Noname manuscript No.
(will be inserted by the editor)

Cloud Platform Datastore Support
UCSB Technical Report #2011-08

Navraj Chohan · Chris Bunch · Chandra Krintz · Navyasri Canumalla

Received: date / Accepted: date

Abstract [1]

Recent technological advances in hardware and software have facilitated the explosive growth in the production of digital information. Cloud systems offer tremendous scale, resource availability, and ease of use, with which we can process this data in the pursuit of scientific, financial, social, and technological advances. However, there are many systems to choose from that differ in many ways including public versus private cloud support, data management interfaces, programming languages, supported feature sets, fault tolerance, consistency guarantees, configuration and deployment processes.

In this paper, we focus on technologies for structured data access (database/datastore systems) in cloud systems. Our goal is to simplify the use of these systems through automation and to facilitate their empirical evaluation using real world applications. To enable this, we provide a cloud platform abstraction layer that decouples a data access API from its implementation. Applications that use this API can use any datastore that "plugs into" our abstraction layer, thus enabling portability. We use this layer to extend the functionality of multiple datastores without modifying the datastores directly. Specifically, we provide support for ACID transaction semantics for popular key-value stores (none of which provide this feature). We integrate this layer into the AppScale cloud platform – an open-source cloud platform that executes cloud applications written in Python, Java, and Go, over virtualized cluster resources and infrastructures-as-a-service (Eucalyptus and Amazon EC2). We use this system to investigate the overhead of providing this application portability layer for disparate datastores and the impact of extending them via the layer with distributed transaction support.

Computer Science Department
University of California, Santa Barbara
E-mail: {nchohan, cgb, ckrintz, navyasri} @ cs.ucsb.edu

[1] This paper is a combined and extended version of papers [5] and [12].

1 Introduction

Recent advances in hardware and software have culminated in the emergence of cloud computing – a service-oriented, computing model that simplifies the use of large-scale distributed systems through transparent and adaptive resource management, automating configuration, deployment, and customization for entire systems and applications. Using this model, many high-technology companies have been able to make their proprietary computing and storage infrastructure available to the public (or internally via private clouds) at extreme scales.

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

Public and private cloud providers increasingly employ specialized databases, called key-value stores (or datastores) [10,13,14,11,20,8,36,39,28,21]. These systems support structured data access over warehouse-scale

resource pools, by large numbers of concurrent users and applications, and with elasticity (dynamic growing and shrinking of resource and table use). Examples of public cloud datastores include Google's BigTable, Amazon Web Services (AWS) SimpleDB, and Microsoft's Azure Table Storage. Examples of private or internal cloud use of datastores include Amazon's Dynamo [14], and customized versions of open source systems (e.g. HBase [20], Hypertable [21], Cassandra [8], etc.) in use by use by Facebook, Baidu, SourceForge, LinkedIn, Twitter, Reddit, and others.

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

In this paper, we address two growing challenges with the use of cloud-base datastore technologies. The first is the vast diversity of offerings: applications written to use one datastore must be modified and ported to use another. Moreover, it is difficult to "test drive" public offerings extensively without paying for such use, and challenging to configure and deploy distributed open source technologies in a private setting. The second challenge is the lack of support for atomic transactions across multiple keys in a table. Most datastores offer atomic updates at the row (key) level only. The lack of all-or-nothing updates to multiple data entities concurrently precludes many business, financial, and data-analytic applications and significantly limits datastore utility for all but very simple applications.

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

To address the second challenge, we extend this layer to provide distributed transactional semantics for the datastore plug-ins. Such semantics increase the range of applications that can make use of cloud systems. Our approach emulates and extends the limited transaction semantics of the Google App Engine cloud platform to provide atomic, consistent, isolated, and durable (ACID) updates to multiple rows at a time for any datastore that provides row-level atomicity. To enable this, we rely on ZooKeeper [43], an open-source locking service in use by cloud fabrics and other distributed systems.

We implement this database-agnostic software layer within the open source AppScale cloud platform and integrate a number of different popular open source and proprietary database and datastore systems. These plug-ins include Cassandra, HBase, Hypertable, and MySQL cluster [30] (which we employ as a key-value store), among others. Moreover, since AppScale executes over different infrastructure-as-a-service (IaaS) cloud systems (Amazon EC2 [1] and EUCALYPTUS [32,16]) and emulates Google App Engine functionality, developers are given the freedom to choose the infrastructure on which their application runs on.

We use the system to empirically evaluate different datastore systems and to evaluate the overhead of employing distributed transactions. We find that the database-agnostic portability layer adds approximately 7-8 ms per put/get (read/write). We find that DAT adds an additional 16 ms to each put/write (and no additional overhead to each get/read) on average. We find that the scalability, throughput, and latency of transactional applications (e.g. a financial exchange) using this layer and DAT, varies significantly across datastores and workloads but that Cassandra significantly outperforms other datastores including MySQL cluster (without DAT) which uses *native* (built-in) transaction support.

In the sections that follow, we first describe the design and implementation of AppScale and its abstract database layer that decouples the AppScale datastore API from the plug-ins (implementations of the API). We then describe how we extend this layer with ACID transaction semantics in a database-agnostic fashion in Section 3. We then present an evaluation of the system using different datastores in Section 6, present related work in Section 7, and conclude in Section 8.

2 The AppScale Database Support and Portability Layer

In this work, we provide a database-agnostic software layer for cloud platforms that decouples the datastore interface from its implementation(s) and automates distributed deployment of datastore systems. We design and implement this layer as part of the AppScale cloud platform and then extend it to support database-agnostic distributed transaction support.

AppScale is an open source cloud runtime system that enables applications written high level languages (Python, Java, and Go) to execute over virtualized clusters and cloud infrastructures. To enable this, AppScale implements a set of APIs for a multitude of cloud services using existing open source technologies. To make AppScale attractive to application and service developers, the AppScale APIs include all those made available by Google App Engine (GAE). By doing so, any application that executes over GAE can execute in a private cluster setting over AppScale and vice versa.

GAE is a public cloud platform to which users upload their applications for execution on Google's resources. Applications invoke API functions for different services. When a user uploads her GAE application (it is made available by GAE via a subdomain on appspot.com), the APIs connect to proprietary, scalable, and highly available implementations of each service. GAE 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. Users are charged a fee based on the resources their applications use beyond a specified quota. AppScale emulates this cloud platform functionality using private/local virtualized clusters and/or infrastructure-as-a-service (IaaS) systems Amazon EC2 and EUCALYPTUS.

AppScale can execute GAE applications without whitelist restrictions at the cost of reverse GAE compatibility, if doing so is desirable by the cloud administrator. AppScale also implements a wide range of other APIs, not available in GAE, in support of more computationally and data intensive tasks. These APIs include those for MapReduce, MPI, and UPC programming, and StochKit for scientific simulations [6].

Figure 1 shows the AppScale software stack. At the top of the stack are the application servers that serve Python, Java, and Go applications. The AppScale APIs that the applications employ leverage existing open source software such as eJabberD [15] and memcacheD [27], or custom services (e.g. blobstore) that we provide, for their implementations. AppScale uses Nginx [31] and HAProxy [19] to route and load balance requests to the application servers. Nginx provides SSL connections, and HAProxy performs health checks on servers, routing only to responsive application servers. A background service on each node in an AppScale cloud restarts any service that stops functioning correctly. An AppScale cloud consists of a set of virtual machine instances (nodes), each of which implement this software stack.

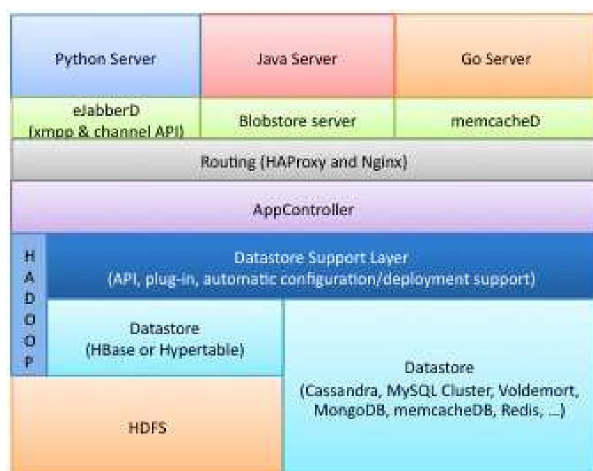


Fig. 1 The AppScale Software Stack. Herein, we present the design and implementation of the database software layer and its extensions in support of distributed, database-agnostic, multi-key transactional semantics.

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

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

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

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

The data values that AppScale stores in the datastore are called entities (data objects) and are similar to those defined by GAE [41]. Each entity has a key object; AppScale stores each entity according to its key as a serialized protocol buffer [35].

GAE implements this API using proprietary key-value systems called Megastore [2] and BigTable [11] and charges for access to these systems both in terms of the amount of storage and number of API calls. AppScale implements its datastore API using popular open source, distributed datastore systems including HBase [20], Hypertable [21], Cassandra [8], Redis [36], Voldemort [39], MongoDB [29], SimpleDB [38], and MySQL Cluster [30]. HBase and Hypertable both rely on HDFS [17] for their distributed file system implementations, as does the MapReduce API which integrates Hadoop MapReduce [18] support.

To automate configuration and deployment of distributed datastores for users in a private setting, we release the AppScale system as a single virtual machine image. This image consists of the operating system kernel, Linux distribution, and the software required for each of the AppScale components services (the software in the stack displayed in Figure 1). When an AppScale cloud is deployed, a cloud administrator employs a set of AppScale tools to instantiate the image (over Xen, KVM, or an IaaS system). This instance becomes the head node which starts all of its own services and then does so for all other nodes (instantiated images) in the system. Each AppScale cloud deployment implements a single datastore (cloud-wide).

The AppController in the system interacts with a template to configure and deploy each datastore dynamically upon cloud instantiation. The set of scripts configure, start, stop, and test an instantiated datastore using the following API:

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

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

3 Database-Agnostic Distributed Transaction Support

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

3.1 DAT Design

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

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

To enable multi-entity transactional semantics, we employ the notion of entity groups as implemented in GAE [41]. Entity groups consist of one or more entities, all of whom share the same ancestor. This relationship is specified programmatically. For example, the Python code for an application that specifies such a relationship looks as follows:

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

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

The keys for each of these entities are `app_id\Parent:Alice`, and `app_id\Parent:Alice\Child:Bob` for p (Alice) and c (Bob), respectively. Alice is a root entity with attributes type (kind), `key_name` (a reserved attribute), and balance. The key of a non-root entity, such as Bob, contains the name of the application and the entire path of its ancestors, which for this example, consists of only Alice. It is possible to have a deeper hierarchy of entities as well. AppScale prepends the application ID to each key to enable multitenancy for datastores which do not support dynamic table creation and thus share one key space (i.e., Cassandra as of version 0.68).

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

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

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

3.2 DAT Semantics

DAT enforces ACID (atomicity, consistency, isolation, and durability) semantics for each transaction. To enable this, we use multi-version concurrent control (MVCC) [3]. When a transaction completes successfully, the system attempts to commit any changes that the transaction procedure made and updates the *valid version number* (the last committed value) of the entity in the system. The operations *put* or *delete* outside of a programmatic transaction are transparently implemented as transactions. If a transaction cannot complete due to a program error or lock timeout, the system rolls back any modifications that have been made, i.e., DAT restores the last valid version of the entity.

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

The implementation of transaction semantics GAE and AppScale differ, each having their own set of tradeoffs. GAE implements transactions using optimistic concurrency control [4]. If a transaction is running, and another one begins on the same entity group, the prior transaction will discover its changes have been disrupted, forcing a retry. An entity group will experience a drop in throughput as contention on a group grows. The rate of serial updates on a single root entity, or an entity group depends on the update latency and contention, and ranges from 1 to 20 updates per second [2].

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

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

4 DAT Implementation

To implement DAT within AppScale, we provide support for entities, an implementation of the programmatic datastore interface for transactions (run_in_transaction), and multi-version consistency control and distributed transaction coordination (global state maintenance and lock-

ing service). To support entities, we extend the AppScale key-assignment mechanism with hierarchical entity naming and implement entity groups. Each application that runs in AppScale owns multiple entity tables, one for each entity `kind` it implements. We create each entity table dynamically when a *put* is first invoked for a new entity type. In contrast, GAE designates a table for all entity types, across all applications. We chose to create tables for each entity kind to provide additional isolation between applications.

We implement an adaptation of multi-version consistency control to manage concurrent transactions. Typically timestamps on updates are used to distinguish versions [3]. However, not all datastores implement timestamp functionality. We thus employ a different, database agnostic, approach to maintaining version consistency. First, with each entity, we assign and record a version number. This version number is updated each time the entity is updated. We refer to this version number as the *transaction ID* since an update is associated with a transaction. We maintain transaction IDs using a counter per application. Each entry in an entity table contains a serialized protocol buffer and transaction ID.

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

4.1 Distributed Transaction Coordinator (DTC)

To enable distributed, concurrent, and fault tolerant transactions, DAT implements a Distributed Transaction Coordinator (DTC). The DTC provides global and atomic counters, locking across entity groups, transaction blacklisting support, and a verification service to guarantee that accesses to entities are made on the correct versions. The DTC enables this through the use of ZooKeeper [43], an open source, distributed directory service that maintains consistent and highly available access to metadata using a variant of the Paxos algorithm [26,9]. ZooKeeper is the open source equivalent to Google's Chubby locking service [7]. The directory service allows for the DTC to create arbitrary paths, on which both leaves and branches can hold values.

The API for the DTC is

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

DAT intercepts and implements each transaction made by an application (*put*, *delete*, or programmatic transaction) as a series of interactions with the DTC via this API. A transaction is first assigned a transaction ID by the DTC (`getTransactionID`) which returns an ID with which all operations that make up the transaction are performed. Second, DAT obtains a lock from the DTC (`acquireLock`) for the entity group over which the operation is being performed. For each operation, DAT verifies that all entities accessed have valid versions (`getValidTransactionID`). For each *put* or *delete* operation, DAT registers the operation with the DTC. This allows the DTC to track of which entities within the group are being modified, and, in the case where the application forces a rollback (applications can throw a rollback exception within the transaction function) or any type of failure, the DTC can successfully know what the current correct versions of an entity are. The API call of `registerUpdateKey` is how previously valid states are registered. This call takes as arguments the current valid transaction number, the transaction number which is attempting to apply changes, and the root entity key to specify the entity group.

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

Transactions that start, modify an entity in the entity table, and then fail to commit or rollback due to a failure, thrown exception, or a timeout, are *blacklisted* by the system. If an application attempts to perform an operation that is part of a blacklisted transaction, the operation fails and DAT returns an exception to the application. Application servers that issue operations for a blacklisted transaction must retry their transaction under a new transaction ID. Any operations which were executed under a failed transaction are rolled back to the previous valid state.

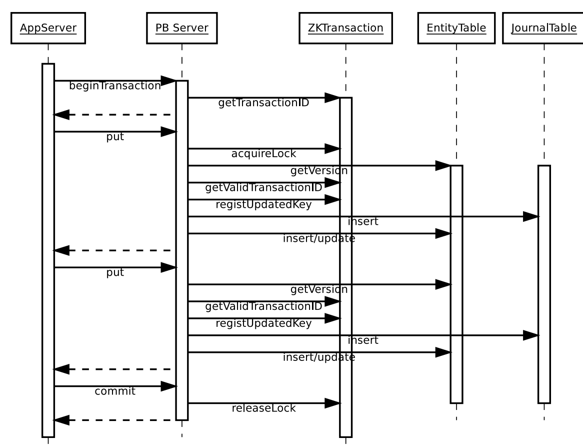


Fig. 2 Transaction sequence example for two puts.

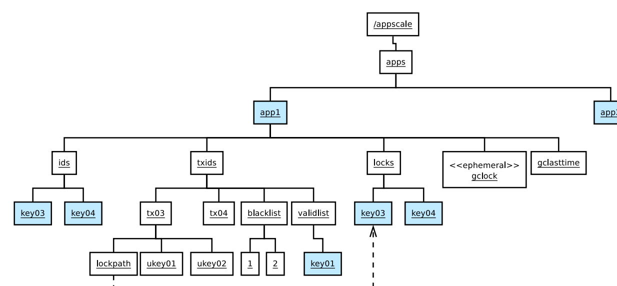


Fig. 3 Structure of transaction metadata in ZooKeeper nodes.

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

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

Either using the original version (transaction ID) or the transaction ID returned from the DTC due to invalidation, DAT creates a new journal key and journal entry (journal keys are guaranteed to be unique), registers the journal key with the DTC, and in parallel performs an update on the entity table. We overview these steps with an example in Figure 2 and show the DTC API being used during the lifetime of a transaction.

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

4.2 ZooKeeper Configuration of the DTC

We present the DTC implementation using the ZooKeeper node structure prefix tree (trie) in Figure 3. We store either a key name as a string (for locks and the blacklist) or use the node directly as an atomically updated counter (e.g., for transaction IDs). The tree structure is as follows:

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

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

Transactions implemented by DAT provide transactional semantics at the entity group level. We implement a lock subtree that is responsible for mapping a transaction ID to the entity group it is operating on. The name of the lock is the root entity key and it stores the transaction ID. We store the locking node path in a child node of the transaction named ”lockpath”. Any new transaction that attempts to acquire a lock on the same entity group will see that this file exists which will cause the acquisition to fail. This lock node is removed when a transaction completes (via successful commit or rollback).

4.3 Scalable Entity Keys

We employ ZooKeeper sequential nodes to implement entity counters (these should not be confused with transaction IDs). When entities are created without specifying a key name, IDs are assigned in an incremental fashion. We ensure low overhead on key assignment by allocating blocks of 1,000 entity IDs at a time to reduce the overhead of counter access. The block of IDs is cached by the instance of the call handler in the database support layer. Keys are provisioned on a first-come-first-serve basis to new entities which do not have a key name. There is no guarantee that entity IDs are ordered.

Entity IDs use two types of counters for concurrent access. One counter is for root keys of a specific entity type, while another counter is created for each child of a root key. Entity IDs are stored under the inner node upon creation and are removed once committed. The node structure holds values for each entity group as seen in the */appscale/apps/app1/ids* path. The ”ids” node contains the next batch value for all root keys, while $key03$ and $key04$ nodes hold values for the next batch of child keys.

4.4 Garbage Collection

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

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

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

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

4.5 Fault Tolerance

DAT handles certain kinds of failures, excluding byzantine faults. Our implementation of the DTC ensures that the worst case timing scenario does not leave the datastore in an inconsistent state (”Heisenbugs”) [22].

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

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

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

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

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

5 Methodology

In this section, we overview our benchmarks and experimental methodology. For our experiments, AppScale em-

ploys Hadoop 0.20.2, HBase 0.89, Hypertable 0.9.4.3, MySQL Cluster 6.3.20, Redis 2.2.11, Voldemort 0.80, MemcacheDB 1.2.1, and Cassandra 0.6.8. We execute AppScale using a private cluster via the Eucalyptus cloud infrastructure. Our Eucalyptus private cloud consists of 12 virtual machines with 4 cores, and 7.5 GB of RAM. We use two public clouds–Amazon's EC2, using m1.large instances (4 cores and 7.5 GB of RAM), and Google App Engine, where the infrastructure is abstracted away. We synchronize the clocks across the cluster using the Linux tool `ntpdate` for both our Eucalyptus cluster and EC2 cluster.

5.1 Single Node Benchmark

Our first experiment measures the latency of datastore operations with and without the database support layer and with and without transaction support. We refer to the measurements without the database support layer as `direct` operations. We refer to the measurements with the database support layer as `indirect` operations. We refer to measurements with transaction support as `trans` and without as `notrans`.

The experiment consists of a single node deployment using different datastore/database technologies over Eucalyptus. We measure datastore latency (end-to-end time) for put and get operations. For each experiment, we make sequential requests, first storing (using `puts`) 1,000 entities in the datastore and measuring each request, and then retrieving each entity (using `gets`). Each entity has a payload of 10 KB. We report the average time with standard deviation across these requests.

5.2 Transaction Benchmarking Application

We then experiment more extensively using distributed cloud deployments and a smaller set of datastores. For these experiments we use a transactional benchmark that emulates a banking application. In particular, our application transfers money from one account to another. Each instance of the application creates 100,000 accounts. A request made to the system has a receiver account number, a sender account number, and the amount to be transferred.

The implementation of the bank transfer requires two separate transactional functions. The first transaction subtracts the requested amount from the sender and creates a transfer record which is a child entity of the sender account. The second transaction adds the requested amount to the receiver and creates a transfer record signifying the successful receivable funds. Errors during the first transaction result in a failure to transfer the funds. Errors in the second transaction require a retry using a cron job which scans for incomplete transfers. Our results only consider a request a success if both transactions were successful.

In our experimental setup, three machines serve to issue different numbers of concurrent bank transfer requests. The numbers we report are for 10, 100, 1,000, and 10,000 requests. We measure the number of successful operations and the latency (round trip time and application time) of the successful requests with 10 trials each. Round trip time is the time it takes for the request to receive a response. Application time is a portion of the round trip time: the time for an application server to handle a request. The difference between the round trip time and application time is the latency to and from the server and all queuing delay.

We set the replication factor to two for these experiments. Our AppScale implementation employs 20 application servers per node for our Eucalyptus and EC2 cluster. For GAE, Google uses its own scheduling policy to enable the scaling of applications, and it is unknown how many servers are being employed.

6 Results

We first present the overhead of the database support layer and DAT. We then present an extensive evaluation of the performance and scalability of DAT. We close this section with a comparison of AppScale/DAT over EC2 and Eucalyptus versus GAE's proprietary implementation.

For this experiment we perform sequential puts and gets for payloads of 10 KB using an AppScale single node deployment cloud as described in the previous section. We consider the different datastores that we currently support with the DAT layer and present the results in Tables 1 and 2 and 3.

Table 1 (labeled `Direct`) presents the average time of puts and gets (with standard deviation) when we access each datastore directly, i.e. without our database support layer. Table 2 (labeled `Non-Trans`) shows this time with our database support layer. The difference between these two sets of numbers is the overhead imposed by this layer to enable application portability across cloud and datastore systems, system simplicity, elasticity, datastore plug-in support, and automation of distributed datastore deployment. On average, the database support layer adds 8.3 ms to each put and 6.9 ms to each get (when we omit consideration of the HBase direct access anomaly). The maximum overhead is 9.8 ms for puts and 7.6 ms for gets. HBase direct access time for gets seems to be an outlier, but the data is repeatable – the database support layer significantly improves its performance.

The layer increases the variance for both operations for all but HBase. The overhead and variance increase

Direct	Puts (msec)	Puts (stdev)	Gets (msec)	Gets (stdev)
Cassandra	0.40	± 0.12	0.45	± 0.10
Redis	0.50	± 0.07	0.22	± 0.03
HBase	39.30	± 5.66	38.10	± 8.48
Hypertable	1.38	± 0.66	0.47	± 0.03
Voldemort	14.50	± 2.08	2.53	± 0.52
MemcacheDB	2.57	± 3.39	0.31	± 0.01

Table 1 A comparison of datastore latencies for put and get using direct accesses.

Non-Trans	Puts (msec)	Puts (stdev)	Gets (msec)	Gets (stdev)
Cassandra	8.81	± 4.99	8.09	± 4.11
Redis	8.25	± 4.60	7.75	± 3.78
HBase	46.50	± 6.87	8.33	± 4.33
Hypertable	9.26	± 4.05	7.34	± 3.38
Voldemort	24.30	± 4.76	7.84	± 3.68
MemcacheDB	11.50	± 19.60	7.61	± 3.70

Table 2 A comparison of datastore latencies for put and get using the database support layer in AppScale (without transactions support).

Trans	Puts (msec)	Puts (stdev)	Gets (msec)	Gets (stdev)
Cassandra	13.30	± 6.83	7.85	± 3.99
Redis	12.00	± 3.78	7.91	± 3.94
HBase	89.70	± 80.20	8.06	± 4.15
Hypertable	15.40	± 5.75	6.96	± 3.08
Voldemort	57.40	± 7.69	7.79	± 3.55
MemcacheDB	21.60	± 63.30	7.36	± 3.47

Table 3 A comparison of datastore latencies for put and get using the database support layer in AppScale (without transactions support).

observed here comes primarily from Nginx and HAProxy layer which intercepts and load balances requests across the system. This impact is similar to that for any use of a load-balancing system in front a distributed database implementation.

Table 3 (labeled `Trans`) presents results with our database support layer (in AppScale) *and* transaction support. The difference between this table and Table 2 is the additional overhead that is required to support transaction semantics in the system. This overhead is required regardless of whether the applications executing over the cloud use transactions in their code. On average, this layer adds 16 ms (with a maximum difference of 43 ms) to each put operation and no additional overhead on gets (the difference is not statistically significant and in some cases shows an improvement in raw value due to noise). The variance is similar to that without transaction support as well. The additional overhead on puts is due to calls to the ZooKeeper system and to writes to the journal. For gets, our system employs a only single, very fast, ZooKeeper call to verify that the version of the entity is valid for a particular get operation. The differences in the amount of overhead across datastores derives from how well the interface to the datastore is implemented (e.g., the handling of connection pools) by its progenitors, and the degree of parallelism that each datastore is able to achieve for journal and entity table writes (puts).

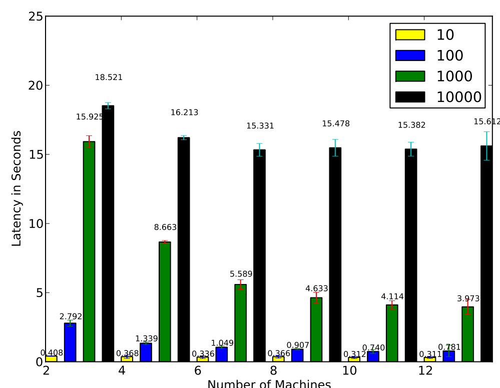
Since the database support layer enables multiple datastore technologies to be plugged in and thus evaluated using the same empirical setup, we can also compare and contrast the performance and feature sets across datastores. The data in these tables also show that Redis has the fastest access time for both puts and gets. This is because it is primarily an in-memory datastore. Because of this, it does not supply the full durability required for ACID semantics. Yet, if the application does not require strong durability in the face of failures and the dataset is small enough to fit within RAM, Redis is an excellent choice of datastore. Other limitations of Redis include a lack of sharding of data across nodes, replication occurs only on and across all slave nodes, and all writes (puts) must go through a master node. All of these factors impact scalability and system performance when significant amounts of data are stored.

Cassandra shows very low latency is as well. Cassandra, however, persists data using a write-ahead-log. This architecture allows for fast puts because seeks are minimized. Writes are done to memtables which are periodically flushed to immutable Sorted String Tables (SSTables). As SSTables accumulate over time, compactions occur to combine these data files. For both puts and gets, we configure Cassandra to use a quorum, where a majority of the replicas must respond.

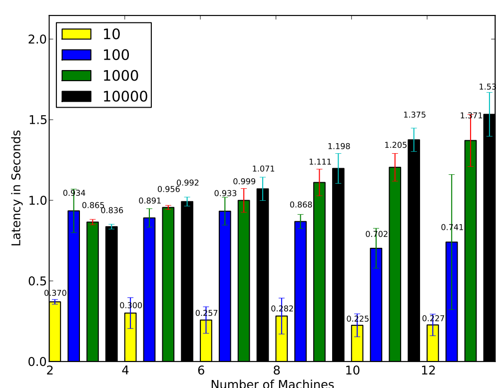
HBase imposes the longest latencies for this set of datastores. HBase was initially designed for fault tolerance and reliability, although newer releases now focus on performance improvements, so its position in this comparison should improve over time. Moreover, HBase's performance is highly dependent on HDFS, which provides the distributed file system. The choice of Java for its implementation (as opposed to C++ used in other systems) also impacts performance, memory footprint, and pause times (for periodic garbage collection). HBase exhibits much higher variance in put and get latency due to these factors in general, and garbage collection interruptions in particular.

Hypertable performs very well compared to its similar counterpart HBase, for both puts and gets, 9.26 ms and 7.34 ms, respectively. While Hypertable also relies on HDFS for its distributed storage, it does not have the drawback of having its own periodic garbage collections (since the datastore is written in C++).

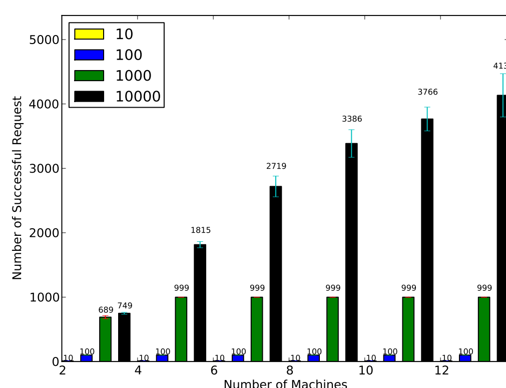
Voldemort and MemcacheDB both use Berkeley DB as their persistence layer, although MemcacheDB has more



(a) Round trip time of successful transaction requests.



(b) Application time of successful transaction requests.



(c) Number of successful transaction requests.

Fig. 4 Cassandra results as the number of machines increases.

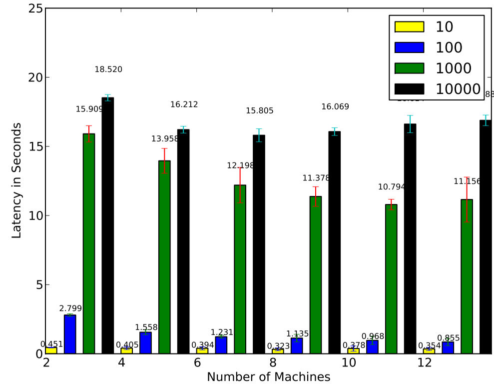
than half the latency of Voldemort for puts, while get performance is very similar. Both datastores require (for AppScale query support) the use of a set of reserved meta-keys and a locking mechanism via memcached to keep track of which keys belong to which table or keyspace because the datastore itself does not provide for it. The management of this keyset imposes very large overheads on all operations for these datastores. As such, we do not include them in our performance evaluation of DAT in the next section. Range query support for these datastores will enable them to be more competitive with the others when/if it is provided by the developers of these systems.

6.1 Transaction Performance

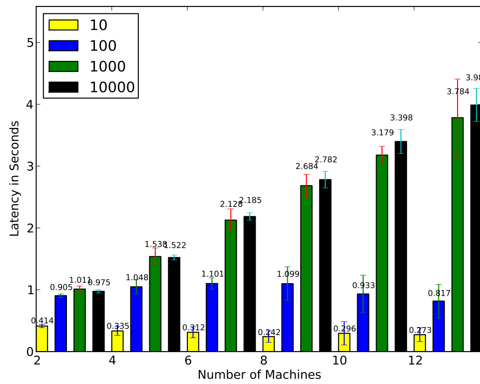
We next evaluate more extensively, the transaction performance of our system and datastores. We measure transaction performance with three metrics, (1) round trip time from the requesting client to the application server and back, (2) the latency of a request to be serviced by an application server, and (3) the number of successful requests out of the total number of requests sent. As described in Section 5, evaluate our system for different loads and number of nodes.

We focus on those datastores which provide consistent, scalable, and dependent performance. From here forward, for brevity and reasons previously mentioned, we focus on Cassandra, HBase, Hypertable, and MySQL Cluster distributed database/datastore systems. Moreover, these datastores are representative of others and represent different points in the design space of database technologies (peer-to-peer vs master/slave, implementation languages, maturity, etc.). Even though MySQL Cluster provides support for relational queries, we use it as a key-value store like the others and use its native transaction support to compare to the other datastores under the DAT system.

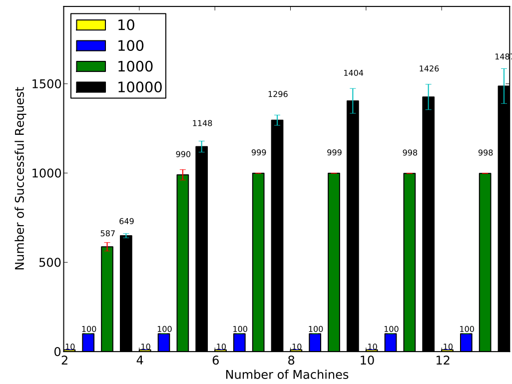
Figure 4 shows Cassandra's performance when the number of machines varies from 2 to 12. Figure 4(a) depicts the round trip time of successful requests and each



(a) Round trip time of successful transaction requests.



(b) Application time of successful transaction requests.



(c) Number of successful transaction requests.

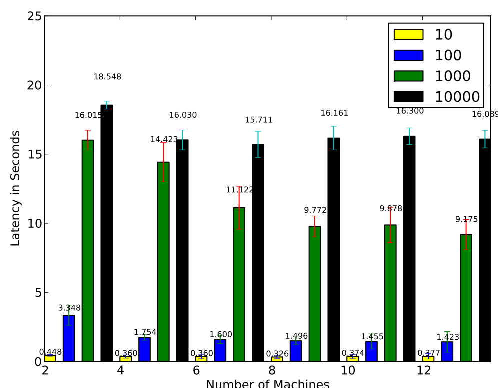
Fig. 5 HBase results as the number of machines increases.

series shows a different load level. The lowest load level, where most the system is left idle has the best response time, 408 ms for a 2 node configuration, and 311 ms for a 12 node configuration. Additional nodes help alleviate load removing queuing delay. This is the case for all load levels. At the highest load we see that the system is overloaded and additional machines are needed beyond 12 nodes. Whereas 10,000 concurrent requests do not see much improvement in latency, Figure 4(c) shows that more requests are being fulfilled linearly as more machines are added. At 2 nodes there are 749 requests being served, while a 12 node configuration can serve 4,134 requests. For lower loads of 1,000 requests or less, most requests are handled successfully with 4 or more machines.

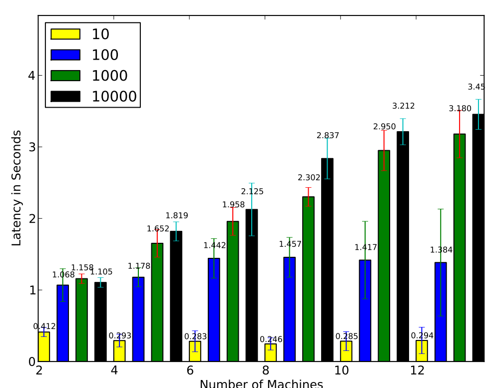
Figure 4(b) shows application time which is dominated by datastore access. There is a general trend of taking longer to access the datastore as more load is added. This also happens when more machines are added because there is more contention for datastore access as more requests are handled.

HBase performance is shown in Figure 5. Figure 5(a) shows round trip time for an increasing number of machines with a varying load. Compared to Cassandra we see that the latency does not drop as much as it did for Cassandra for the 1,000 request case. For Cassandra the 1,000 request case for 12 nodes was measured at 3.973 seconds, but HBase is 11.156 seconds and exhibits higher variance under the same conditions. Moreover, we see that Figure 5(b) shows higher application time which is dominated by HBase's handling of datastore requests.

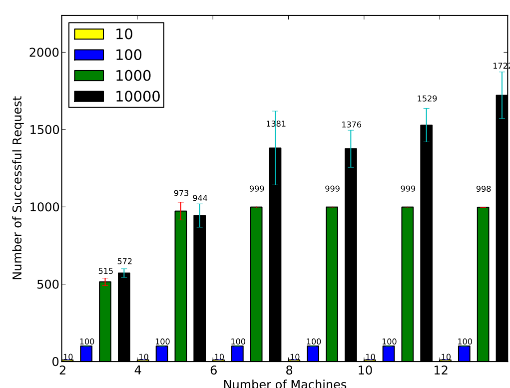
Figure 5(c) has similarity with Cassandra for loads of 1,000 or less in that near 100 percent of requests are handled, but these requests are handled with much more latency than Cassandra. Lastly, the most requests that are handled under a 12 node configuration is on average only 1,487. We suspect this is due to hot tablet contention, where the key space is located on a particular node that is overwhelmed with requests.



(a) Round trip time of successful transaction requests.



(b) Application time of successful transaction requests.



(c) Number of successful transaction requests.

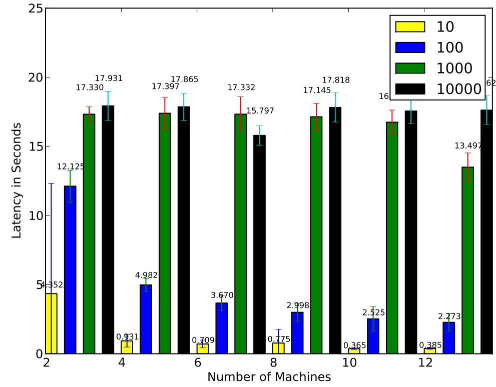
Fig. 6 Hypertable results as the number of machines increases.

Hypertable has very similar characteristics to that of HBase as seen in Figure 6 because both provide similar implementations of Google's BigTable architecture. The round trip time (Figure 6(a)) and the application time (Figure 6(b)) both show slightly favorable latencies for Hypertable over HBase primarily because of Hypertable's use of C++ over Java and thus better memory management (no garbage collection overhead). At 12 nodes with the highest load, Hypertable completes 1,732 requests with an average latency of 16.089s, compared with HBase's 1,437 requests at 16.884s.

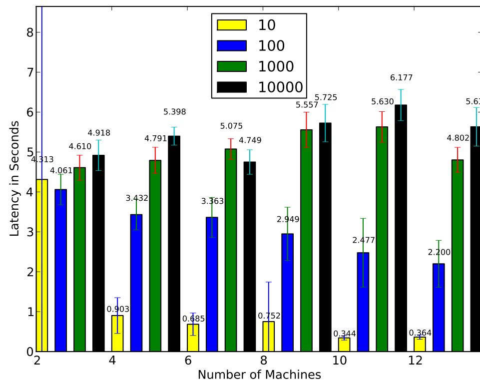
Figure 7 presents MySQL Cluster's performance for a varying number of machines. MySQL's transaction implementation does not use the DAT system, rather it employs its own native transaction support. All requests to begin a transaction and commit a transaction go directly to the datastore and transaction handles are assigned by MySQL as opposed to using ZooKeeper. MySQL is not informed of entity groups and cannot leverage this information for its locking purposes and thus uses course grain locks limiting its ability to scale the number of concurrent transactions as see in Figure 7(c). Latencies are also much higher for application time, which go as high as 6.177s.

Our next set of graphs (Figure 8 provide a side-by-side comparison between different databases (although the data is repeated from the graphs presented previously) for the heaviest load (10,000 requests). Figure 8(a) compares the round trip time, Figure 8(b) compares application request latencies, and Figure 8(c) compares the number of successful transactions completed for the workload. Application request time again is the time between when the application server receives the request and when it replies (the round trip time minus the routing time and queuing delay) to the application.

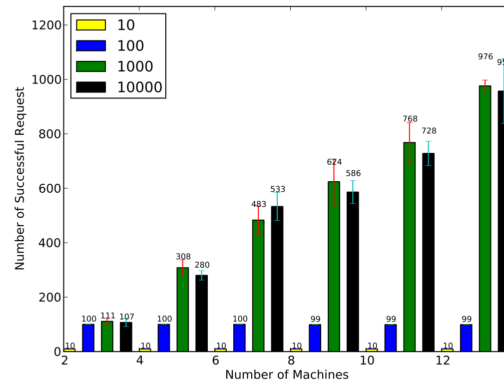
The comparison shows similar round trip time across datastores. However, there is a significant difference in the number of completed requests (Figure 8(c)). The lower the latency on the application time, the more requests can be served. Also for all systems, as the number of ma-



(a) Round trip time of successful transaction requests.



(b) Application time of successful transaction requests.



(c) Number of successful transaction requests.

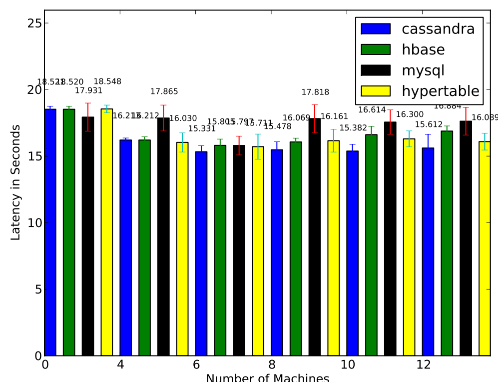
Fig. 7 MySQL Cluster results as the number of machines increases.

chines increases there is more time spent in the datastore layer and less in front-end queuing delay. In summary, under this load, Cassandra serves the most requests and MySQL cluster serves the least. Overall Cassandra with our DAT layer imposes lower latencies and higher scalability than any other datastore, followed by Hypertable and HBase and then by MySQL cluster. MySQL cluster performance and scalability are limited by its use of course-grain locking.

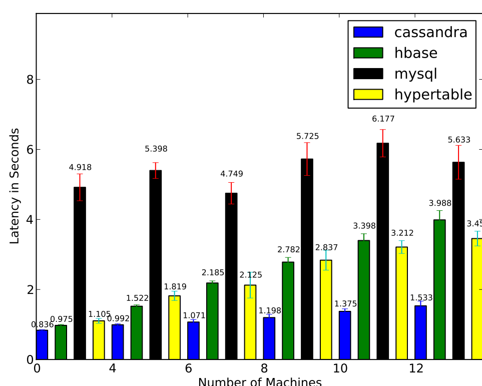
We next focus on the Cassandra data store and investigate the overhead of the DAT system for transactional workloads. We measure the overhead of DAT measured by disabling the calls to ZooKeeper and journal writes, each of which adds latency and therefore reduces the total number of processed requests when under high load. Note that this disables full ACID semantics and we only include it to lend insight into the performance of the DAT layer. In this scenario, each transactional block become a series of unprotected put and get operations.

The results are shown in Figure 9. While round trip time is very similar (Figure 9(a)), we see that application time is much less and ranges from 0.84-1.53 seconds under heavy load to 0.64-0.90 seconds with transactions disabled.

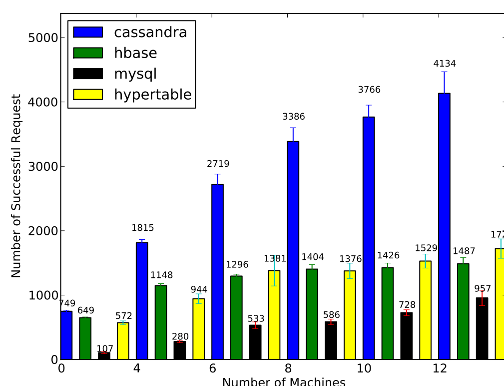
The number of successful requests, shown in Figure 9(c), is as high as 7,182 with transaction support turned off (it is 4,134 with it turned on). The overhead comes in two primary forms which are required for DAT. First is the additional write which happens to the journal. This means there is double the load to the datastore system for each application put. To keep this overhead low, both puts to the entity table and the journal are performed in parallel. However, under heavy loads response times can increase. Additionally, each ZooKeeper operation is performed in a blocking manner, which is required for transactional semantics. An optimization we employ is to use asynchronous calls to ZooKeeper when appropriate (when doing so does not invalidate the semantics). These calls are not required to be successful because they



(a) Round trip time of successful transaction requests.



(b) Application request time of successful transaction requests.



(c) Number of successful transaction requests.

Fig. 8 A comparison of different datastores under heavy load as the number of machines increases. This data is also presented in the previous graphs and combined here for easy comparison.

will be redone through the garbage collection mechanism. An example of when this is employed is during the commit phase of a transaction. Nodes can be asynchronously deleted, and if a failure happens, a garbage collection thread finds and removes them.

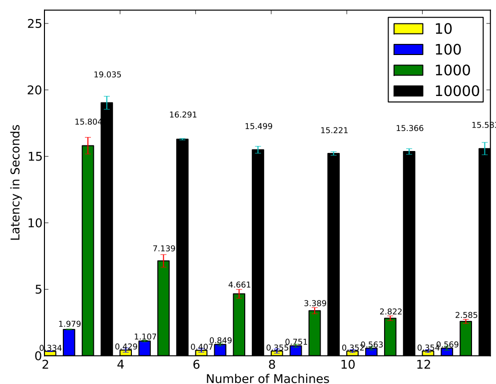
6.2 AppScale and Other Cloud Fabrics

We next compare our transactional system across two different public cloud offerings, and do so using the same transaction benchmark. We employ AppScale on EC2 using Cassandra for a 12 node configuration and benchmark it under high load (10,000 requests). The requests originate from three nodes within the same EC2 region to minimize latency for the round trip time. Google App Engine runs the same application unmodified, yet in this deployment the number of machines and resources is unknown. For the App Engine deployment billing is enabled to ensure our benchmarking is unimpeded by quotas. The requests for GAE originate from the same three machines we used for our Eucalyptus testing, which are three nodes from within our local cluster.

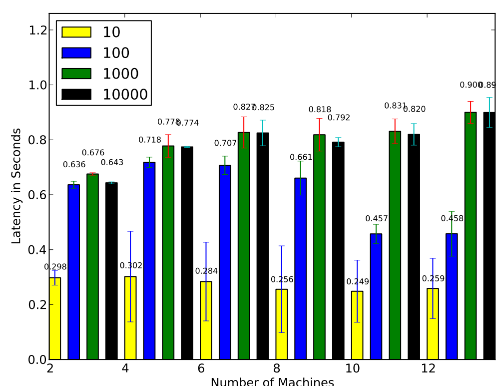
	Successes	App Time (s)	RTT (s)
GAE	7263.1 ± 1736.29	0.187 ± 0.022	12.41 ± 2.078
EC2	2169.7 ± 133.30	2.843 ± 0.144	15.52 ± 0.577
Euca	4134.2 ± 336.21	1.533 ± 0.136	15.61 ± 1.031

Table 4 A comparison between GAE, 12-node Cassandra AppScale on EC2, and 12-node Cassandra AppScale on a Eucalyptus cluster under high load.

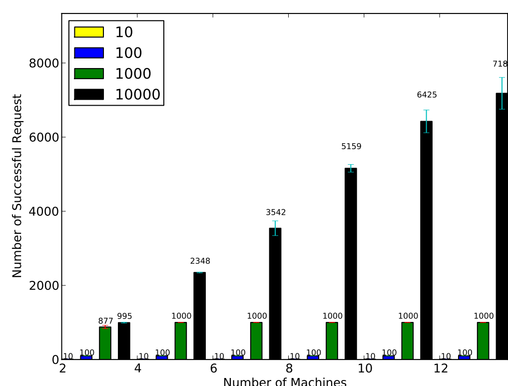
Table 4 compares the two public clouds and our private cloud all under heavy load using the same metrics as we used in the previous graphs (number of successful requests out of 10,000, the average application time per request, and the average round trip time (RTT) per request). The data shows that GAE scales very well, contrary to previous results [23], due to upgrades since it was last benchmarked. GAE has multiple advantages compared to our EC2 and Eucalyptus deployment. Firstly,



(a) Round trip time of successful transaction-disabled requests.



(b) Application time of successful transaction-disabled requests.



(c) Number of successful transaction-disabled requests.

Fig. 9 Cassandra results with transactions disabled.

GAE does not operate using virtualization (virtual machines). Second, while our AppScale deployments are 12 nodes, GAE is capable of leveraging far more machines within the capacity of Google's massive infrastructure. GAE is also finely tuned to use Megastore and BigTable, whereas AppScale trades off potential performance gains by adding an abstraction layer for flexibility in the backend.

Our private Eucalyptus cluster performs significantly better than a comparable EC2 cluster of 12 nodes under high load with lower application latency and a higher number of successful requests. Our private cluster has a few advantages compared to the EC2 cluster in that it is in a non-shared environment and there are no other users of this cluster. Disk bandwidth and network bandwidth are not in contention in our private cluster, whereas in EC2 the virtual machines performance may vary depending on the level of provisioning onto the physical machines. Moreover, the nodes are all located on a single rack, whereas for EC2 machines are placed on different racks as available [37].

7 Related Work

Distributed transactions, two-phase locking, and multiversion concurrency control (MVCC) have been employed in a multitude of distributed systems since the distributed transaction process was defined in [3]. Our design is based on MVCC and uses versioning of data entities. Google App Engine's implementation of transactions uses optimistic currency control [41], which was first presented by Kung et al. in 1981 [24].

There are two systems closely related to our work that provide a software layer implementing transactional semantics over top of distributed datastore systems. They are Google's Percolator [34] and Megastore [2]. Percolator is a system, proprietary to Google, that provides distributed transaction support for the BigTable datastore. The system is used by Google to enable incremental processing of web indexes. Megastore is the most similar

to our system as it is used directly by Google App Engine for transactions and for secondary indexing. Our approach is database agnostic and not tied to any particular datastore. Prior approaches tightly couple transaction support to the database. DAT can be used for any key/value store and, with AppScale, provide scale, fault tolerance, and reliability with an open source solution. Moreover, our system is platform agnostic as well (running in/on Eucalyptus, OpenStack [33], EC2, VMWare, Xen [42], and KVM [25]) while automatically installing and configuring a datastore and the DAT layer for any given number of nodes.

Cloud TPS [40] provides transactional semantics over key spaces in datastores such as HBase. Cloud TPS achieves high throughput because its design is based heavily in in-memory storage. Replication is done across nodes in memory, and the system will periodically flush the data to a persistence layer such as S3 or another cloud storage. DAT differs from Cloud TPS providing higher durability because DAT requires each write to be written to disk. In the case of system wide outages, it is possible to lose all transactions which have not been persisted with Cloud TPS, while in DAT all writes are written to a journal which is replicated on disk at multiple nodes.

In [23] Kossman et al. compared different clouds and datastores, one of which is GAE. GAE has improved over time so the results, while valid at that point in time, are no longer valid. The same can be said for the other clouds which were benchmarked, as each system has evolved over time.

8 Conclusions

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

We extend this layer to provide distributed ACID transaction semantics to applications – that is independent, AKA agnostic, of any particular datastore system and that does not require any modifications to the datastore systems that plug into our cloud portability layer. These semantics allow applications to update atomically multiple key-value pairs programmatically. We refer to this extension as DAT for database-agnostic transactions. Since no open source datastore today provide such semantics, this layer facilitates their use by new applications and application domains including those from the business, financial, and data analytic communities, that depend upon such semantics. We implement this layer within the open source AppScale cloud platform. Our system (including all databases) is available from `http://appscale.cs.ucsb.edu`.

We empirically evaluate the overhead and scalability of this layer using a number of popular database technologies and different cloud systems. We find that the portability layer adds approximately 7-8 ms to puts/gets (reads/writes) due to load balancing across datastore entry points in the cluster. We find that DAT adds an additional 16 ms on average across datastores to each put/write for lock acquisition and journaling (and no additional overhead to get/read operations). We find that the scalability, throughput, and latency of transactional applications (e.g. a financial exchange) using our system varies significantly across datastores but that the Cassandra datastore performs best. Finally, we find that DAT with the Cassandra datastore performs better and is more scalable than native MySQL cluster transaction support.

9 Acknowledgements

This work was funded in part by Google, IBM, and the National Science Foundation (CNS/CAREER-0546737, CNS-0905273, and CNS-0627183).

References

1. Amazon Web Services home page. `http://aws.amazon.com/`.
2. J. Baker, C. Bond, J. Corbett, J. Furman, A. K. J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, January 2011.
3. P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.
5. 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.

6. C. Bunch, N. Chohan, C. Krintz, and K. Shams. Neptune: a domain specific language for deploying hpc software on cloud platforms. In *International Workshop on Scientific Cloud Computing*, pages 59–68, 2011.
7. M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.
8. Cassandra. `http://cassandra.apache.org/`.
9. T. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live - An Engineering Perspective. In *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*, 2007.
10. 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.
11. F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. *Proceedings of 7th Symposium on Operating System Design and Implementation(OSDI)*, page 205218, 2006.
12. N. Chohan, C. Bunch, Y. Nomura, and C. Krintz. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing*, 2011.
13. B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.
14. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Symposium on Operating System Principles*, 2007.
15. ejabberd. `http://ejabberd.im`.
16. Eucalyptus home page. `http://eucalyptus.cs.ucsb.edu/`.
17. Hadoop Distributed File System. `http://hadoop.apache.org`.
18. Hadoop MapReduce. `http://hadoop.apache.org/`.
19. HAProxy. `http://haproxy.1wt.eu`.
20. HBase. `http://hadoop.apache.org/hbase/`.
21. Hypertable. `http://hypertable.org`.
22. G. Kola, T. Kosar, and M. Livny. Faults in large distributed systems and what we can do about them. *Lecture Notes in Computer Science*, 3648:442–453, 2005.
23. D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *International Conference on Management of Data*, pages 579–590, 2010.
24. H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.
25. Kernel based virtual machine. `http://www.linux-kvm.org/`.
26. L. Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems*, 1998.
27. "memcached". `http://memcached.org`.
28. MemcacheDB. `http://memcachedb.org/`.
29. MongoDB. `http://mongodb.org/`.
30. MySQL Cluster. `http://www.mysql.com/cluster`.
31. Nginx. `http://www.nginx.net`.
32. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *"9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)"*, May 2009.
33. OpenStack. `http://openstack.org`.
34. D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Symposium on Operating System Design and Implementation*, 2010.
35. Protocol Buffers. Google's Data Interchange Format. `http://code.google.com/p/protobuf`.
36. Redis. `http://redis.io`.
37. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.
38. SimpleDB. `http://aws.amazon.com/simpledb/`.
39. Voldemort. `http://project-voldemort.com/`.
40. Z. Wei, G. Pierre, and C.-H. Chi. Scalable transactions for web applications in the cloud. In *Proceedings of the Euro-Par Conference*, Delft, The Netherlands, Aug. 2009. `http://www.globule.org/publi/STWAC_europar2009.html`.
41. What is Google App Engine? `http://code.google.com/appengine/docs/whatisgoogleappengine.html`.
42. XenSource. `http://www.xensource.com/`.
43. ZooKeeper. `http://hadoop.apache.org/zookeeper`.