud DB, evaluate this Software-as-a-Service, and conclude.

2 AppScale

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 [10]. Users can execute their existing Google App Engine applications over AppScale without modification. [6] 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

AppServer within AppScale. AppScale integrates open-source datastore systems as well as new software components that faciliate 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. [3] 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 [4]. It incorporates the flexible column layout from the former and the peer-to-peer node topology from the latter. Its peer-to-peer layout allows the system to avoid having a single point of failure as well as be resiliant 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 [12] 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 [9] is a modification of the popular open source distributed caching service `memcached`. Buildling upon the `memcached` API, MemcacheDB adds replication and persistence using Berkeley DB [2] 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.

3 Active Cloud DB

We next present Active Cloud DB, a Software-as-a-Service that executes over AppScale to expose datastores to cloud clients. Clients are web-based applications and software implemented using languages and web frameworks other than those supported by Google App Engine. Active Cloud DB exports RESTful [8] access to cloud-based distributed datastore technologies automatically and scalably.

Active Cloud DB is implemented as a Google App Engine application that executes over AppScale (and thus Google App Engine). Its implementation requires no modification to either AppScale or Google App Engine. To enable scalability and low response times, we also extend Active Cloud DB with datastore caching support to improve the performance of datastore access by applications. We first overview Active Cloud DB and then describe the design and implementation of our caching scheme.

3.1 Implementation

Our goal with Active Cloud DB is to remove the limitation imposed by AppScale and Google App Engine that requires that applications be written in Python or Java to gain access to the functionality of the distributed datastore back-ends that these cloud fabrics implement. This is particularly important for Google App Engine, since not only must applications be written in these languages but they must employ the web frameworks specified by Google using a restricted set of "whitelisted" libraries. AppScale removes this constraint but currently still only supports Python and Java Google App Engine applications.

Active Cloud DB is a Google App Engine application that runs over AppScale that exposes an RESTful API to the underlying datastore. Internally, Active Cloud DB allows for objects to be created with a given name (key) and one string within (its value).

This can be trivially extended to allow for all data types, but this work considers only string-type keys and values. It can make requests either to AppServers directly or to the AppLoadBalancer in the system, who can intelligently route traffic to AppServers but can result in higher latency (as it is an extra hop on all requests).

Active Cloud DB exposes a single URL route in the typical RESTful fashion. Specifically, the route (named `resources`) and supports the standard Google App Engine datastore functions depending which HTTP Method is used. If the route is called with the `GET` method with no parameters (e.g., simply `/resources`), a datastore query is performed, returning all the keys in the datastore and their associated values. If the route is called with the `GET` method with a single parameter (e.g., `/resources/foo`), then a key-lookup is performed on that parameter, returning the associated value (in our example this would look up the value associated with the key `foo`). If the route is called with the `POST` method with two paramters (e.g., `/resources/foo/bar`), then a put operation is performed with the given key and value (in our example this would store the key `foo` with value `bar`). Finally, if the route is called with the `DELETE` method with a single parameter (e.g., `/resources/foo`), then the key and associated value are deleted from the system (in our example this would delete the key `foo` and its associated value).

3.2 Integration

To communicate with Active Cloud DB, developers implement the client side-interface. We have done this for four popular web frameworks, Rails and Sinatra for Ruby, and Django and web.py for Python. In Rails and Django, we remove the built-in database abstractions and in all add in a wrapper for communicating with Active Cloud DB. It abstracts all remote communication logic and error handling such that the application developer need not be aware that the database is located remotely. In all frameworks we use the default templating library for creating the presentation layer of the applications.

The four prototype applications with which we make use of Active Cloud DB implement a simple bookstore application (inspired by the application given in [13]). To access Active Cloud DB, the application makes a RESTful request with the key and value to put, or the key to get or delete. The bookstore application maintains a special key containing a list of all the books in the datastore, which is maintained whenever books are added or removed. When a book is added, we change this special key accordingly and then add an entry to the datastore with the book's name (the key) and a summary of the book's contents for users to view (the value). When users wish to view all books in the bookstore, we access the special key to get a list of all the books, and for each book (key), we return the corresponding book information (its value).

In lieu of using this special key, we could have simply performed a database query, but previous work [3] has found that the query operation tends to take much longer than the get operation once a non-trivial number of keys are in the datastore.

3.3 Caching Support

To reduce latency and improve throughput to/from the datastores in AppScale and Google's hosting environment, we provide a transparent data caching layer. To enable this, we leverage the Google Memcache API (similar to `memcached`). The API caches web service data and provides a least-recently-used replacement policy.

To efficiently cache data, we combine two caching strategies, write-through and generational. For basic operations (get, put, delete) we employ a *write-through* caching strategy. With this strategy all put operations are written to the datastore as well as to the cache. In doing so, subsequent requests are likely to be served directly from cache, avoiding any datastore interaction.

Efficiently caching query operations is more complex than basic operations because query results can contain multiple data items. Furthermore, when a particular data item is updated one must expire all query results which contain that item so that stale data is not returned. In order to ensure that this property is maintained we utilize a *generational* caching strategy. In essence, a generation value is maintained for the data. The generation value is included in the cache key for all query operations. Hence, by changing the generation value all prior cached results are implicitly expired as they can never be accessed again.

Specifically, the operations are:

– Get(k): Return value associated with key k. If the value was not in the cache, store it for future accesses.
– Put(k, v): Add key k and value v to table and the cache. Increment the generation value.
– Delete(k): Remove key k and its value from the table and the cache. Increment the generation value.
– Query(q): Perform query q using the Google Query Language (GQL) on a single table, returning a list of values. Store the result in the cache with the current generation number for future queries.
– Count(t): Acquire the query data via the new query technique, and return the size of the list of values found.

4 Evaluation

We next employ Active Cloud DB over AppScale to evaluate the performance characteristics of the various supported datastores. We begin by describing our methodology and then present our results.

4.1 Methodology

For our experiments, we measure the performance of the primitive operations performed in bulk and as part of an overall workload. In both scenarios, this is done over two back-end AppScale datastores, Cassandra version 0.5.0 and MemcacheDB version 1.2.1-Beta. For the first set of experiments, we fill a table in each database with 1000 items and perform the get, put, query, and delete operations. To provide a baseline measurement we also perform a no-op operation which simply returns and performs no back-end processing. We invoke each operation 1000 times (100 times for query since it takes significantly longer than the others). A query retrieves all 1000 items from the datastore.

For each experiment, we access Active Cloud DB using a machine on the same network. Our measurements are of round-trip time to/from the AppServer as well as all database activity. For each experiment, there are nine concurrent threads that each perform all of the operations and record the times for each. We consider multiple static configurations of the AppScale cloud that consists of 4, 8, 16, 32, 64, and 96 nodes. On each node, AppScale runs a Database Slave/Peer, a Protocol Buffer Server, and an AppServer. Each thread accesses a single AppServer, so nine threads are in use in our setting. For each configuration, there is also a head node that implements the AppController and the Database Master if there is one (otherwise it is a Database Peer).

The second set of experiments test the performance of the primitive operations of the system when performed as part of an overall workload. Here, the number of nodes is constrained to 16 nodes (due to space limitations) and 10000 random operations are performed with a 50/30/20 get/put/query ratio. Once an operation is selected, nine concurrent threads perform the operation and access their corresponding AppServers. We intentionally perform the operation on a single key in the datastore in order to maximize the amount of contention in the system.

As both of the datastores here have a number of settings that can be used, we configure both Cassandra and MemcacheDB in a particular way throughout AppScale deployments. While Cassandra allows the user to specify the consistency requirements on all operations, we set it to use inconsistent reads and writes: only one node in the system is needed to participate in these operations. Conversely, in MemcacheDB, we direct all reads and writes to the master node in the system. While it does offer the ability to read from any database node, these initial tests access only the master node and use the replicas for data backup. Current work is underway to expand AppScale to read from any database node in the system.

4.2 Results

We first present results for web application response time between Active Cloud DB and the datastores. Response time includes the round-trip time between Active Cloud DB and datastore including the processing overhead of the Protocol Buffer Server.

Figure 1 displays results for the get, put, and delete operations. The left graph shows the performance of the get operation. The additional entry points for Cassandra allows it to process reads faster than MemcacheDB. Additionally, varying the number of nodes in the system does not have a significant impact on the performance of the get operation. With caching, there is an improvement in performance for both Cassandra and MemcacheDB. This is because write-through caching leads to cache hits for all but the first event and cache access is significantly faster than datastore access.

The right graph shows the performance of the put operation. As was the case in [3], the increased number of entry points for Cassandra allow it to process writes faster than MemcacheDB. The reduced consistency requirements also allow for faster writes in Cassandra. Like in the case of the get operation, we see that the performance does not drastically change in either direction for either datastore with respect to the number of nodes in the system. We see the same trends occurring for the same datastores when

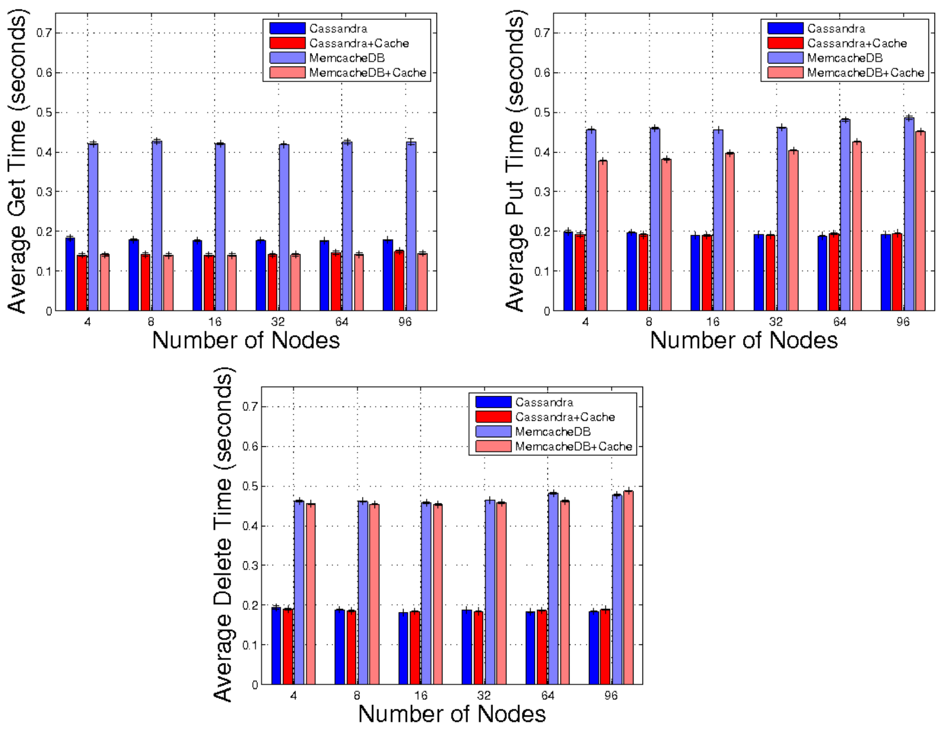

Fig. 1. Average round-trip time for get (left), put (right), and delete (bottom) operations under a load of 9 concurrent threads for a range of different AppScale cloud sizes (node count).

caching is employed, and about the same performance for a datastore regardless of whether the cache is employed or not. This shows that the overhead of performing caching is negligible with respect to the overall time of the operation. The bottom graph shows the performance of the delete operation. Deletes perform similarly to puts for both datastores as well as with and without caching.

Figure 2 shows the performance of the query operation. This operation is to be the slowest in the system since it operates on an entire table and returns all the keys instead of operating on a single key. For our experiments a query operation returns all 1000 items in the datastore, and we see that having more entry points negatively impacts the datastore's ability to return all the items for a given table. This impact is consistent across the various node deployments, with Cassandra consistently performing worse than MemcacheDB.

Employing the caching scheme negates this difference since all reads (except for the first) access the cache instead of the datastore. This yields results and conclusions very similar to that for the get operation, but with a degraded performance due to the fact that

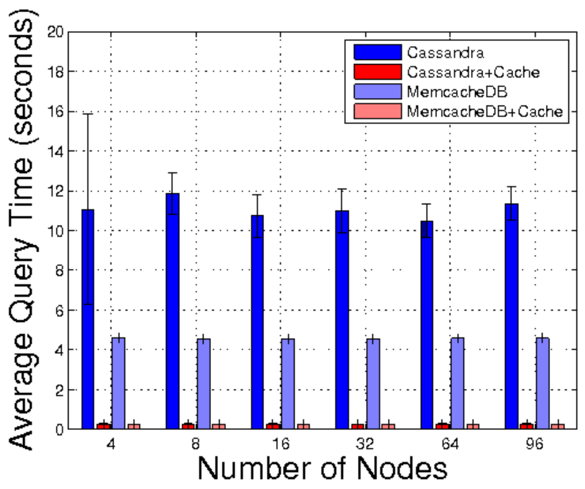

Fig. 2. Average time for query operation for different node configurations and 9 concurrent threads.

our caching scheme must marshal and un-marshal the data when caching it. Therefore, un-marshalling all 1000 items in the datastore constitutes the difference between these operations. Similarly to the get operation, this speedup only applies without writes: a single write causes performance to degrade for the next read.

We next consider a second workload. Figure 3 shows the performance of the system under a 50/30/20 get/put/query workload across 16 nodes. All operations perform faster than in the previous experiments, as this workload is performed on an initially empty database. However, the same trends from before are preserved in this workload analysis. Get operations are still faster for Cassandra than MemcacheDB, but now both are significantly slower than their cached equivalents. This is likely due to the substantially smaller amount of data in memcached, allowing for much faster read access. As was the case in the previous experiments, write performance is roughly the same whether or not caching is employed. Finally, query performance is substantially better than in the previous experiments. This is to be expected since the database has substantially less information in it than in the previous experiments. Caching the data has less of an impact here since the non-cached versions perform much better than in the previous experiments.

Finally, we experiment with executing Active Cloud DB on Google's resources. Table 1 shows the average response time using a load of three threads as opposed to nine for our original first workload (in-order, repeated operation execution). Google continuously killed our 9-thread even though we were within our quota. We also summarize the AppScale results (the same as from the previous graphs using the 16-node configuration). For this data, we include the no-op data which we did not present in graph form. Note that for experiments on Google's resources, we have no control over the number of nodes we have been allocated for our application.

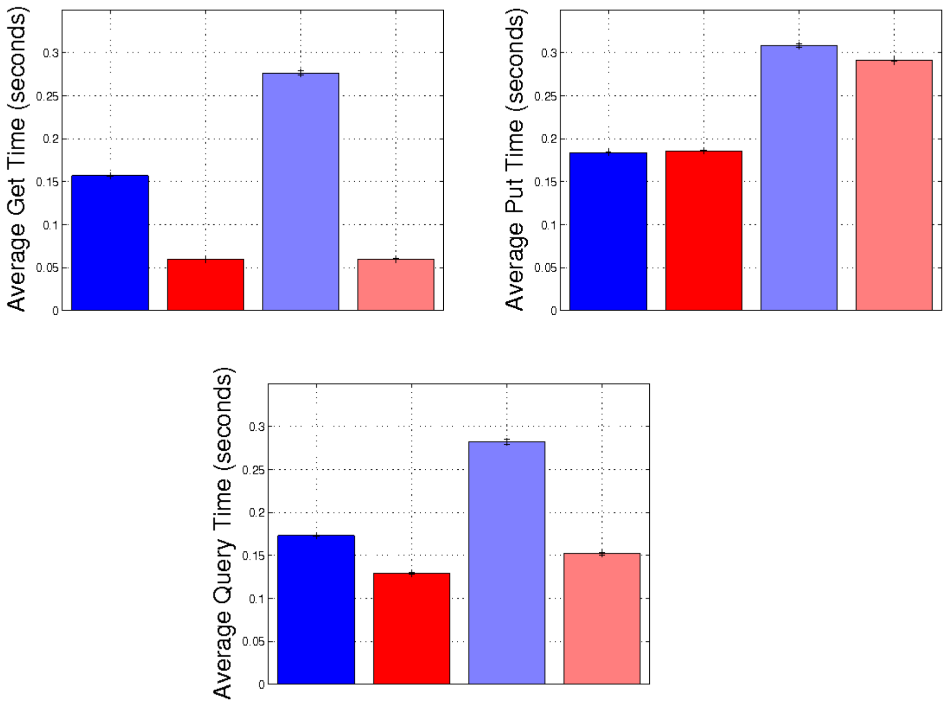

Fig. 3. Average round-trip time for get (left), put (right), and query (bottom) operations under a load of 9 concurrent threads for a 50/30/20 get/put/query workload, run over 16 nodes. The legend is the same as that used for Figure 1 and Figure 2.

5 Conclusion

We present Active Cloud DB, a Software-as-a-Service that runs over the Google App Engine cloud. Active Cloud DB is a Google App Engine application that executes over Google App Engine or over its open-source counterpart, AppScale. It exposes the Google Datastore API via REST to other languages and frameworks. We evaluate its use within Google and AppScale and present a number of proof of concept applications that make use of the interface to access a wide range of diverse key-value stores easily and automatically. We also extend Active Cloud DB with simple caching support to significantly improve query performance for Active Cloud DB and other Google App Engine applications that execute using an AppScale cloud. The proof of concept applications, AppScale, and Active Cloud DB, can all be found at `http://appscale.cs.ucsb.edu`.

Table 1. Average Active Cloud DB time for each operation using Google App Engine (node count unknown) and AppScale (16 nodes).

	Google App Engine Medium Load	AppScale 16 node Cassandra Cache	AppScale 16 node MemcacheDB Cache
put	0.28	0.19	0.40
get	0.24	0.14	0.14
delete	0.28	0.18	0.14
query	2.62	0.26	0.26
no-op	0.18	0.14	0.14

References

1. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.
2. BerkeleyDB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.
3. C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura. An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *IEEE International Conference on Cloud Computing*, 2010.
4. Cassandra. `http://incubator.apache.org/cassandra/`.
5. 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.
6. 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 *ICST International Conference on Cloud Computing*, Oct. 2009.
7. B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB, Mar. 2010. `http://www.brianfrankcooper.net/pubs/ycsb.pdf`.
8. R. T. Fielding. Architectural styles and the design of network-based software architectures. Ph.D. Dissertation, University of California, Irvine, 2000.
9. MemcacheDB. `http://memcachedb.org/`.
10. 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`.
11. Protocol Buffers. Google's Data Interchange Format. `http://code.google.com/p/protobuf`.
12. M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation, Apr. 2007. Facebook White Paper.
13. D. Thomas and D. Hansson. Agile Web Development with Rails: Second Edition. 2006.