Scalable Transactions across Heterogeneous NoSQL Key-Value Data Stores

Akon Dey

Supervised by Alan Fekete and Uwe Röhm
University of Sydney

akon.dey@student.usyd.edu.au, {fekete, roehm}@it.usyd.edu.au

ABSTRACT

Many cloud systems provide data stores with limited features, especially they may not provide transactions, or else restrict transactions to a single item. We propose a approach that gives multi-item transactions across heterogeneous data stores, using only a minimal set of features from each store such as single item consistency, conditional update, and the ability to include extra metadata within a value. We offer a client-coordinated transaction protocol that does not need a central coordinating infrastructure. A prototype implementation has been built as a Java library and measured with an extension of YCSB benchmark to exercise multi-item transactions.

1. INTRODUCTION

Cloud computing infrastructures are emerging as the preferred deployment platforms for a wide variety of applications, particularly web-based applications. Increasingly, desktop and mobile applications are using the cloud infrastructure to take advantage of the high-availability and scalability characteristics. In the past, these type of systems used local databases to store information and application state. There are many new applications that share some or all their data with applications running on other hosts or in the cloud and use these data stores for persistence.

The data management infrastructure available in the cloud such as Google Cloud Storage (GCS), Windows Azure Storage (WAS), Amazon SimpleDB, and others are simple to setup, access, and require little system administration. They scale in and out seamlessly and are highly available and fault-tolerant.

However, many cloud-based distributed data stores have some limitations. Firstly, there is limited capability to query the data, often restricted to access via the primary key. Complex queries that involve other fields or joins may not be possible. Secondly, the services often provide no transactions or only transactions that access a single record.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 12
Copyright 2013 VLDB Endowment 2150-8097/13/10... $ 10.00.

The limited query capability is usually worked around by designing new data structures and using different data access patterns. However, the weak transaction guarantees are a severe hindrance to application development and requires major application rearchitecture and often prevent of the application from being deployed using these technologies altogether.

There are a few ways to support multi-item ACID [13] transactions when using a data store that provides only single item transactional guarantees. The obvious choice is for every application to manage transactional access to the data store. This is complex, prone to programmer error, and almost certainly results in incorrect behavior.

One way is to implement transaction support in the data store itself. This is complicated and is difficult to implement without compromising scalability and availability.

Another approach is to use middleware to coordinate transactional access to the data store. This approach has similar implementation complexities but is suitable for situations where the applications are deployed in a controlled environment.

A different approach is to define a transactional access protocol to each data store and provide a transaction and data store abstraction API to enable the client applications to access the data with transactional semantics while continuing to take advantage of scalable and reliable access to the data store.

We use this approach to implement a library with an easy-to-use API that defines a client coordinated transaction management protocol with a plugable data store abstraction layer enabling it to handle transactions across more than one data stores.

In this paper we make the following contributions:

- We define a client coordinated transaction protocol to enable efficient multi-item transactions across heterogeneous key-value store by distributed applications.
- We define a data store implementation that provides a corresponding interface to support multi-item transactions.
- We describe the implementation of a library in Java that uses the protocol to perform multi-item transactions across Windows Azure Storage and Google Cloud Storage.
- We describe an extension of the YCSB [7] benchmark, we call YCSB+T, that we use to evaluate the transaction throughput.

2. BACKGROUND

In recent years there have been numerous implementations of distributed key-value stores, each exhibiting different mix of performance, scalability, availability characteristics and alternate architectures. These include Amazon Dynamo [11], Google BigTable [5], Yahoo! PNUTS [6], HBase, and Cassandra [15], Amazon SimpleDB, Amazon S3. These systems use commodity hardware and exhibit scalability and high-availability but provided lower consistency guarantees, often limited to only eventual consistency [21]. Typically only single item transactions are guaranteed and query capability is limited. Data is accessed using the primary key and data scan support is limited to only a few systems like PNUTS and SimpleDB.

More recently, there have been developments like Spinnaker [20], Windows Azure Storage [4], Google Cloud Storage provide single item consistency guarantees. The system design focuses on scalability, performance and single key consistency. Spinnaker uses a Paxos-based protocol to ensure consistency while COPS [17] and Granola [9] use replication protocol optimizations to achieve greater performance while supporting native multi-item transactions.

Despite these advances, the bulk of the systems leave multi-item transactional data access to the client application. This is prone to programmer error and the results are often completely incorrect.

In order to address these issues, some systems have implemented a relational database engine to provide the query capabilities and transaction support with the raw data stored in a distributed key-value store [3]. This is suitable for applications that require an RDBMS for persistence with the advantage that it provides a complete SQL interface with full transaction support. The performance and transaction throughput of the system is limited only by the underlying queue implementation.

Most applications built using key value stores work well because of the relative simplicity of the programming interface to the data store. Many of these applications use write-once-read-many (WORM) data access to the key value store and function well under the eventual consistency setting. However, there are applications that are built to run on the same data that require better consistency across multiple keys.

The first approach to address this issue is to implement transactional capabilities within the data store itself. The data store manages the storage as well as transaction management. The Spanner [8] is a distributed data store that supports transactions.

The COPS [17] and Granole [9] implement the distributed key-value store with a custom API to enable applications to transactional access the data store. Similarly, HyperDex Warp [12] is a high-performance distributed key-value store that provides a client library that supports linearizable transactions. The client library simplifies the access to the data items on behalf of the application using an API provided by the data store which maintains multiple versions of each data item. These systems support transactions across multiple keys with a distributed, homogeneous key-value store. The focus of these systems is to build a better, more capable distributed data store and optimize the transaction coordination across it.

Another way is to use middleware to provide caching and transactional data access. Google Megastore [1] is a transactional key-value store built on top of BigTable. Records are collocated in a tree structure called entity groups where each leaf record is associated with the root record using a foreign key. The foreign key is used to cluster related records and partition data across storage nodes and transactions across records spanning clusters is done using 2PC.

In G-Store [10], related data items are grouped into key groups that are cached locally on a single node. Transactions are only allowed within key groups and keys are allowed to migrate from one key group to another using a group migration protocol. Greater transaction throughput is achieved because data items are cached on the local node.

Deuteromony [16] unbundle the data storage and transaction manager into two separate entities. It defines a protocol, which is an optimization of their earlier work [18], to perform transactional data access using the transaction component (TC) and the data component (DC). This system allows multiple hybrid data stores to be used.

The CloudTPS [22] design uses data store access through a transaction manager split across multiple nodes into local transaction managers (LTM). LTM failures are handled using transaction logs replication across LTMs.

The middleware approach works well when the application is hosted in the cloud and there is a known and controlled set of data stores used by the application. They perform well in this situations and provides one programming interface to the application simplifying the data store access. However, these systems require to be setup and maintained separately.

This is not suitable in our use case where individual application instances need the hands-off, low maintenance features of key value stores and each may have different access privileges to individual data stores.

Another way of implementing multi-key transaction support for distributed key-value stores is to incorporate the transaction coordinator into the client. We know of two implementations that use this approach. Percolator [19] implements multi-key transactions with snapshot isolation semantics [2]. It depends on a central fault-tolerant timestamp service called a timestamp oracle (TO) to generate timestamps to help coordinate transactions and a locking protocol to implement isolation. The locking protocol relies on a read-evaluate-write operation on records to check for a lock field associated with each record. It does not take advantage of test-and-set operations available in most key value stores making this technique unsuitable for client applications spread across relatively high-latency WANs. No deadlock detection or avoidance is implemented further limiting its use over these types of networks.

ReTSO [14] relies on a transaction status oracle (TSO) that monitors the commit of all transactions to implement a lock-free commit algorithm resulting in high transaction throughput. It utilizes a high-reliability distributed write-ahead log (WAL) system called BookKeeper to implement the TSO providing snapshot isolation semantics. Timestamps are generated by a central timestamp oracle. The need to have a TSO and a TO for transaction commitment is a bottleneck over a long-haul network. This prevents this approach to be effective in a WAN layout.

Our approach is similar in many ways to Percolator and ReTSO. We use the transaction start time to obtain the transaction read set. We also use the transaction commit timestamp to tag all the records that belong to the transaction write set. Unlike these systems ours does not depend on any centralized timestamp oracle or logging infrastructure. We utilize the underlying key-value store and its features to provide transaction across multiple records. There is no need to install or maintain additional infrastructure. Our approach enables transactions to span across hybrid data stores that can be deployed in different regions and does not rely upon a central timestamp manager. In the current version, we rely on the local clock to keep time but it is compatible with approaches like TrueTime [8]. We use a simple ordered locking protocol to ensure deadlock detection and recovery without the need of a central lock manager.

3. CONTRIBUTIONS

Current distributed NoSQL data stores exhibit the following characteristics:

- Limited transactional capabilities.
- Rudimentary record locking capability without an ability to avoid or handle deadlocks.
- No standard application interface making it hard for applications to perform transactions across different data stores.

It is evident that the current systems do not address the problem of transaction coordination across heterogeneous data stores. We expect our work to help us better understand and utilize heterogeneous data stores more efficiently in an increasingly varied variety of applications.

```
public void UserTransaction() {
  Datastore cds = Datastore.create("credentials.xml");
  Datastore gds = Datastore.create("goog_creds.xml");
  Datastore wds = Datastore.create("msft_creds.xml");
  Transaction tx = new Transaction(cds);
  try {
    tx.start();
    Record saving = tx.read(gds, "saving")
    Record checking = tx.read(wds, "checking");
    int s = saving.get("amount");
    int c = checking.get("amount");
    saving.set("amount", s - 5);
    checking.set("amount", c + 5);
    tx.write(gds, "saving", saving);
    tx.write(wds, "checking", checking);
    tx.commit();
  } catch (Exception e) {
    tx.abort();
  }
}
```

Listing 1: An example using the library API to access two data records spanning two data stores

Listing 1 is an example of a Java application using the library to access multiple items in more than one data stores.

3.1 Client-coordinated transaction protocol

The protocol depends on the following capabilities of modern distributed key-value data stores:

- single-item strong consistency (always read latest version of the item)
- conditional update and delete, similar to Test-and-Set
- ability to add user-defined metadata to the content of an item; we use this to tag each version with the information about the creating transaction, and also to include both current and preceding version within the item
- a data store that can support global read-only access to records, so we make transaction state visible to all clients

The essence of our proposal is that each item stores the last committed and perhaps a currently active version. Each is tagged with the transaction that created it, and a well-known globally visible transaction status record is kept so the client can determine which version to use when reading, and so that transaction commit can happen just by updating (in one step) the transaction status record. Based on transaction status record, any failure can be either rolled forward to the later version, or rolled back to the previous version. When transactions attempt concurrent activity, the test-and-set capability on each item allows a consistent winner to be determined. A global order is put on transactions, through a consistent hash of their ids, and this can prevent deadlock. This approach is optimized to have parallel processing of the commit activity. Details of the algorithms are shown in Section 4.

The transaction start time, T_{start}, is set at the start of the transaction and is used to pick the correct version of the items in the read set. The transaction commit timestamp, T_{commit}, to tag all the records that belong to the transaction write set. At transaction start, the transaction start time T_{start} is recorded from the clock.

Data items are read from individual containing data stores and cached in an in-memory cache in the client library. Modified objects are kept in the library cache until the transaction is committed.

When the transaction is finished, we perform two stages of processing.

Phase 1: The transaction commit time, T_{commit}, is obtained from the clock. Then every dirty record in the client's record cache is stamped with the commit timestamp metadata. Each record is marked with a PREPARED flag and written to the respective data store.

Phase 2: If all the records have been successfully written, a Transaction Status Record (TSR) is written to a data store that is globally readable to indicate that the transaction is considered as committed. Any future failure will be recovered by rolling forward.

Once the TSR is created, the records in the write set are marked as committed in their respective data stores. This operation can happen in parallel to improve performance. Once all records have been committed, the TSR is deleted.

The library does not use any centralized managed infrastructure to perform transaction commits. The underlying key value store and its inherent features are used to implement transaction across multiple records. As there is no need to install or maintain any central infrastructure it is suitable for use across hybrid data stores that can span multiple data centres.

3.2 YCSB+T transactional NoSQL benchmark

Traditional database benchmarks like the TPC-W are designed to measure the transactional performance of RDBMS implementations against an application domain. Cloud services benchmarks like YCSB, on the other hand, are designed to measure the performance of web-services without focusing on the transactional aspects and data validation.

We are in the process of developing a comprehensive set of benchmarks to measure the transactional performance of applications that use key-value stores with web-service APIs. Performance is measured in terms of both transaction throughput and the probability of data anomalies per transaction. Operations on data items are performed within transaction boundaries and performance of individual start, commit, and abort operations are measured in addition to

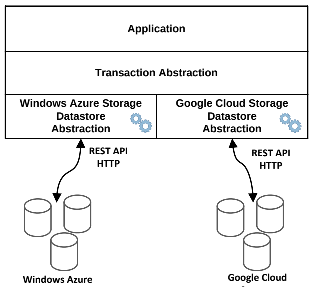
Application
Transaction Abstraction
Windows Azure Storage Datastore Abstraction
Google Cloud Storage Datastore Abstraction
REST API HTTP
REST API HTTP
Windows Azure
Google Cloud

Figure 1: Library architecture

data reads and writes. It also provides a platform for eventual users of the library to evaluate their application use cases against specific data stores.

3.3 Transaction-friendly NoSQL data store

We are in the process of developing a distributed data store that will have native support for 2-phased updates and deletes without compromising the scalability and availability characteristics of typical distributed NoSQL data stores. We are of the opinion that these extensions to the traditional API with the operations; PREPARE, COMMIT, ABORT, and RECOVER in addition the the standard GET, PUT, and DELETE methods will enable the client-coordinated transaction commitment protocol to work more efficiently by reducing the number of API calls to the data store while continuing to support traditional non-transactional access using only the basic set.

4. PROTOTYPE IMPLEMENTATION

The system is implemented as a runtime library that is linked to the client application. The architecture of the library is described in Figure 1. The library provides programming abstraction for transactions and data stores.
The implementation makes the following data store assumptions:

- Support for single item transactions - write followed by read returns the last updated version
- Support for test-and-set operation - conditional writes
- Support for global read-only access

The interface to the data store is provided to the application through a library which exposes an abstraction to transactions through Transaction class and to the data store through a Datastore class.
Record structure: The data record has a record header and data. The header is a collection of fields and their values.

The fields are:

- *valid time start (T_{valid}):* The timestamp after which the version of the record is considered committed if in the COMMITTED state.
- *valid time end (T_{valid_end}):* The timestamp after which the record is considered invalid. This is used to indicate that a record is deleted.
- *lease time (T_{lease_time}):* The transaction lease time to complete the transaction commit. The record state recovery is executed if the lease time expires and the record is in the PREPARED state.
- *transaction identifier (TxID):* Signifies the URI of the transaction that last updated the record. The transaction URI can be examined to determine the fate of the transaction.
- *transaction state (TxState):* The last update state of the record, whether PREPARED or COMMITTED.
- *last update time (T_{last_update}):* The last update timestamp on the data store server.
- *version tag (ETag):* A data store generated version tag for the version of the record. This tag can be used to perform conditional writes and reads.
- *previous version (Prev):* A reference to the previous version of the record.

The Datastore class implements the following methods:

- *start()* - start a transaction
- *read(key)* - read a record with the specified key from the data store.
- *write(key, record)* - write a record with the specified key to the data store.
- *delete(key)* - delete a record identified by the key from the data store.
- *prepare(key, record)* - if the version of the record matches the version on the data store or if the record is a new record, write the contents of the record and its previous version to the data store and mark it with the *PREPARED* state.
- *commit(key, record)* - update record identified by the key to the *COMMITTED* state.

The Transaction class implements the following methods:

- *start()* - start the transaction
- *read(ds, key)* - read a consistent version of the record specified key from the data store and cache it in the transaction cache.
- *write(ds, key, record)* - write the record to the transaction cache to be persisted in the data store specified by ds using transaction commitment protocol.
- *delete(ds, key)* - mark the record identified by the specified key as deleted in the transaction cache to be deleted from the data store at commit time.
- *commit()* - commit all the changes to the data stores using the commitment protocol.
- *abort()* - abort the transaction
- *recover(txid)* - return the status of the transaction. The returned state can be used to rollforward or abort the transaction.

The transaction context is available to the application in the form of an object of the Transaction class. It has a local cache that caches objects that have been read or are to be written to their data store of origin. The cache maps the key and data store to the corresponding data record.

In addition to this, the transaction context must have a location to host the TSR. The TSR is a record is that is identified by the transaction identifier (UUID), is written to a data store that is called the Coordinating Data Store (CDS) and must be globally readable. It must be noted that the choice of the CDS is determined by the application and is suitable as long as the global read visibility constraints are fulfilled.

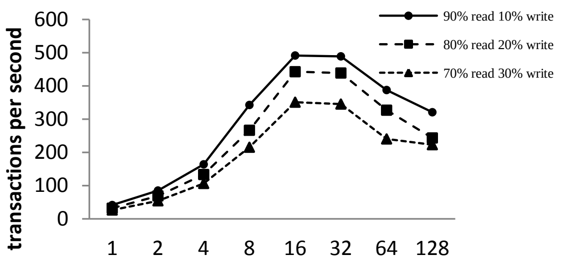

Figure 2: YCSB+T on a EC2 host against WAS

A transaction context is created in the application by specifying the CDS to host TSRs. In addition, the record cache, a hash table that maps the data store and key to a record is created. Once this context is successfully create it can be used to perform multi-record read and write transactions.

Start transaction: The transaction is started by creating a unique transaction identifier using a UUID generator and setting the transaction start time (T_{start}) to the current.

Transactional read: The record is read from the data store using the supplied key. The data store populates the record header and contents that can be used by the transaction code. The record state is checked to see if the transaction has been committed i.e. if it is in the COMMITTED state. If it is in the COMMITTED state, the T_{valid_time} record header is compared with the current transaction start time, T_{start}. If the version of the record read is created after the start of the current transaction, the previous version is inspected for validity. If the record is in the PREPARED state, the transaction URI is used to inspect the state of the TSR. If the TSR exists, the record version is considered committed. If the transaction lease time has expired, the record is rolled forward and marked COMMITTED.

Once a valid version of the record is read from the data store, it is put into the transaction record cache and then returned to the caller.

Transactional write: Transactional writes are simple. The record associated to the key is written to the cache. If an earlier version exists it is marked as the previous version. The record is written to the data store at transaction commit time.

Transaction commit: The transaction commit is performed in two phases.

Prepare: The record cache is inspected and all dirty objects are inserted the write-set. The records are marked with the transaction status record URI, the transaction commit time, the transaction state to PREPARED then conditionally written to their respective data stores in the order of the hash values of the record keys. This is done by using the data store prepare() method. This method performs a conditional write using the record version tag (ETag). The prepare phase succeeds if all dirty records are successfully prepared.

Deadlock detection and avoidance: Concurrent conflicting transactions prepare the records in the same order as they use the hash values of the record keys. Only one of the transactions will succeed in performing a conditional write operation to the data store. The other transaction aborts after rolling back its prepared records.

Commit: The TSR is written to the coordinating data store to indicate that all the records have been successfully prepared. The records are then committed by calling the data store commit() method in parallel. The record commit method marks the record with the COMMITTED state. Once the records are committed the transaction status record is deleted asynchronously from the coordinating data store.

Transaction abort: If the transaction commit operation has not been initiated the abort operation is trivial. The record cache is cleared and the transaction is marked as aborted.

If the transaction is partially committed the transaction is aborted if the TSR has not been written to the transaction coordinating data store. In this case, the abort is performed by undoing the record prepare operation by writing the previous version of the record to the data store. Once the transaction status record has been written to the coordinating data store the transaction cannot be aborted.

Transaction recover: Transaction recovery is performed lazily in case of application failure. The transaction state is inspected in the record header to see if the record is in the PREPARED or COMMITTED state. If the record is in the COMMITTED state the recovery is unnecessary. If the record is in the PREPARED state, the record is rolled forward and committed if the transaction status record (TSR) exists and rolled back otherwise. The rollforward is performed by marking the record with the COMMITTED state. A rollback is performed by overwriting the record with its previous versions.

There are a number measures we have used in order to improve the performance of the library and reduce TCP/IP connection overheads we use a connection pool for each data store endpoint. In addition, we use a thread pool to implement parallel asynchronous writes and asynchronous deletes. The current version does not perform one-phase commit optimization for transactions with only one object in the write-set. We are exploring ways to further improve the performance.

5. EVALUATION

We have done preliminary evaluations with YCSB+T running on Amazon Elastic Compute Cloud (EC2) hosts using Windows Azure Storage (WAS) and Google Cloud Storage (GCS). Performance measurements were taken while varying the ratio of reads to writes from 90:10, 80:20, to 70:30 using 1, 2, 4, 8, 16, 32, 64, and 128 client threads with 10000 records accessed in a Zipfian distribution pattern from a single host. The graph in Figure 2 describes the results of the tests performed with the client running on a c1-xlarge EC2 host against one WAS data store container.

The number of transactions scales linearly up to 16 client threads (this gives approximately 491 transactions per second with a 90:10 mix of read and write transactions respectively using a single WAS data store container). With 32 threads, the number of transactions remains roughly the same as with 16 threads. This appears to be caused by a bottleneck in the network or the data store container itself and needs to be further studied.

Subsequently, increasing the number of client threads to 64 and 128 with the same transaction mix reduces the net transaction throughput. Our investigations indicate that this may be a result of thread contention. Increasing the ratio of writes reduces the throughput but does not change the performance characteristics. We ran YCSB+T instances on multiple EC2 hosts but the net transaction throughput across all parallel instances was similar to the throughput from the same number of threads on a single host. This supports our argument that we are hitting a request rate limit. The initial results look encouraging and we intend to publish our findings once we have completed our evaluations against a different data store where we have more control over the server side performance characteristics.

6. CONCLUSIONS AND FUTURE WORK

This paper describes a reliable and efficient multi-item transactional data store access protocol. We have described an implementation that enables ubiquitous client applications to access large-scale distributed key-value data store in a transactional manner. The initial evaluations and analysis indicates that this approach is suited for high transaction rates by applications deployed across vastly diverse network latencies.

We describe an extension of the YCSB cloud services benchmark with transaction support, we call YCSB+T. It is suitable for evaluation of transaction access to distributed key-value stores by enabling operations on the data store to be grouped into transactions.

Further, we have given a description of an implementation of our client library in Java with data store abstraction plugins for Windows Azure Storage (WAS) and Google Cloud Storage (GCS) and have written applications to evaluate it.

The system described here relies on the local system clock to obtain transaction timestamps. However, it can be made to work with the Spanner TrueTime API [8] after a few modifications. We are currently exploring ways to determine degree of the uncertainty in the local clock on the client using the information available from the data exchanged with the NoSQL data stores participating in a transaction so that it can be used with techniques like the TrueTime API.

We are continuing to evaluate our library using YCSB+T on EC2 with data stored in WAS and GCS. Initial results are promising and we will publish the results once we have a more information in future publications. The evaluations conducted so far have focused on YCSB+T instances running in a single cluster. Next, we will evaluate our system in a more heterogeneous setup that spans multiple regions and differing network latencies, bandwidth, and reliability chatacteristics.

We are currently in the process of implementing a distributed key-value store with the extended API that will enable the client coordinated transaction protocol to work more efficiently and yet continue to support traditional application use cases. We will describe the system in more detail and share our experiences in future publications.

7. REFERENCES

[1] J. Baker, C. Bondç, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, pages 223–234, Jan. 2011.

[2] H. Berenson, P. Bernstein, et al. A critique of ANSI SQL isolation levels. In *SIGMOD '95*, pages 1–10, 1995.

[3] M. Brantner, D. Florescu, et al. Building a database on S3. In *SIGMOD '08*, pages 251–264, 2008.

[4] B. Calder et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP'11*, pages 143–157, 2011.

[5] F. Chang, J. Dean, S. Ghemawat, et al. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):1–26, June 2008.

[6] B. F. Cooper, R. Ramakrishnan, et al. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1:1277–1288, August 2008.

[7] B. F. Cooper, A. Silberstein, et al. Benchmarking cloud serving systems with YCSB. In *SoCC '10*, pages 143–154, 2010.

[8] J. C. Corbett, J. Dean, et al. Spanner: Google's globally-distributed database. In *OSDI '12*, pages 251–264, 2012.

[9] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC'12*, pages 21–21, 2012.

[10] S. Das, D. Agrawal, et al. G-Store: a scalable data store for transactional multi key access in the cloud. In *SoCC '10*, pages 163–174, 2010.

[11] G. DeCandia, D. Hastorun, et al. Dynamo: amazon's highly available key-value store. In *SOSP '07*, pages 205–220, 2007.

[12] R. Escriva, B. Wong, et al. Warp: Multi-Key Transactions for Key-Value Stores. Technical report, United Networks, LLC, 05 2013.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Mogan Kaufmann Publishing Inc., 1992.

[14] F. Junqueira, B. Reed, et al. Lock-free transactional support for large-scale storage systems. In *IEEE/IFIP DSN-W '11*, pages 176 –181, June 2011.

[15] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[16] J. J. Levandoski, D. B. Lomet, et al. Deuteronomy: Transaction support for cloud data. In *CIDR '11*, 2011.

[17] W. Lloyd, M. J. Freedmand, et al. Don't settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP '11*, Oct. 2011.

[18] D. B. Lomet, A. Fekete, et al. Unbundling transaction services in the cloud. In *CIDR '09*, 2009.

[19] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI'10*, pages 1–15, 2010.

[20] J. Rao, E. J. Shekita, et al. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4:243–254, January 2011.

[21] W. Vogels. Eventually consistent. *Queue*, 6:14–19, October 2008.

[22] W. Zhou, G. Pierre, et al. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *IEEE Transactions on Services Computing*, 2011.