

# Database-Agnostic Transaction Support for Cloud Infrastructures

Navraj Chohan  
Chris Bunch  
Chandra Krintz  
Computer Science Department  
University of California, Santa Barbara  
{nchohan,cgb,ckrintz}@cs.ucsb.edu

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

## Abstract

*In this paper, we present and empirically evaluate the performance of database-agnostic transaction (DAT) support for the cloud. Our design and implementation of DAT is scalable, fault-tolerant, and requires only that the datastore provide atomic, row-level access. Our approach enables applications to employ a single transactional datastore API that can be used with a wide range of cloud datastore technologies. We implement DAT in AppScale, an open-source implementation of the Google App Engine cloud platform, and use it to evaluate DAT's performance and the performance of a number of popular key-value stores.*

## 1. Introduction

Technological advances have significantly driven down the cost of off-the-shelf compute power, storage, and network bandwidth. As a result, two key shifts have emerged. The first is cloud computing, a service-oriented computing model that uses operating systems, network, and storage virtualization to allow users and applications to control their resource usage dynamically through a simple programmatic interface, and for users to reason about the capability they will receive across the interface through well-defined Service Level Agreements (SLAs). Using this model, the high-technology sector has been able to make its proprietary computing and storage infrastructure available to the public (or internally via private clouds) at extreme scales.

The second key shift that has resulted from readily available, low-cost, high-performance, and high-density compute capability is that applications have become data-centric and our data resources and products have grown explosively in both number and size. Public and private cloud providers have turned to distributed, highly available, and elastic key-value stores to provide scalable data access [7], [10], [5]. Although highly scalable, one disadvantage of key-value stores is that most offer no support for atomic multi-row updates to their data. Such transactional semantics, however, are vital to a majority of commercial and data-analytic applications that require all-or-nothing updates across multiple data

entities concurrently.

In this paper, we investigate the performance impact of providing a limited form of transactional semantics for key-value datastores. Our approach is unique in that our goal is to do so for a wide range of datastores using a single software layer that is database agnostic. Such a layer can be employed by public/private cloud providers to enable them to select across a wide range of database offerings, to leverage technologies already available in the cloud (e.g. system-wide, distributed locking support) and to decouple (and thus simplify) the implementation of the database. We refer to this layer as DAT for **Database Agnostic Transactions**.

We implement DAT within AppScale [8], an open source implementation of the Google App Engine (GAE) cloud platform. AppScale implements the GAE APIs and provides a private cloud platform and toolset for distributed execution of GAE and web-based applications over virtualized clusters and cloud infrastructures (Amazon EC2 [1] and EUCALYPTUS [17]). AppScale uses the GAE Datastore API to “plug-in” a wide range of open source datastore technologies, including those used by Facebook, PowerSet, Reddit, Baidu, Twitter, and others, to this API. The GAE public cloud implements a single proprietary datastore (BigTable [7]) using this API. GAE has recently provided limited transactional semantics to this datastore via Megastore [2]. Our approach emulates and extends this proprietary implementation within AppScale 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 [22], an open-source locking service used by clouds and other distributed systems.

We employ DAT and AppScale to empirically evaluate the performance impact of employing distributed transactions for different datastore technologies. In particular, we evaluate DAT for HBase [10], Hypertable [11], and Cassandra [5]. We also compare DAT to MySQL

Cluster [16] which provides built-in transactional support (although with more extensive functionality). In summary, with this paper, we contribute:

- An explanation of the transactional programming model in AppScale and GAE,
- An implementation of an ACID software layer for clouds that is database agnostic, and
- A performance evaluation of DAT and a comparison of its use across multiple datastores popular in public and private cloud systems.

In the following sections, we describe how AppScale and DAT provides database abstraction and agnosticism. In Section 2 and Section 3, we detail the design and implementation of DAT, respectively. We then present an evaluation of DAT for different datastores in Section 4, present related work in Section 5, and conclude in Section 6.

## 2. Database Abstraction

DAT is a database-agnostic transaction layer for cloud platforms that facilitates the use of a single API through which different datastore implementations can be instantiated. Such a layer enables application portability (programs can use different datastores with transactional semantics without modification), to compare datastores empirically, and to ensure that transactional semantics are available regardless of whether the underlying datastore provides them. Separating the transactional layer from the datastore also simplifies the datastore implementation.

The trade-off of such a separation of concerns, however, is the level of indirection that such separation requires and the lack of a tight coupling between transactional support and the datastore; both of which may introduce overhead. To evaluate whether it is possible to provide such an abstraction layer while maintaining efficiency and scalability, we implement the DAT layer within the open source cloud platform AppScale [8]. AppScale is an open source implementation and extension of the Google App Engine (GAE) cloud platform. It supports a number of different open source and proprietary distributed database technologies using a similar abstraction for datastore access. We extend this API and AppScale to implement DAT.

The AppScale/GAE Datastore API provides a put-get interface that is similar to those employed by a number of non-relational datastore technologies. The API consists of the *put*, *get*, *delete*, and *query* operations. Queries are limited to filters which can be applied through the use of range queries (start and end keys of indexes) and have a limited subset of functionality from standard SQL. The DAT layer extends this API with support for begin/end transaction (run\_in\_transaction in GAE terminology).

We have implemented this API in a modular and pluggable fashion using a variety of datastore technologies. In this paper, we focus on Cassandra, HBase, Hypertable, and MySQL Cluster. 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 all the others.

The data values that AppScale stores in the datastore are entities (data objects) as defined by GAE [20]. Each entity has a key object; AppScale stores each entity according to its key as a serialized protocol buffer [19].

### 2.1. DAT Assumptions

We make three key assumptions in the design of our DAT layer. First, we assume that each of the underlying datastores provide strong consistency. Each of the AppScale datastores implement a strongly consistent configuration option, which we employ. HBase, Hypertable, and MySQL Cluster are all strongly consistent. Cassandra, which is known for its eventual consistency, is also configurable for strong consistency by dictating how many replicas must be written before confirming the write. 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 updates to a row's columns are all or nothing. 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 [20]. Entity groups consist of one or more entities, all of whom share the same ancestor, which are 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 or ancestor. In addition, entity groups can consist of multiple kinds. A 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. A root entity is an entity without a parent attribute. Alice is a root entity with its attributes being: `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, but it is possible to have a higher level hierarchy of entities as well. The application ID is prepended to each key to enable multitenancy for datastores which do not allow for dynamic table creation and share one key space (i.e., Cassandra as of version 0.68), which is required when new applications are uploaded.

A transactional workflow 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.

## 2.2. DAT Semantics

The DAT layer 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 *put* value) of the entity in the system. The operations *put* or *delete* outside of a programmatic transaction are transparently handled 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.

GAE and our implementation has its 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 set 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 group as it does for GAE, but a transaction must acquire the lock in DAT before moving forward, a restriction GAE does not have. For future work we will do a direct comparison under high load to see the effectiveness of both design decisions. In practice, our locking mechanism has worked well and has given sufficient throughput.

## 3. DAT Implementation

To implement the DAT layer 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. 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.

### 3.1. Distributed Transaction Coordinator

To enable distributed, concurrent, and fault tolerant transactions, the DAT layer 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 [22], an open source, distributed directory service that maintains consistent and highly available access to data using a variant of the Paxos algorithm [15], [6]. The directory service allows for arbitrary paths to be created, where 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)`

The DAT layer 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, the DAT layer obtains a lock from the DTC (`acquireLock`) for the entity group over which the operation is being performed. For each operation, the DAT layer verifies that all entities accessed have valid versions (`getValidTransactionID`). For each *put* or *delete* operation, the DAT layer registers the operation with the DTC. This allows the DTC to keep 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 previous 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), the DAT layer 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 which are issuing 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. The DAT layer 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 the DAT layer 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 keeping your entity groups small as possible to enable scaling (parallelizing access across entity groups) and keeping bottlenecks to a minimum.

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 hence we assign it to 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.

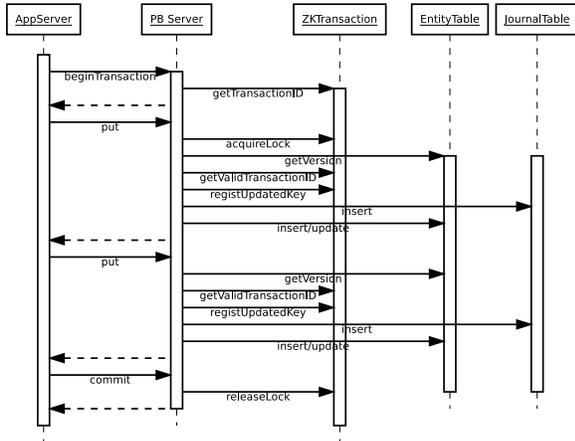


Fig. 1. Transaction sequence example for two puts.

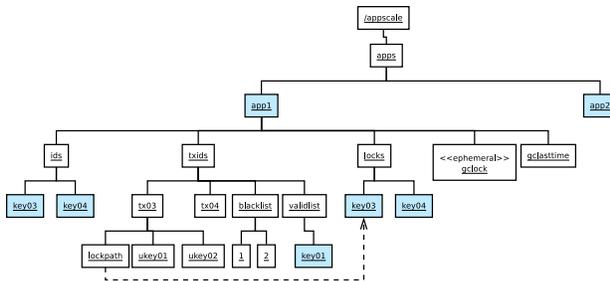


Fig. 2. Structure of transaction metadata in ZooKeeper nodes.

Either using the original version (transaction ID) or the transaction ID returned from the DTC due to invalidation, the DAT layer 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 1 and show the DTC API being used during the lifetime of a transaction.

The DAT layer 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.

**3.1.1. ZooKeeper Configuration of the DTC.** We present the DTC implementation using the ZooKeeper node structure prefix tree (trie) in Figure 2. 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 the DAT layer 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).

**3.1.2. 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, the DAT layer 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 "glock" 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 "glock". 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.

### 3.2. Fault Tolerance

The DAT layer handles certain kinds of failures, excluding byzantine faults. The design of the DTC ensures that the worst case timing scenario does not leave the datastore in an inconsistent state ("Heisenbugs") [12]. 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). These silent faults are detected 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 initialization.

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 configurable for all databases at initiation.

## 4. Evaluation

We next employ AppScale and the DAT layer to evaluate how well different databases perform with transaction support. We first overview our benchmark application and experimental methodology, and then present our results.

For these experiments, AppScale employs Hadoop 0.20.2, HBase 0.89, Hypertable 0.9.4.3, MySQL Cluster

6.3.20, Cassandra 0.6.8. We execute AppScale using a private cluster and the Eucalyptus cloud infrastructure. Our Eucalyptus private cloud consists of 12 virtual machines with 4 cores, and 7.5 GB of RAM. Our clocks are synchronized across the cluster using the Linux tool `ntpdate`.

### 4.1. Benchmarking Application

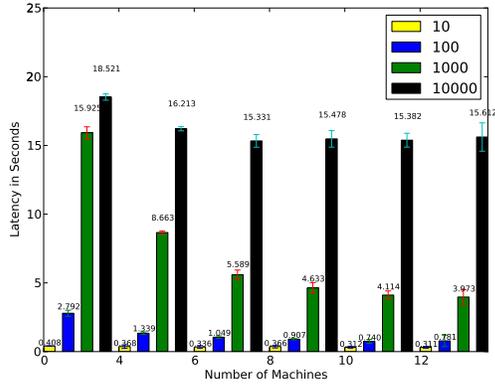
Our testing benchmark is a banking application which transfers money from one account to another. Each run has 100,000 accounts created. A request made to the system has a receiver account number, a sender account number, and the amount to be transferred. The implementation of how such a transfer occurs requires two separate transactional functions for App Engine and AppScale [9]. 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 incompleting transfers. Our results only consider a request a success if both transactions were successful.

Three machines serve to create concurrent request of 10, 100, 1,000, and 10,000. We measure the number of successful transactions and the latency (round trip time and application time) of the successful requests with 10 trials each. The replication factor was set to two and there were 20 application servers per node.

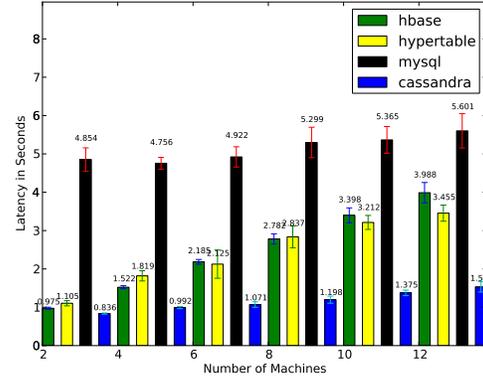
### 4.2. Results

We first show our best performing datastore, Cassandra. Figure 3(a) shows Cassandra's average round trip time (RTT) along with the standard deviation for a varying size workload. The round trip time is the total time taken from the beginning of the HTTP request to when a successful response is received. The graph shows that as additional machines are added the RTT diminishes for all load levels 1,000 and below, while 10,000 sees little improvement in terms of latency due to the machines being overloaded. Figure 3(b) has the number of successful request given a certain workload. For concurrent workloads of 10, 100, and 1,000, we see a high success rate at 4 or more machines. The heaviest load shows linear growth for success rate as machines are added but because the machines are overwhelmed, average latency stays about the same.

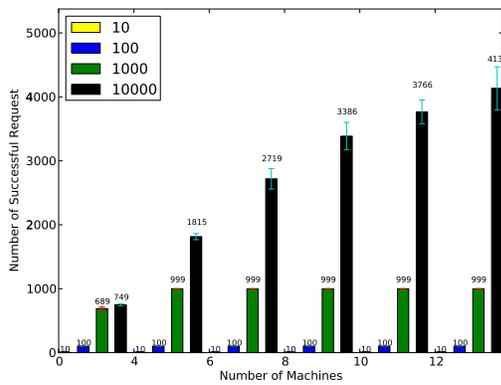
Our next set of graphs show a comparison between different databases. The RTT comparison is not shown due to space constraints, but followed a very similar



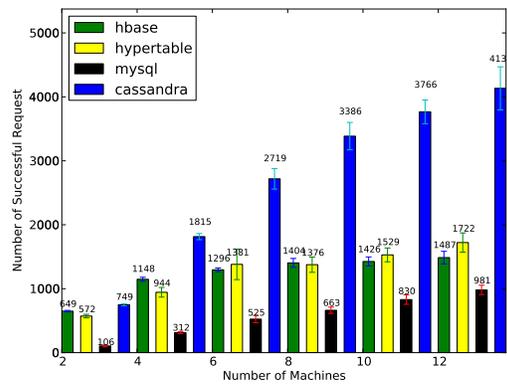
(a) Round trip time of successful transaction requests.



(a) Application request time of successful transaction request.



(b) Number of successful transaction request.



(b) Number of successful transaction request.

Fig. 3. Cassandra results as the number of machines increases.

Fig. 4. A comparison of different datastores as the number of machines increases.

pattern to Figure 3(a). Figure 4(a) shows a comparison of how long each request took to fulfill for a load of 10,000 simultaneous requests. We captured latency at the time when the application server received the request and when it replied (The RTT minus the routing time and queuing delay). MySQL, with its native transaction, performs worse due to its coarse-grain locks, while Cassandra has a latency that is multitudes faster, with Hypertable and HBase in-between. As the number of machines increases, there is more time being spent in the datastore layer, and less in front-end queuing delay. Figure 4(b) compares the datastores with the amount of successful responses with 10,000 concurrent request made and shows high correlation to Figure 4(a). This graph shows that Cassandra performs best, followed by HBase and Hypertable.

MySQL Cluster with its native transaction support does not scale as well as our DAT implementation. Hypertable is also consistent with its response time, but is almost twice as slow as Cassandra. Additional inves-

tigations show that while Hypertable actually has the best performance for gets across all datastores (average of 5ms), it is the slowest of the group in terms of put times (17ms for the 2 node case, and up to 23ms for the 12 node case). HBase's performance levels off as more machines are added, allowing for Hypertable to leap it with 10 nodes or more. Trials without transactions were done, where calls to the DAT layer were stubbed out and journal writes were disabled. The number of successes for Cassandra were on average 7,182 out of 10,000 as seen in Figure 5. This shows that there is significant overhead for enabling transactions because of the ZooKeeper latency, and therefore AppScale allows for the disabling of transaction semantics on a per application basis in return for throughput and latency improvements.

## 5. Related Work

Distributed transactions, two-phase locking, and multiversion concurrency control (MVCC) have been employed in a multitude of distributed systems since the

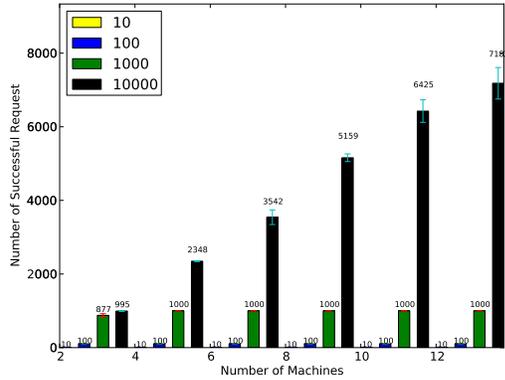


Fig. 5. Cassandra results with transactions disabled.

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 [20], which was first presented by Kung et al. in 1981 [13].

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 [18] 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, EC2, VMWare, Xen [21], and KVM [14]) while automatically installing and configuring a datastore and the DAT layer for any given number of nodes.

## 6. Conclusions

We present an open source implementation of the Google App Engine (GAE) Datastore API with transaction semantics within a cloud platform called AppScale. The implementation unifies access to a wide range of open source distributed database technologies and provides ACID semantics on groups of entities. We describe this implementation and use the platform to empirically evaluate our transactional middleware for a variety of datastores. Our system (including all databases) is available as a virtual machine image at

<http://appscale.cs.ucsb.edu>.

## 7. 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. <http://aws.amazon.com/>.
- [2] J. Baker, C. Bondç, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In G. Weikum, J. Hellerstein, and M. Stonebraker, editors, *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] Cassandra. <http://cassandra.apache.org/>.
- [6] 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.
- [7] 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.
- [8] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open App Engine Application Development and Deployment. In *International Conference on Cloud Computing*, Oct. 2009.
- [9] Distributed Transactions in App Engine. <http://blog.notdot.net/2009/9/Distributed-Transactions-on-App-Engine>.
- [10] HBase. <http://hadoop.apache.org/hbase/>.
- [11] Hypertable. <http://hypertable.org>.
- [12] 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.
- [13] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.
- [14] Kernel Based Virtual Machine. [http://www.linux-kvm.org/page/Main\\_Page/](http://www.linux-kvm.org/page/Main_Page/).
- [15] L. Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems*, 1998.
- [16] MySQL Cluster. <http://www.mysql.com/cluster>.
- [17] D. Nurmi, R. Wolski, C. Grzegorzczak, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *IEEE International Symposium on Cluster Computing and the Grid*, 2009. <http://open.eucalyptus.com/documents/ccgrid2009.pdf>.
- [18] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Symposium on Operating System Design and Implementation*, 2010.
- [19] Protocol Buffers. Google’s Data Interchange Format. <http://code.google.com/p/protobuf/>.
- [20] What is Google App Engine? <http://code.google.com/appengine/docs/whatisgoogleappengine.html>.
- [21] XenSource. <http://www.xensource.com/>.
- [22] ZooKeeper. <http://hadoop.apache.org/zookeeper>.