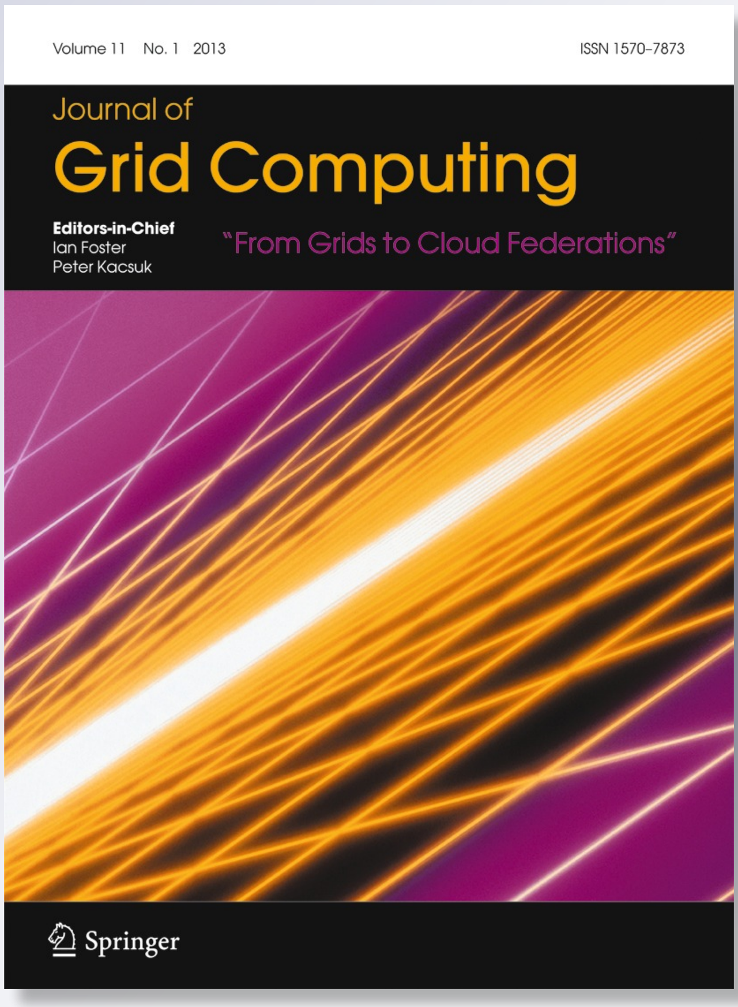
Cloud Platform Datastore Support

Navraj Chohan, Chris Bunch, Chandra Krintz & Navyasri Canumalla

Journal of Grid Computing
From Grids to Cloud Federations

ISSN 1570-7873
Volume 11
Number 1

J Grid Computing (2013) 11:63-81
DOI 10.1007/s10723-012-9238-z



J Grid Computing (2013) 11:63–81
DOI 10.1007/s10723-012-9238-z

Cloud Platform Datastore Support

Navraj Chohan · Chris Bunch · Chandra Krintz · Navyasri Canumalla

Received: 26 August 2011 / Accepted: 13 September 2012 / Published online: 14 October 2012

Abstract There are many datastore 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 their 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 datastore 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 application 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, OpenStack, and Amazon EC2). We use this system to investigate the impact of extending disparate datastores via the application portability layer with distributed transaction support.

Keywords Cloud computing · Cloud platform · Datastore · Key-value structured storage · Online transaction processing

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

N. Chohan · C. Bunch · C. Krintz (✉) · N. Canumalla
Computer Science Department, University of California, Santa Barbara, Santa Barbara, CA, USA
e-mail: ckrintz@cs.ucsb.edu

N. Chohan
e-mail: nchohan@cs.ucsb.edu

C. Bunch
e-mail: cgb@cs.ucsb.edu

N. Canumalla
e-mail: navyasri@cs.ucsb.edu

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 available on-demand, 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 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) [8, 10, 11, 13, 14, 20, 21, 28, 36, 38]. These systems support 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.) is in 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 [42], an open-source distributed directory service that maintains consistent and highly available access to metadata using a variant of the Paxos algorithm [9, 26].

Transaction are crucial for correctness of many applications. An example of a procedure which requires a transaction is the updating of a counter where a value is first read and then incremented or decremented. If multiple updates occur to the counter it is plausible that an update maybe lost or the update is not seen in the case of eventual consistency. The case for atomic and transactional updates becomes more apparent when the counters are critical entities which represent monetary values. In business critical procedures the overhead of ACID transactions are small compared to how important of a feature it is.

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 [16, 32]) and emulates Google App Engine functionality, developers are given the freedom to choose the infrastructure on which their application runs on, providing far reaching application portability.

In the sections that follow, we first present related work in Section 2, and then 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 describe how we extend this layer with ACID transaction semantics in a database-agnostic fashion in Section 4. We then present an evaluation of the system using different datastores in Section 7, and conclude in Section 8.

2 Related Work

Distributed transactions, two-phase locking, and multi-version 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 [40], which was first presented by Kung et al. in 1981 [25].

There are two systems closely related to our work that provide a software layer implementing transactional semantics on 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. Our datastore-agnostic transactions (DAT) system 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 [41], and KVM [22]) while automatically installing and configuring a datastore and the DAT layer for any given number of nodes.

Cloud TPS [39] provides transactional semantics over key spaces in datastores such as HBase. Cloud TPS achieves high throughput because its design is based heavily on 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 [24] 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. Likewise, any results given in this paper is also a snapshot in time for any given technology.

3 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 in 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 (see Table 1). 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. AppScale emulates this cloud platform functionality using private/local virtualized clusters and/or infrastructure-as-a-service (IaaS) systems such as Amazon EC2 and EUCALYPTUS.

Table 1 Google App Engine APIs

Name	Description
Datastore	Schemaless object storage
Memcache	Distributed caching service
Blobstore	Storage of large files
Channel	Long lived JavaScript connections
Images	Simple image manipulation
Mail	Receiving and sending email
Users	Login services with Google accounts
Task Queues	Background tasks
URL Fetch	Resource fetching with HTTP request
XMPP	XMPP-compatible messaging service

AppScale can execute GAE applications without white-list library 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 AppScale restarts any service that stops functioning correctly. An AppScale cloud consists of a set of virtual machine instances (nodes) working together in a distributed system, each of which implement this software stack.

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)

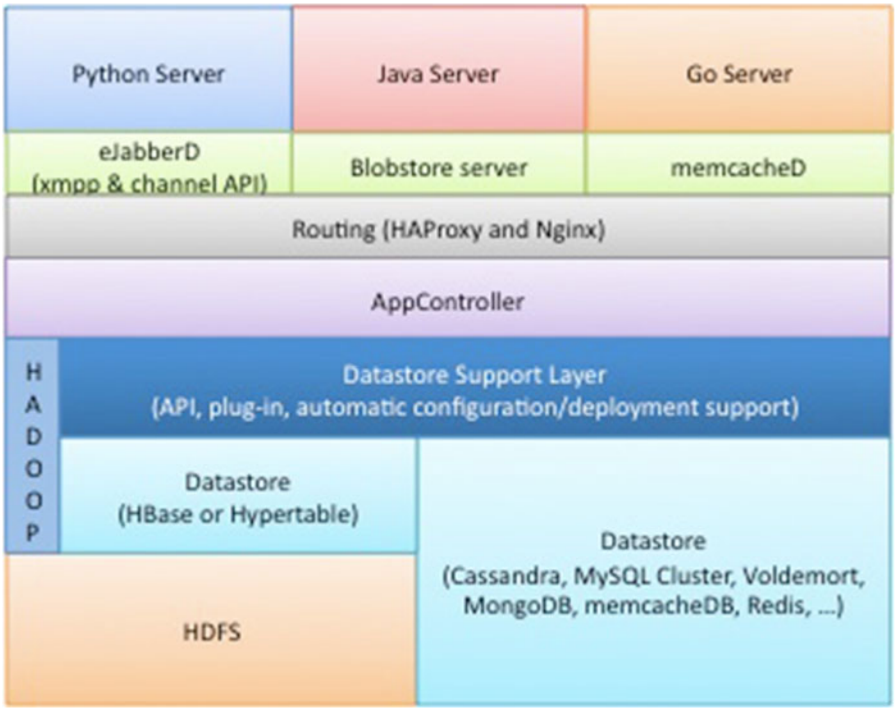

Python Server
Java Server
Go Server
eJabberD (xmpp & channel API)
Blobstore server
memcacheD
Routing (HAProxy and Nginx)
AppController
HADOOP
Datastore Support Layer (API, plug-in, automatic configuration/deployment support)
Datastore (HBase or Hypertable)
HDFS
Datastore (Cassandra, MySQL Cluster, Voldemort, MongoDB, memcacheDB, Redis, ...)

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

– 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 [40]. 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 [38], MongoDB [29], SimpleDB [37], and MySQL Cluster [30]. HBase and Hypertable both rely on HDFS [17] for their distributed file system implementations, as does the Map-Reduce 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 Fig. 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.

4 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.

4.1 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 [40]. 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.

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.

4.2 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 either 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 trade-offs. 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.

5 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 locking 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.

5.1 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 [42], an open source, distributed locking service that maintains consistent copies of data in a distributed setting via the Paxos algorithm [9, 26]. ZooKeeper is the open source equivalent to Google's Chubby locking service [7] which is fault tolerant and provides strong consistency for the data it stores. 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 s (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 either 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.

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, ongoing 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 Fig. 2 and show the DTC API being used during the lifetime of a transaction where two *put* operations take place. It illustrates the

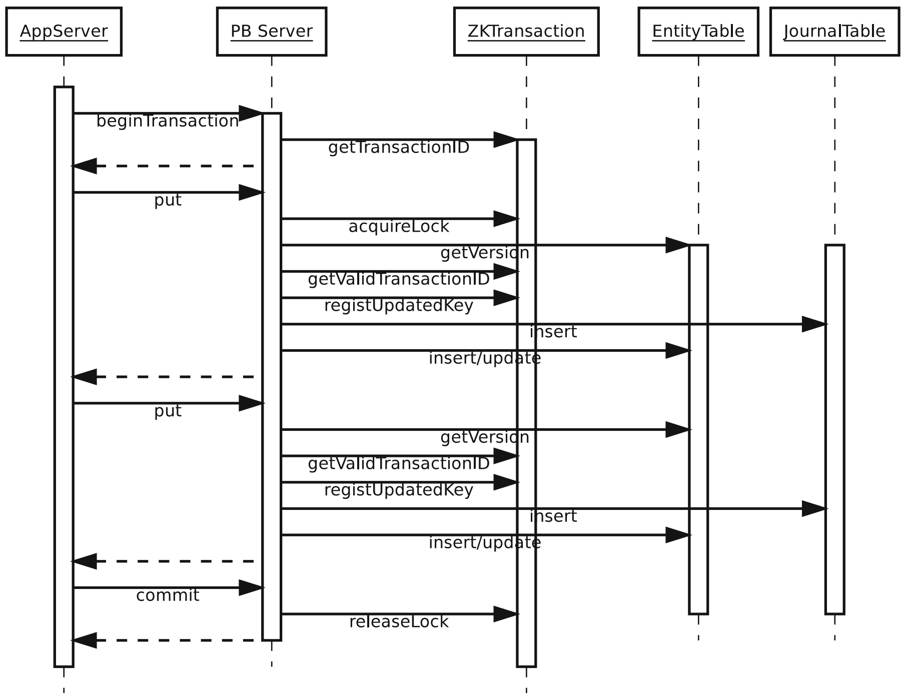


Fig. 2 Transaction sequence example for two puts

transaction starting where a transaction ID is attained; a *put* request then triggers the acquisition of a lock, version validation, key registration for rollback, and entity updates. The second *put* repeats the same steps sans lock acquisition. Lastly, the transaction is committed.

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.

5.2 ZooKeeper Configuration of the DTC

We present the DTC implementation using the ZooKeeper node structure prefix tree (trie) in Fig. 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). State of ZooKeeper is shared among clients, showing a strongly consistent view. 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).

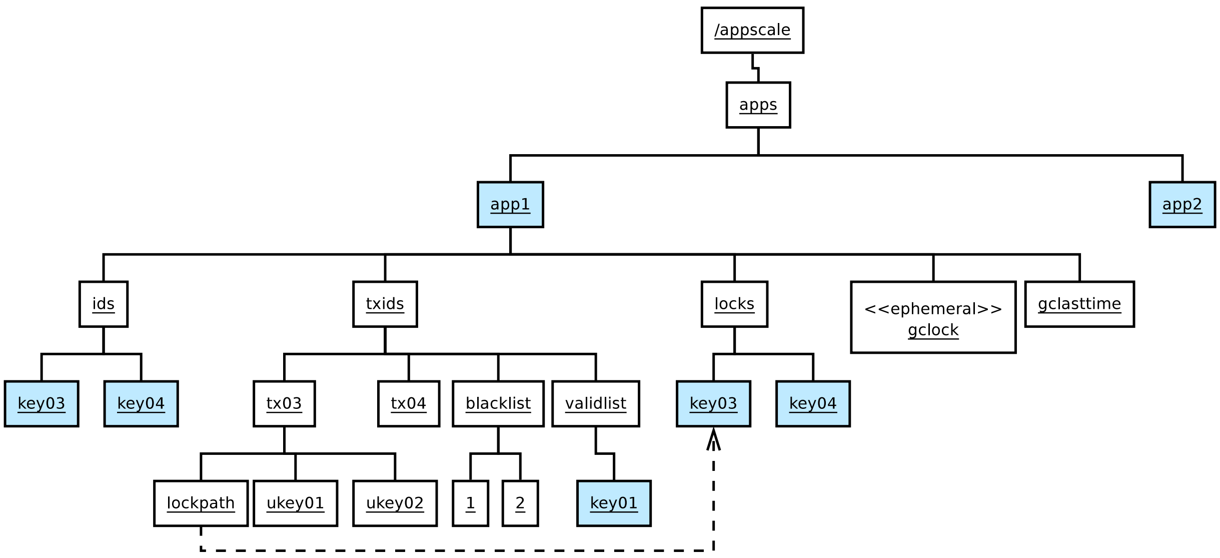


Fig. 3 Structure of transaction metadata in ZooKeeper nodes

5.3 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 (Fig. 3) 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 *key*03 and *key*04 nodes hold values for the next batch of child keys.

5.4 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 s on each lock lease acquired by applications. 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 30 s. 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 s 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.

5.5 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") [23].

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.

6 Methodology

In this section, we overview our benchmarks and experimental methodology. For our experiments, AppScale employs Hadoop 0.20.2-cdh3u3, HBase 0.90.4-cdh3u3, Hypertable 0.9.5.5, MySQL ndb-7.0.9, Redis 2.2.11, and Cassandra 1.0.7. 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 also employ our benchmark on Google App Engine, where the infrastructure is abstracted away. We synchronize the clocks using the Linux tool `ntpdate` for our Eucalyptus cluster.

6.1 Benchmarking Application

Our benchmark measures reads and writes of each datastore where transactions are enabled as well as disabled, the difference of which gives us the overhead imposed by the DAT layer. AppScale is configured to have the head node act as a full proxy, randomly distributing request across application servers. The benchmark is run for a single node deployment (it acts as both a load balancer and runs application servers), the default four node deployment, and a 12 node deployment.

We use the Apache Benchmark tool as our load generator, which targets a URL at the head node. The datastore is first primed with 1000 entries, for which random reads are done on. The writes use random keys, but each key always starts with the application name for isolation and lexicographical entity placement.

The Apache Benchmark tool is used with three different load levels: 10, 100, and 1000 concurrent requests. The tool measures latency and throughput for these different loads. Reported numbers are averages of 10 trials.

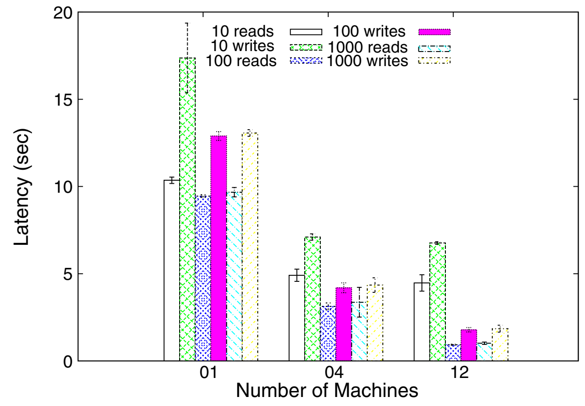
Each server runs ten process instances of the benchmark application. We set the replication factor to one for these experiments for all datastores, which was the common factor given our one node deployment. For GAE, Google uses its own scheduling and replication policy to enable the scaling of applications, and it is unknown how many physical servers are being employed.

7 Results

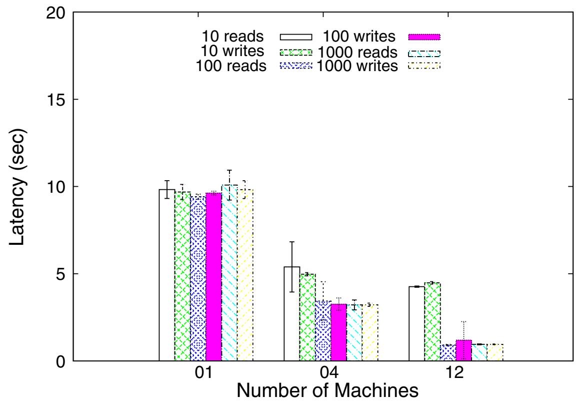
Figure 4 shows measurements for the Cassandra datastore with a varying number of machines, for writes and reads of a concurrency level of 10, 100, and 1000, with and without transaction support. Figure 4a and b chart latency of requests with and without transactions enabled. For all sizes of clusters we see additional latency for writes, regardless of the concurrency level when transactions are enabled. However, reads see no statistically significant overhead when enabled. Latency for reads and writes are in close range to each other when transactions are disabled, but the overhead of transactions for writes causes asymmetrical latency. Moreover, latency drops as more nodes are added to the cluster with and without transactions.

Figure 4c has throughput of requests, while Fig. 4d has the same but with transactions disabled. Writes have less throughput when transactions are enabled, while read throughput is unchanged. With the lower latency of additional nodes the throughput rises.

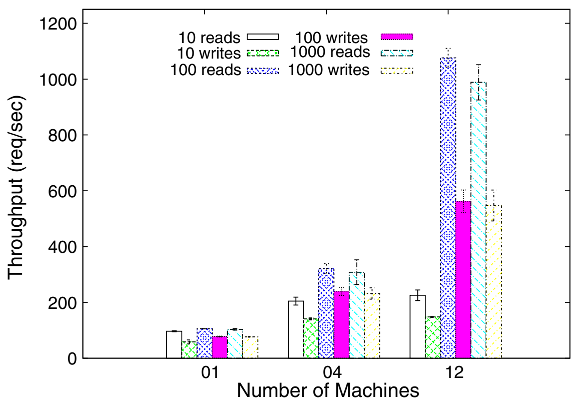
HBase latency is shown in Fig. 5a and b, for transactions enabled and disabled, respectively. Compared to Cassandra, HBase performs similarly for latency, yet for the 12 node case, Cassandra is able to get higher throughput for reads and writes as seen in Fig. 5c and d. Both datastores see similar drop offs in throughput due to the



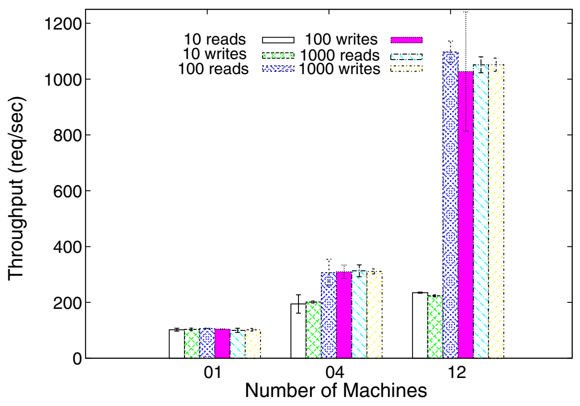
(a) Latency for Cassandra with transactions enabled.



(b) Latency for Cassandra with transactions disabled.



(c) Throughput for Cassandra with transactions enabled.



(d) Throughput for Cassandra with transactions disabled.

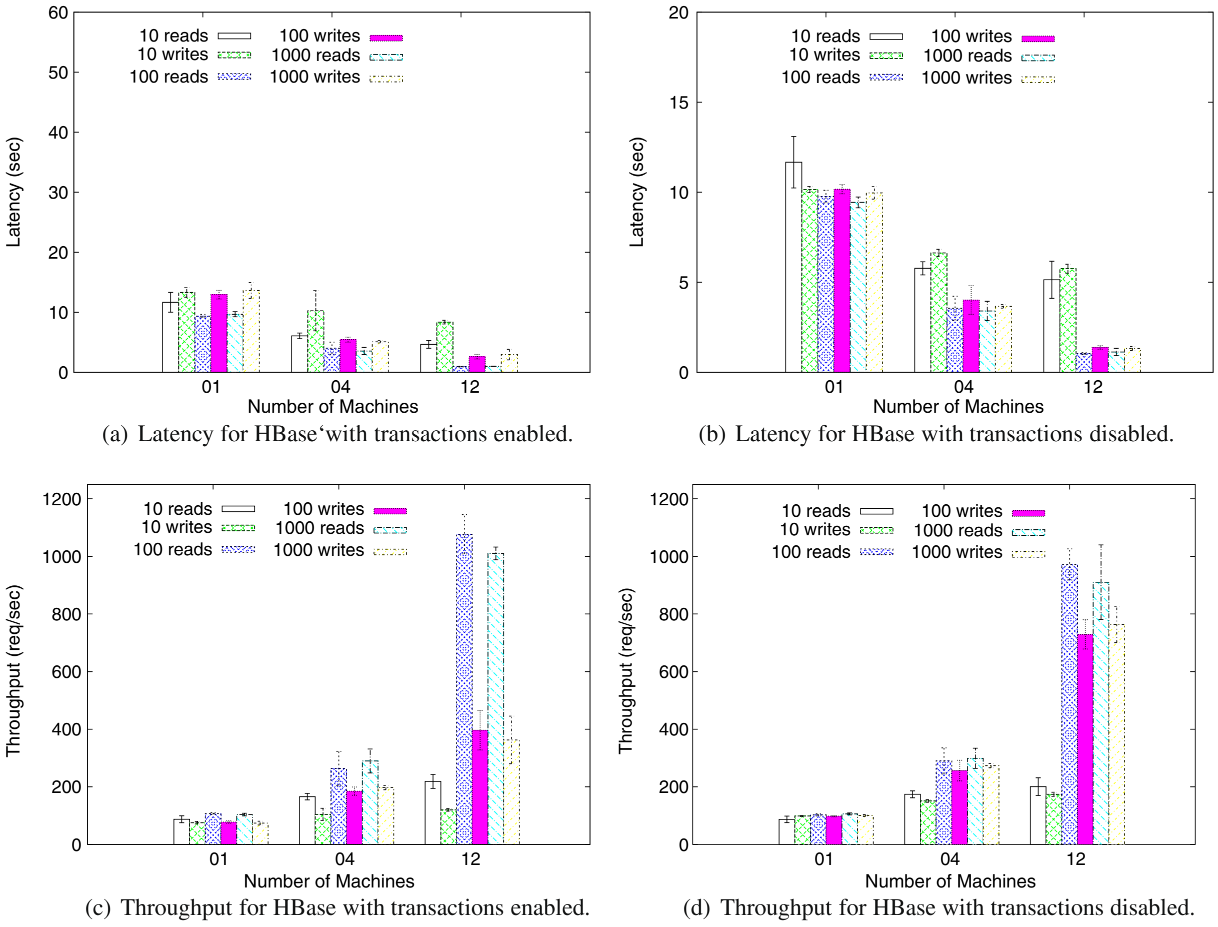
Fig. 4 Cassandra results as the number of machines increases

overhead of transaction support, yet Cassandra sees more than 100 more request per second in throughput compared to HBase when transactions are disabled.

Hypertable has slightly more latency for the single node case for both reads and writes as presented in Fig. 6a and b, yet for higher node counts it is comparable to HBase. Hypertable achieves high throughput for reads with over 1000 requests per second as seen in Fig. 6c and d. Hypertable, compared to HBase, gets more read and write throughput, where for high load it is over 1000 requests per second.

Redis performance numbers are presented in Fig. 7, where we see some deviation from the previously presented datastores. Redis stores data in memory (asynchronously writing to disk which loosens our consistency guarantees for ACID semantics) and does so in the master node which handles all requests. Slaves store copies of the master, yet in these experiments we set replication to one. Where the previous datastores are able to have clients and scale with larger deployments, Redis does not benefit because all request go to a single node causing saturation of the node more quickly, and hence the lower performance in throughput.

The MySQL Cluster deployment does not use the DAT layer for transaction support, but rather its own native implementation. Figure 8a shows latency numbers for reads and writes. The latency is much higher than previous datastores, along with higher variance. As more nodes are added, the latency does drop, but even at 12 nodes it is over 10 s for writes. Reads scale much better for 12 nodes, but with high variance.



(a) Latency for HBase‘with transactions enabled.

(b) Latency for HBase with transactions disabled.

(c) Throughput for HBase with transactions enabled.

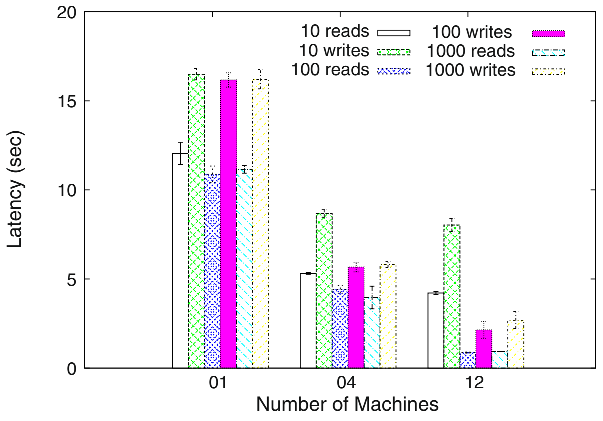
(d) Throughput for HBase with transactions disabled.

Fig. 5 HBase results as the number of machines increases

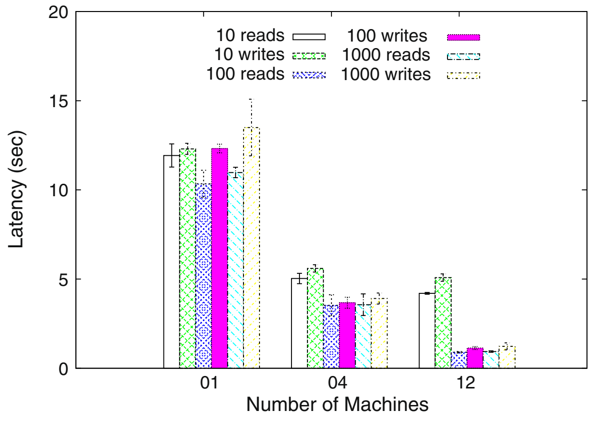
Figure 9b shows a CDF of latency of writes and reads for transactions disabled for Cassandra on a 12 node deployment, while Fig. 9a shows the same with transactions enabled. The overhead can be seen with 1000 writes, where latency is much higher. Reads do no see the same overhead, where the latency stays the same. When transactions are disabled, writes are faster than reads until the 90th percentile, where writes have a longer tail for latency.

The breakdown of an entity *put* for Cassandra is presented in Fig. 10. Each operation which is a part of the transaction adds some amount of overhead, where the "Puts" are parallel writes to the entity table and journal table. The majority of overhead comes from checking to see if the current key exists and, if it does, to register that transaction value for any required rollback. This figure measures it for the case where the key did not exist before, which for Cassandra is a higher latency operation than looking for a key which does exist. It should be noted that this is the highest amount of relative overhead because this looks at only a single *put*. If the transaction had multiple reads and writes, then much of the overhead associated with starting a transaction, getting a lock, releasing it, and committing is amortized.

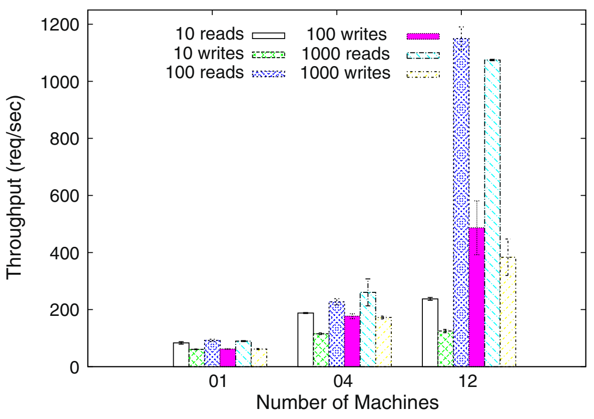
For comparison purposes we also ran the benchmark on GAE. Figure 11a has the latency of gets and writes for different concurrency levels. Figure 11b has throughput for the same experiment, showing much less throughput than our scalable datastores, yet in these cases the round trip time is much higher (a ping averaged 27ms to our application hosted by Google) as our load generator is local to our private cluster. Round



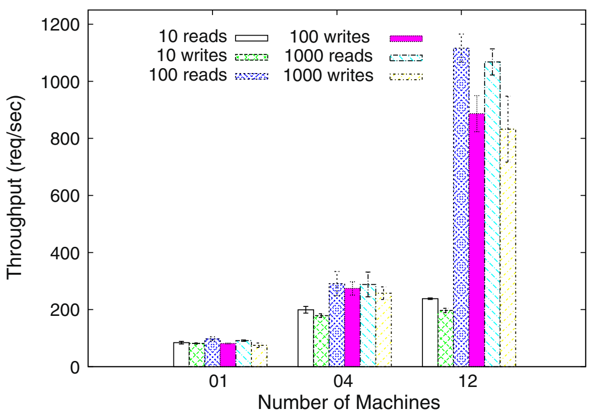
(a) Latency for Hypertable‘with transactions enabled.



(b) Latency for Hypertable with transactions disabled.



(c) Throughput for Hypertable with transactions enabled.



(d) Throughput for Hypertable with transactions disabled.

Fig. 6 Hypertable results as the number of machines increases

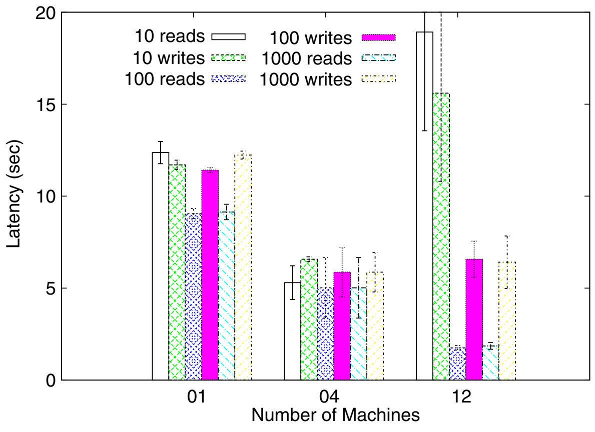
trip time for our local tests, by comparison, where sub 1ms.

7.1 Discussion

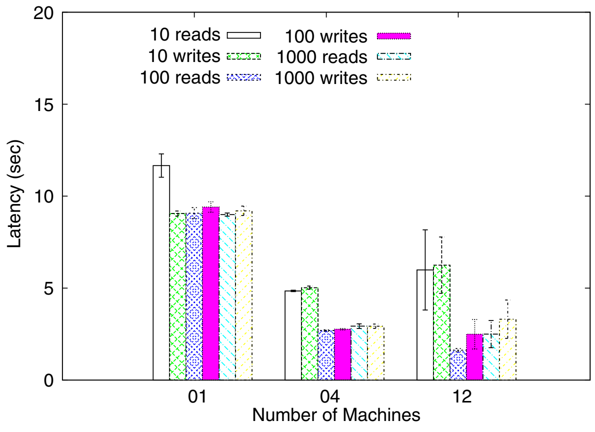
We find that Cassandra, HBase, and Hypertable were the most scalable, all being BigTable clones. Cassandra performed best in our study, followed by Hypertable, and then HBase. Although Cassandra is able to do placement using random partitioning for range query support it requires lexicographical partitioning. Because data is placed based on key names, we see that data is stored on a single node until a tablet server is split and stored on another node. If the data set is based out of one tablet we see certain nodes can become hotspots causing slowdown if the number of clients becomes very large. Larger scale deployments were attempted, but due to the aforementioned placement of data, we found additional nodes saw no improvement in terms of throughput.

MySQL Cluster had much higher latency and lower throughput than the NoSQL stores. MySQL is at a disadvantage as it is not aware of the entity group abstraction. Hence, it uses course grain locks which limit the throughput of updates.

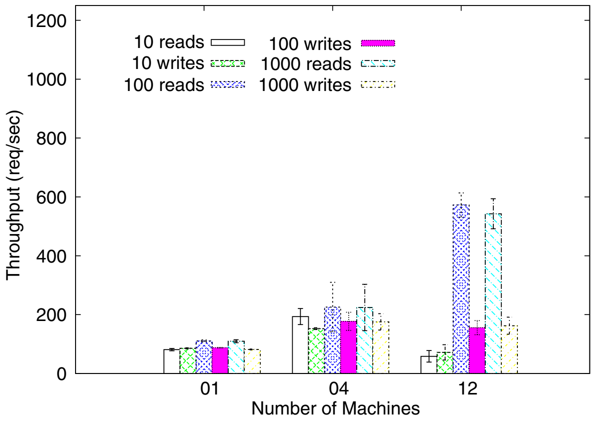
Throughput of transactions for *puts* does drop by as much as 50 % compared to transactions being disabled. AppScale allows developers who do not require transaction semantics, and hence the required overhead, to disable them by the use of namespaces. Any namespace which preprends *notrans* will give direct access to the datastore, where access to DAT is circumvented. Any



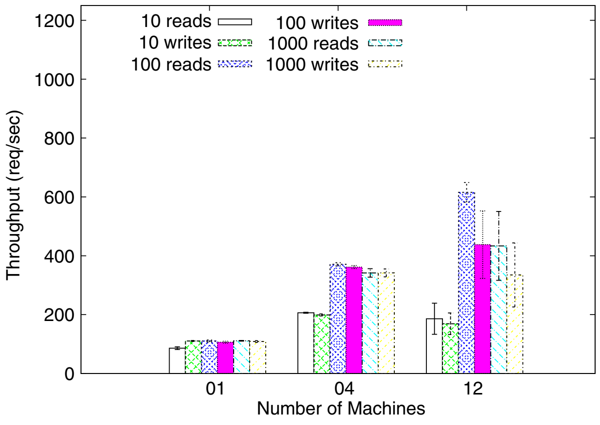
(a) Latency for Redis‘with transactions enabled.



(b) Latency for Redis with transactions disabled.

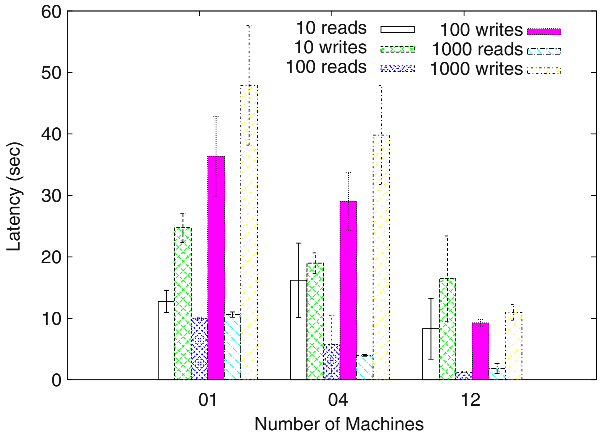


(c) Throughput for Redis with transactions enabled.

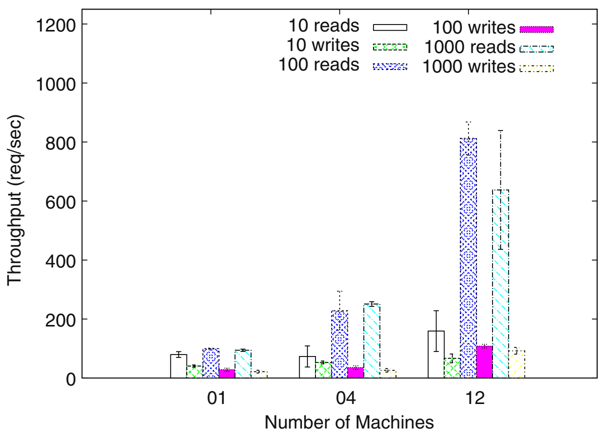


(d) Throughput for Redis with transactions disabled.

Fig. 7 Redis results as the number of machines increases

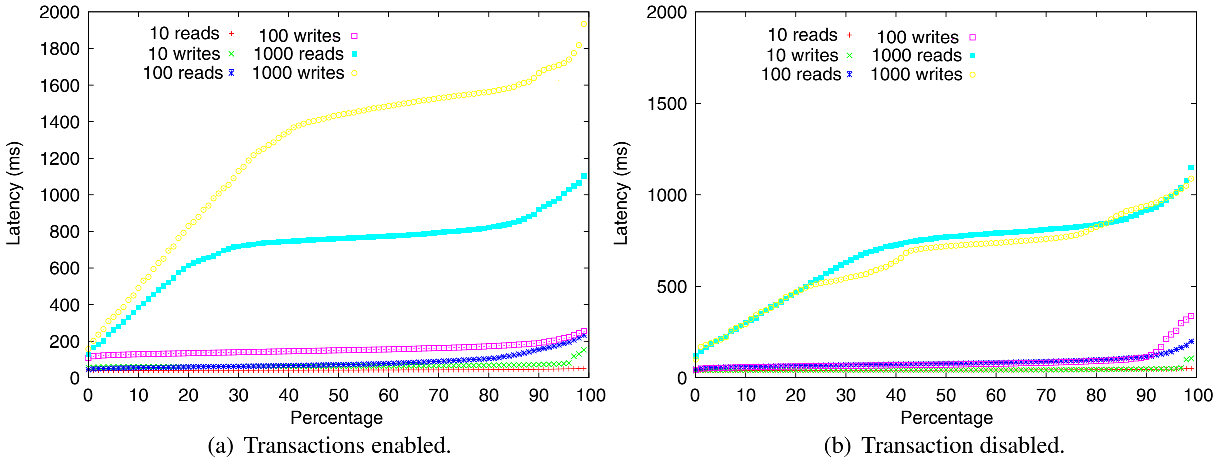


(a) Latency for MySQL with native transactions.



(b) Throughput for MySQL with native transactions.

Fig. 8 MySQL results as the number of machines increases



(a) Transactions enabled. (b) Transaction disabled.

Fig. 9 Latency CDFs for Cassandra 12 nodes for reads and writes

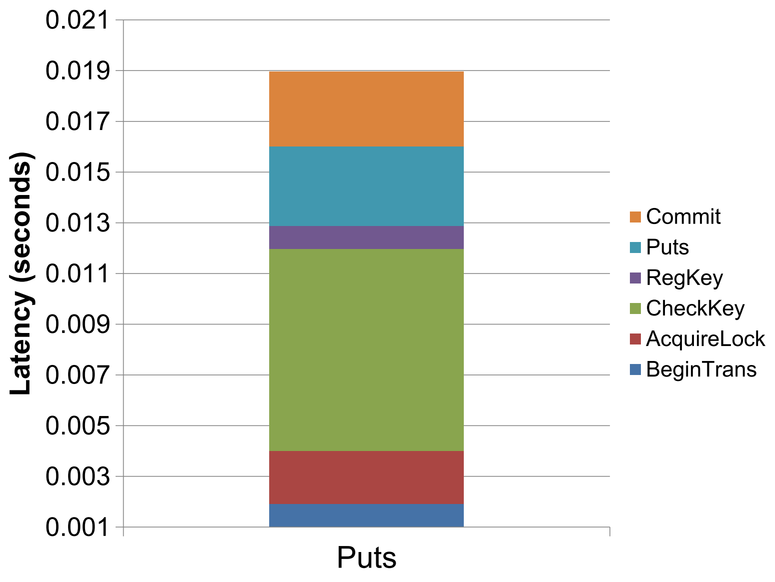
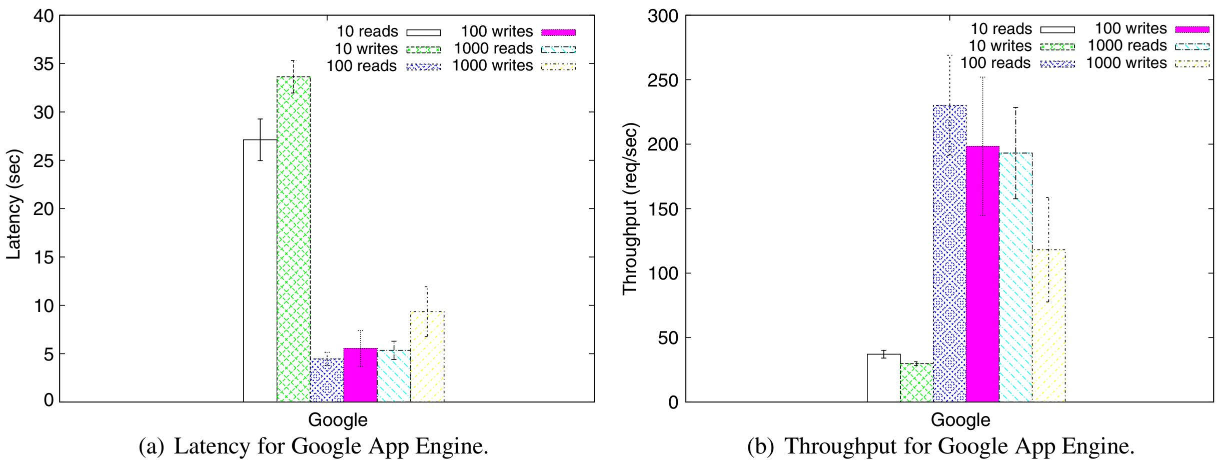


Fig. 10 Time breakdown of an entity *put*



(a) Latency for Google App Engine. (b) Throughput for Google App Engine.

Fig. 11 Google App Engine results with auto-scaling

application which uses transaction semantics will still work although no ACID correctness guarantees are given.

Read throughput remains unchanged when transactions are enabled, as the only overhead associated with the read is to check to make sure the version of the entity is not from an on going transaction, or a black listed transaction. Many systems and workloads are read heavy, one example of which comes from the Megastore paper [2] which used 20:1 read to write ratios for their evaluations similar to what they see internally at Google.

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, independent and 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.

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/. Accessed 1 Aug 2011
2. Baker, J., Bond, C., Corbett, J., Furman, J., Larson, A.K.J., Leon, J., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: providing scalable, highly available storage for interactive services. In: Conference on Innovative Data Systems Research (CIDR), pp. 223–234 (2011)
3. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. **13**(2), 185–221 (1981)
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman, Boston (1987)
5. Bunch, C., Chohan, N., Krintz, C., Chohan, J., Kupferman, J., Lakhina, P., Li, Y., Nomura, Y.: An evaluation of distributed datastores using the AppScale cloud platform. In: IEEE International Conference on Cloud Computing (2010)
6. Bunch, C., Chohan, N., Krintz, C., Shams, K.: Neptune: a domain specific language for deploying hpc software on cloud platforms. In: International Workshop on Scientific Cloud Computing, pp. 59–68 (2011)
7. Burrows, M.: 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/. Accessed 1 Aug 2011
9. Chandra, T., Griesemer, R., Redstone, J.: Paxos made live—an engineering perspective. In: PODC '07: 26th ACM Symposium on Principles of Distributed Computing (2007)
10. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: a distributed storage system for structured data. In: Symposium on Operating System Design and Implementation (2006)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: a distributed storage system for structured data. In: Proceedings of 7th Symposium on Operating System Design and Implementation (OSDI), pp. 205–218 (2006)
12. Chohan, N., Bunch, C., Nomura, Y., Krintz, C.: Database-agnostic transaction support for cloud in-

frastructures. In: IEEE International Conference on Cloud Computing (2011)

13. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. **1**(2), 1277–1288 (2008)
14. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Symposium on Operating System Principles (2007)
15. ejabberd: http://ejabberd.im. Accessed 1 Aug 2011
16. Eucalyptus home page: http://eucalyptus.cs.ucsb.edu/. Accessed 1 Aug 2011
17. Hadoop Distributed File System: http://hadoop.apache.org. Accessed 1 Aug 2011
18. Hadoop MapReduce: http://hadoop.apache.org/. Accessed 1 Aug 2011
19. HAProxy: http://haproxy.1wt.eu. Accessed 1 Aug 2011
20. HBase; http://hadoop.apache.org/hbase/. Accessed 1 Aug 2011
21. Hypertable: http://hypertable.org. Accessed 1 Aug 2011
22. Kernel based virtual machine: http://www.linux-kvm.org/. Accessed 1 Aug 2011
23. Kola, G., Kosar, T., Livny, M.: Faults in large distributed systems and what we can do about them. Lect. Notes Comput. Sci. **3648**, 442–453 (2005)
24. Kossmann, D., Kraska, T., Loesing, S.: An evaluation of alternative architectures for transaction processing in the cloud. In: International Conference on Management of Data, pp. 579–590 (2010)
25. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981)
26. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
27. memcached: http://memcached.org. Accessed 1 Aug 2011
28. MemcacheDB: http://memcachedb.org/. Accessed 1 Aug 2011
29. MongoDB: http://mongodb.org/. Accessed 1 Aug 2011
30. MySQL Cluster: http://www.mysql.com/cluster. Accessed 1 Aug 2011
31. Nginx: http://www.nginx.net. Accessed 1 Aug 2011
32. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The eucalyptus open-source cloud-computing system. In: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID) (2009)
33. OpenStack: http://openstack.org. Accessed 1 Aug 2011
34. Peng, D., Dabek, F.: 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. Accessed 1 Aug 2011
36. Redis: http://redis.io. Accessed 1 Aug 2011
37. SimpleDB: http://aws.amazon.com/simpledb/. Accessed 1 Aug 2011
38. Voldemort: http://project-voldemort.com/. Accessed 1 Aug 2011
39. Wei, Z., Pierre, G., Chi, C.-H.: Scalable transactions for web applications in the cloud. In: Proceedings of the Euro-Par Conference, Delft, The Netherlands (2009). http://www.globule.org/publi/STWAC_europar2009.html. Accessed 1 Aug 2011
40. What is Google App Engine? http://code.google.com/appengine/docs/whatisgoogleappengine.html. Accessed 1 Aug 2011
41. XenSource: http://www.xensource.com/. Accessed 1 Aug 2011
42. ZooKeeper: http://hadoop.apache.org/zookeeper. Accessed 1 Aug 2011