YCSB+T: Benchmarking web-scale transactional databases

Conference Paper · March 2014

DOI: 10.1109/ICDEW.2014.6818330

4 authors, including:

Akon Dey
The University of Sydney
10 PUBLICATIONS **58** CITATIONS

SEE PROFILE

Raghu Nambiar
Cisco Systems, Inc
78 PUBLICATIONS **1,030** CITATIONS

SEE PROFILE

YCSB+T: Benchmarking Web-scale Transactional Databases

Akon Dey #1, Alan Fekete #2, Raghunath Nambiar *3, Uwe Röhm #4

School of Information Technologies, University of Sydney
Sydney NSW 2006 Australia
[1] akon.dey@sydney.edu.au
[2] alan.fekete@sydney.edu.au
[4] uwe.roehm@sydney.edu.au

* *Cisco Systems, Inc.*
275 East Tasman Drive
San Jose, CA 95134 USA
[3] rnambiar@cisco.com

***Abstract*—Database system benchmarks like TPC-C and TPC-E focus on emulating database applications to compare different DBMS implementations. These benchmarks use carefully constructed queries executed within the context of transactions to exercise specific RDBMS features, and measure the throughput achieved. Cloud services benchmark frameworks like YCSB, on the other hand, are designed for performance evaluation of distributed NoSQL key-value stores, early examples of which did not support transactions, and so the benchmarks use single operations that are not inside transactions. Recent implementations of web-scale distributed NoSQL systems like Spanner and Percolator, offer transaction features to cater to new web-scale applications. This has exposed a gap in standard benchmarks.**

We identify the issues that need to be addressed when evaluating transaction support in NoSQL databases. We describe YCSB+T, an extension of YCSB, that wraps database operations within transactions. In this framework, we include a validation stage to detect and quantify database anomalies resulting from any workload, and we gather metrics that measure transactional overhead. We have designed a specific workload called *Closed Economy Workload (CEW)*, which can run within the YCSB+T framework. We share our experience with using CEW to evaluate some NoSQL systems.

I. Introduction

Cloud computing or utility computing [1] is gradually gaining popularity as the deployment architecture of choice for application in large established enterprises and small startups and businesses alike. The inherent distributed nature of these architectures coupled with the use of commodity hardware has given rise to the development of new data management technologies broadly classified under the name NoSQL databases which include Amazon S3 [2], Google BigTable [3], Yahoo! PNUTS [4] and many others. These systems take advantage of the distributed nature of the deployment platform to support web-scale storage and throughput while tolerating failure of some component machines. As tradeoff, they typically offer clients less in query expressivity (for example, they may not perform joins) and less transactional and consistency guarantees.

Early examples of NoSQL systems such as AWS' S3 [2] or Amazon's internal-use Dymano [4] provide little if any support for transactions and information consistency. Some allow a read operation to return somewhat stale data, and typically only eventual consistency [5] or time-line consistency [4] is offered respectively. More recent commercial offerings like like Windows Azure Storage (WAS) [6] and Google Cloud Storage (GCS) [7] allow the transaction-like grouping of multiple operations but restrict this to involve either a single item, or a collection of items that are collocated. Another approach to better support for application logic lies in richer operations such as *test-and-set* or *conditional put*.

Several designs have recently appeared to overcome the limited transactional semantics of these NoSQL databases. Examples include Google Percolator [8], G-Store [9], CloudTPS [10], Deuteronomy [11], Megastore [12] and Google Spanner [13].

How can these designs be evaluated? Traditional data management platforms were measured with industry standard benchmarks like TPC-C [14] and TPC-E [15]; these have focused on emulating end-user application scenarios to evaluate the performance (especially the throughput, and throughput relative to system cost) of the underlying DBMS and application server stack. These benchmarks run a workload with queries and updates that are performed in the context of transactions, and the integrity of the data is supposed to be verified during the process of the execution of the benchmark. If the data is corrupted, the benchmark measurement is rejected entirely.

For web-scale data management, especially the available-despite-failures key-value stores in the NoSQL category, a new benchmark YCSB [16] has become accepted. The focus of this benchmark is raw performance and scalability; correctness is not measured or validated as part of the benchmark, and the operations do not fall within transactions (since these systems may not support transactions nor guarantee data consistency). YCSB is actually a flexible framework within which the workload and the measurements can be extended.

Our proposed benchmark (called YCSB+T) retains the flexibility of YCSB by allowing the user to implement the DB

interface to their database/data store; it allows for additional operations apart from the standard read, write, update, delete and scan; it enables defining workloads in terms of these operations; and most importantly it allows these operations to be wrapped into transactions. Further, there is a validation stage to specify consistency checks to be conducted on the database/data store after the completion of the workload in order to detect and quantify transaction anomalies.

Our approach is intended to fill the gap between traditional TPC-C-style benchmarks that are designed for transactional RDBMSs and the non-transactional HTTP/web-service benchmarks which have no ability to define transactions.

We make the following contributions in this paper:

- Describe our extension of the YCSB benchmark that we call YCSB+T, with support for transactional operation and workload validation to detect and quantify anomalies resulting from the workload.
- Describe a simple benchmark we call the closed economy benchmark built using YCSB+T, that we can use to evaluate the performance and correctness of systems.
- Share our experience in using the benchmark to evaluate some NoSQL data stores.

II. Background

In this section we give an overview of NoSQL database systems and then describe approaches to providing transactional access to them. We also summarise the key features of the YCSB benchmark for NoSQL systems.

A. Overview of NoSQL database systems

Different distributed key-value data stores or NoSQL data store exhibit a mix of different performance, scalability, availability characteristics and architectures. While these systems may differ in architecture and implementation they strive to achieve the following characteristics. This is discussed in further detail in the original YCSB paper [16].

- **Scale-out:** Distributed NoSQL databases are able to support the large data sizes and high request rates because they are able to spread the request load and data to be stored across a large number of commodity servers each hosting a part of the data. A successful scale-out mechanism is able to effectively spread the data and client requests across these machines without exposing any bottlenecks.
- **Elasticity:** Elasticity enables a system to add capacity by adding new servers and spreading the load effectively while the system is still running. It complements the scale-out capabilities of a system.
- **High availability:** Commodity hardware is prone to failure making the availability of the system an important requirement. This is particularly important when these systems host data belonging to multiple tenants.

It is impossible to provide all the desirable features into one system and each one makes different design and architecture choices. For instance it is impossible to simultaneously achieve consistency, availability and partition tolerance [17]. Therefore, these systems have to make the following tradeoffs in order to exhibit the characteristics mentioned above:

- **Read versus write performance:** High read performance is achievable with good random I/O throughput while higher write performance can be achieved using append-only log-structured storage structured.
- **Latency versus durability:** Persisting data to disk achieves durability but increases write latency significantly. Not synching writes to the disk reduces latency and improves throughput but reduces durability guarantees.
- **Synchronous versus asynchronous replication:** Replicating data improves performance, system availability and avoids data loss. This can be done either synchronously or asynchronously. Synchronous replication increases write and update latency while asynchronous replication reduces latency but also reduces consistency guarantees caused by stale data.
- **Data partitioning:** Data can be stored in a row-oriented or column-oriented storage structure. Row-oriented storage structures are suitable for applications that need to access a large proportion of the fields in each data record during a transaction while column storage structures are suited to applications that use a very small number of the total fields in each record and are suitable for situations when the records have a very large number of fields.

We are interested in systems that choose consistency over availability and provide a transactional interface to the application. In particular, we would like to evaluate systems that support multi-item transactions.

Examples of NoSQL data stores are Amazon Dynamo [18], Google BigTable [3], Yahoo! PNUTS [4], HBase, Cassandra [19], Amazon SimpleDB and Amazon S3. Typical deployments utilize commodity hardware and are scalable and highly available but provide limited consistency guarantees. Some limited to just eventual consistency [5]. Transactions are guaranteed on single-item operations only and query capability is limited. Data is looked up using the primary key and more complex queries are supported in only a few systems like Yahoo! PNUTS and SimpleDB.

Recent developments like Spinnaker [20], Windows Azure Storage [6], Google Cloud Storage and others support single item transactions. The system design focuses on scalability, performance and consistency of operations on single data items. Spinnaker uses a Paxos-based protocol to ensure consistency while COPS [21] and Granola [22] use replication protocol optimizations to achieve greater performance while supporting native multi-item transactions.

However, most systems leave multi-item transactions to be handled by the application. This is susceptible to programmer error and the resulting implementation is often completely wrong.

B. Transactions in NoSQL database systems

One way to address this is to implement a relational database engine to provide the query capabilities and trans-

action support with the raw data stored in a distributed key-value store [23]. This is suitable for applications that require a complete SQL interface with full transaction support. The performance and transaction throughput of the system is limited only by the underlying data store and queue implementation.

The relative simplicity of the the data store API makes application development simple and robust. These applications most often use write-once-read-many (WORM) data access pattern and function well under eventual consistency guarantees. However, there are increasing demands for applications that are built to run on the same data that require better consistency guarantees across multiple records.

One way to solve this is to implement transactional capabilities within the data store itself. The data store manages the storage as well as transaction management. The Spanner [13] system developed at Google is a distributed NoSQL data store that supports native transactions built on top of BigTable [3].

COPS [21] and Granole [22] implement a distributed key-value store that provides a custom API to enable transactional access to the data store. HyperDex Warp [24] is another high-performance distributed key-value store that provides a client library that supports linearizable transactions and simplifies access to data items in 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 and focus on providing a better, more capable distributed data store and optimize the transaction coordination across it.

Another approach uses middleware to for caching and coordinating transactional data access. Google Megastore [12] implements a transactional key-value store built on top of BigTable in which related data records are collocated in a tree structure called *entity groups*. 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 entity groups use 2PC.

Similarly in G-Store [9], related records are arranged into key groups that are cached locally on a single server node. Transactions are only allowed within key groups but keys are allowed to migrate from one key group to another using a group migration protocol. Improved transaction throughput is observed as a result of caching data items on the local node.

Deuteromony [11] separates the data storage and transaction manager into two different entities and a protocol, an optimization of their earlier work [25], is used to perform transactional data access with the help of the transaction component (TC) and the data component (DC). It supports the use of multiple heterogeneous data stores.

The CloudTPS [10] design uses data store access through a transaction manager that is split across the nodes into multiple local transaction managers (LTM). LTM failures are handled using transaction logs replicated 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 increasing the management overhead.

This is not suitable for all use cases, particularly where individual application instances need the hands-off, low maintenance features characteristic 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 [8] implements multi-key transactions with snapshot isolation semantics [26]. 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 [27] 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.

We have implemented a system similar to Percolator and ReTSO. It uses the transaction start time to obtain the transaction read set and the transaction commit timestamp to tag all the records that belong to the transaction write set. It does not depend on any centralized timestamp oracle or logging infrastructure. It utilizes the underlying key-value store and its features to provide transaction across multiple records. There is no need to install or maintain additional infrastructure. It 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, it relies on the local clock to keep time but it is compatible with approaches like TrueTime [13]. It uses a simple ordered locking protocol to ensure deadlock detection and recovery without the need of a central lock manager.

We were inspired to develop YCSB+T and the Closed Economy Workload (CEW) in order to effectively evaluate the performance, scalability, transactional overhead and consistency of our library and NoSLQ key-value store implementation.

C. Overview of YCSB

The Yahoo! Cloud Services Benchmark (YCSB) is an extensible benchmarking framework that is widely used to

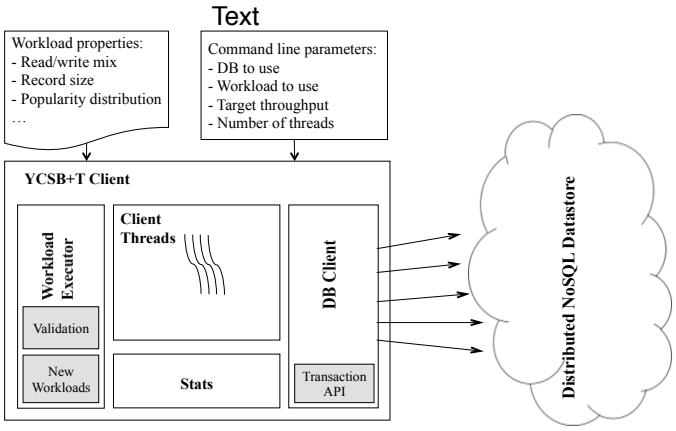

Fig. 1. YCSB+T architecture

measure the performance and scalability of large-scale distributed NoSQL systems like Yahoo! PNUTS [4] as well as traditional database management systems like MySQL. Figure 1 shows the architecture of YCSB+T. The light coloured rectangles represent YCSB components and the dark coloured rectangles represent additions and enhancements implemented by YCSB+T.

The **workload executor** instantiates and initializes the workload then loads the data into the database to be tested or executes the workload on the database using the **DB client** abstraction. The CoreWorkload workload defined by default with the YCSB framework defines different mixes of the simple database operations read, update, delete and scan operations on the database. The **doTransactionRead()** operation reads a single record while the **doTransactionScan()** operation is used to fetch more than one record identified by a range of keys. Data records are inserted using the **doTransactionInsert()** method and **doTransactionUpdate()** is issued to perform record updates in the database. The **doTransactionReadModifyWrite()** reads a record determined by the *keysequence* generator, assigns a new value and writes it back to the data store.

Operations on records in the database are defined by a distribution that is specified in the **workload parameter file**. The YCSB distribution defines six workloads called workloads A, B, C, D, and E that generate different read-intensive, update-intensive and scan-intensive loads on the database respectively.

The distribution also defines a DB client abstractions for various NoSQL and other data stores and databases including Cassandra, HBase, MongoDB and MySQL.

The YCSB client starts the specified number of client threads each of which use the workload to perform the specified database operations using the DB client abstraction. The client can be passed command-line parameters to alter the behavior of the client.

III. Benchmark Tiers

YCSB contains two benchmark tiers for evaluating the performance and scalability of the cloud serving system. In addition, they discuss two additional tiers to address availability and replication.

We propose two additional tiers for evaluating the transactional overhead on throughput of the data store, and transactional consistency of the database as a result of concurrent execution of the benchmark workload.

Tier 1 and 2 are described in detail in the YCSB paper [16] and tiers 3 and 4 are discussed as future work. Here we discuss our two new tiers in further detail.

A. Tier 5 - Transactional Overhead

The Transactional Overhead tier of the benchmark focuses on measuring the overhead of the individual database operations (exposed via the DB client class) when they are executed in a transactional context. The latency of database CRUD and scan operations are measured in both transactional and non-transactional modes. In addition, the latency for DB start(), abort() and commit() method calls are also gathered in both modes.

The Transactional Overhead tier of the YCSB+T framework aims to characterize the overhead of transactions and identify the impact of each database operation that executes within and outside a transactional context.

This tier of the benchmarking framework requires the workload executor to capture the metrics associated with the database transactional operations start(), commit() and abort() and the workload methods to capture the latencies measured for the CRUD and scan database operations.

The latency of each raw database CRUD and scan operation along with the start, commit and abort operations captured makes it possible to measure the overhead of transactional access to the data store.

B. Tier 6 - Consistency

The primary reason to execute database operations within the context of a transaction is to ensure that the ACID transaction properties and consistency in particular are preserved. The isolation level used to execute transactions has a significant impact on the performance of the operations.

The Consistency tier of the YCSB+T framework is designed to detect consistency anomalies in the data introduced during the execution of the workload and quantify the amount of anomalies detected.

To achieve this, a database validation phase is added to the workload executor that is implemented as a method in the workload class. The validation stage goes over all the records in the database and applies an application-defined check to the content of the database. The validation stage also calculates an application-specific *anomaly score* to quantify the degree of inconsistency detected. It is intended that a score of zero should mean that the data is entirely consistent (as from a serialisable execution).

IV. Benchmark details

In this section we describe the architecture of YCSB+T and provide a detailed descriptions of the enhancements made to

support transactional access to data stores and the validation stage added to the workload.

A. Architecture

The YCSB+T workload consists of an additional method that is called after the workload executor is used to loads data or execute the workload on the database to validate its consistency. The validate() method by default is a *no-op* and is only implemented by workload that performs a validation process.

The client thread wraps the operations performed on the database through the workload abstraction with transactions by calling DB.start() before calling the workload *doTransaction()* or *doInsert()* method. If the workload method succeeds, the transaction is committed by calling the *DB.commit()* method else the *DB.abort()* method is called.

The DB *start()*, *commit()* and *abort()* methods are empty, *no-op* methods by default. The allows for backward compatibility for existing code written to run with YCSB.

B. Data validation after benchmark execution

We add a method to validate the data in the database that is executed by the YCSB+T client after the data has been loaded (in the *load phase*) or the benchmark workload has completed (*transaction phase*). We call this state the *validation stage*. This is achieved by adding a method *validate(DB db)* to the Workload class. This method is an empty method by default and can be overloaded by derived Workload classes. The workload executor calls the *Workload.validate()* method after executing the workload just before printing the results of the measurements.

C. Closed Economy Workload

This is a simplified simulation of a closed economy, one in which money does not enter or exit the system during the evaluation period. The economy consists of a collection of predetermined number of accounts and a total cash in the system that is distributed evenly across all the accounts.

It is implemented in a class called the *ClosedEconomyWorkload* which extends the Workload class. It simulates a closed economy where everyone has a bank account which has an initial balance of $1000.

1) Load phase: The *doInsert()* method is overloaded to use the *CounterGenerator* to generate keys starting from 0 (or value specified by the *insertstart* workload parameter) to the maximum key value specified by the *insertcount* workload parameter. Each key denotes an account number and is assigned an initial balance in each account that is set to a portion of the amount set by the workload parameter *total_cash*.

2) Transaction phase: The transaction phase is implemented in the *doTransaction()* method in the Workload class. Its primary task is to pick an operation based on the specified mix and call the database operations *insert()*, *read()*, *update()*, *delete()* or *scan()*. The operations perform the following tasks:

- **doTransactionInsert()** creates a new account with an initial balance captured from doTransactionDelete() operation described below.
- **doTransactionRead()** reads a set of account balances determined by the key generator.
- **doTransactionScan()** scans the database given the start key and the number of records and fetches them from the database.
- **doTransactionUpdate()** reads a record and add $1 from the balance captured from delete operations to it and write it back.
- **doTransactionDelete()** reads an account record, add the amount to the captured the balance (capture used in doTransactionInsert()) and then deletes the record.
- **doTransactionReadModifyWrite()** reads two records, subtracts $1 from the one of the two and adds $1 to the other before writing them both back.

3) Validation phase: In the case of CEW, each transaction ought to maintain an invariant value for the sum of all account values. Thus an inconsistency is shown as a difference between the sum of account values at the start and end of the execution. The *validation phase* is implemented by overloading the *validate()* method from the Workload class. This method iterates all the keys and adds up the account balance and validates the total against the total stored after the load stage using the *total_cash*. Because one would expect more anomalies to be introduced as more operations are performed, we present the metric for inconsistency as the amount of change in total balance, divided by the number of operations that were executed. Concretely, the following is a formal definition of the *simple anomaly score*:

$$\gamma = \frac{|S_{initial} - S_{final}|}{n}$$

where,
γ - simple anomaly score
$S_{initial}$ - initial sum of all the accounts
S_{final} - final sum of all the accounts
n - the total number of operations executed

V. Evaluations

The focus of this section is to demonstrate the effectiveness of our framework to produce useful information when running on example NoSQL stores. In particular, we focus on showing how the existing YCSB metrics remain useful for performance and scalability, and also how the new tiers give valuable insight to the extent of anomalies introduced and the overhead of transactions.

A. Evaluating Performance and Scalability

To show how YCSB+T includes the Tier 1 and Tier 2 features of YCSB, we give a selection of measurements produced when running against a client coordinated transaction library from our previous research. We ran the YCSB+T client on Amazon Elastic Compute Cloud (EC2) hosts using our library [28] to manage access to both Windows Azure Storage (WAS) and Google Cloud Storage (GCS). Performance was measured while varying the ratio of reads to writes from 90:10, 80:20, to 70:30, and using 1, 2, 4, 8, 16, 32, 64, and 128 client

Fig. 2. YCSB+T throughput on EC2 with WAS

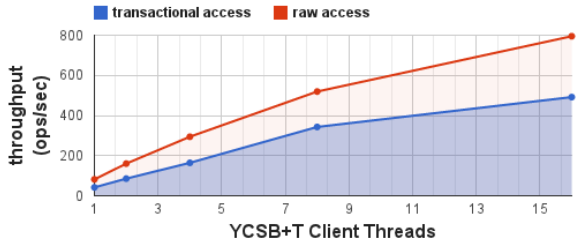

Fig. 3. Impact of transactions on throughput

threads on a single host. 10000 records were accessed in a Zipfian distribution pattern. One set of experiments placed the client on a c1-xlarge EC2 host against one WAS data store container. The resulting graph is in Figure 2.

We observed that the number of transactions scales linearly up to 16 client threads. This results in roughly 491 transactions per second with a 90:10 mix of read and write transactions respectively with the data stored in 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.

B. Performance impact of Transactions

We again use a client coordinated transaction library [28] as the platform when we demonstrate the Tier 5 aspects of YCSB+T. The graph in Figure 3 shows the overhead of transactions on the throughput of the system. As the number of threads increase from 1, 2, 4, 8, through to 16 the non-transactional access to the database scales from 81.57 operations per second for 1 thread to 794.97 operations per second for 16 threads. In the same environment when the operations are each run singly within a transaction, the throughput for 1 thread is 41.69 transactions per second to 491.66 transactions per second for 16 threads with transactions access. Thus the throughput is reduced by about 30 to 40% from the overhead of transaction management.

C. Evaluating Correctness

To show the Tier 6 aspects of YCSB+T and the CEW workload, we ran against a WiredTiger[1]key-value store augmented with an HTTP interface that we implemented using the Boost ASIO library. In these experiments we do not run transactionally (so that anomalies do arise). A DB class extension is used to access the key-value store. The start(), commit() and abort() methods are not implemented in this situation, and therefore these methods revert to the default *no-op* methods defined in the DB class.

We ran the key-value store server and the YCSB+T client run on the the same machine to reduce network latency and maximize throughput. We ran our tests on a MacBook Air running an Intel Core i5 dual-core 1.8 GHz processor with 8 GB 1600 MHz DDR3 RAM and 125 GB SSD. We varied the number of threads from 1, 2, 4, 8 to 16. The following is an example of the command line used to execute the YCSB+T client with 16 threads as described in Listing 1.

```
$ java com.yahoo.ycsb.Client -db com.yahoo.ycsb.db.RawHttpDB -P workloads/
      closed_economy_workload -threads 16 -t
```

Listing 1. YCSB+T client command line

An example of the *workload/closed_economy_workload* properties file is described in Listing 2.

```
recordcount=10000
operationcount=1000000
workload=com.yahoo.ycsb.workloads.ClosedEconomyWorkload
totalcash=10000000
readproportion=0.9
readmodifywriteproportion=0.1
requestdistribution=zipfian
fieldcount=1
fieldlength=100
writeallfields=true
readallfields=true
histogram.buckets=0
```

Listing 2. A CEW workload properties file

```
YCSB+T Client 0.1
Command line: -db com.yahoo.ycsb.db.RawHttpDB -P workloads/closed_economy_workload
      -threads 16 -t
Loading workload...
Starting test.
Validation failed
[TOTAL CASH], 1000000
[COUNTED CASH], 999971
[ACTUAL OPERATIONS], 1000000
[ANOMALY SCORE], 2.9E-5
Database validation failed
[OVERALL], RunTime(ms), 124619.0
[OVERALL], Throughput(ops/sec), 8024.458549659362
[UPDATE], Operations, 200206
[UPDATE], AverageLatency(us), 1536.4616944547117
[UPDATE], MinLatency(us), 1202
[UPDATE], MaxLatency(us), 80946
[UPDATE], Return=0, 200206
[UPDATE], >0, 200206
[COMMIT], Operations, 1000000
[COMMIT], AverageLatency(us), 0.083521
[COMMIT], MinLatency(us), 0
[COMMIT], MaxLatency(us), 795
[COMMIT], Return=0, 1000000
[COMMIT], >0, 1000000
[START], Operations, 1000000
[START], AverageLatency(us), 0.081408
[START], MinLatency(us), 0
[START], MaxLatency(us), 658
[START], Return=0, 1000000
[START], >0, 1000000
[READ-MODIFY-WRITE], Operations, 100103
[READ-MODIFY-WRITE], AverageLatency(us), 6.128447698870164
[READ-MODIFY-WRITE], MinLatency(us), 5
[READ-MODIFY-WRITE], MaxLatency(us), 187
[READ-MODIFY-WRITE], >0, 100103
[TX-READMODIFYWRITE], Operations, 100103
[TX-READMODIFYWRITE], AverageLatency(us), 6134.376252459966
[TX-READMODIFYWRITE], MinLatency(us), 5204
```

[1]http://www.wiredtiger.com

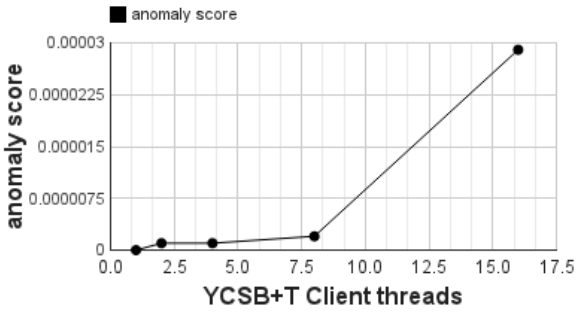

Fig. 4. Number of threads vs anomalies score

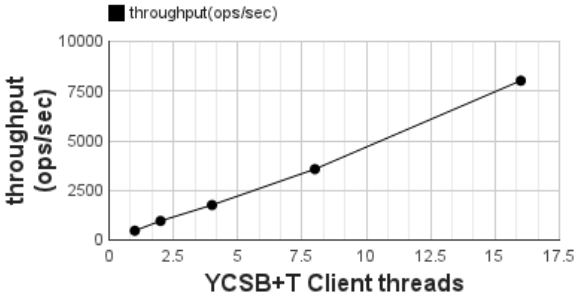

Fig. 5. Number threads vs throughput

```
[TX-READMODIFYWRITE], MaxLatency(us), 186959
[TX-READMODIFYWRITE], Return=0, 100103
[TX-READMODIFYWRITE], >0, 100103
[READ], Operations, 1110103
[READ], AverageLatency(us), 1522.2613180939065
[READ], MinLatency(us), 1174
[READ], MaxLatency(us), 165508
[READ], Return=0, 1110103
[READ], >0, 1110103
[TX-READ], Operations, 899897
[TX-READ], AverageLatency(us), 1526.208302727979
[TX-READ], MinLatency(us), 1178
[TX-READ], MaxLatency(us), 165585
[TX-READ], Return=0, 899897
[TX-READ], >0, 899897
```

Listing 3. Output of running the CEW workload

The graph in Figure 4 shows the anomalies score when the number of YCSB+T threads are increased from 1, 2, 4, 8, through to 16. Note that no anomalies are present at all with a single thread (as there is no concurrency in this setting). It is clear that the increased contention as a result of more concurrent threads and the Zipfian key distribution pattern causes some records to be written to by multiple threads simultaneously causing the anomalies.

To provide context for this, the graph in Figure 5 describes the linear increase in throughput achieved as the number of YCSB+T threads are increased from 1, 2, 4, 8 through to 16.

VI. Related work

Computer systems of all types are evaluated using benchmarks that measure features at a particular level of abstraction. Our work, like YCSB and the TPC suite of benchmarks, deal with data management access, with a large collection of items and operations that access and modify those items (perhaps as get/put, or as SELECT/UPDATE). An aim of these benchmarks is to satisfy the criteria of a successful benchmark prescribed by Gray [29]. They are: *relevance* to an application domain, *portability* to allow benchmarking of different systems, *scalability* to support benchmarking large systems, and *simplicity* to ensure that the results are easy to understand. In this paper we have endeavoured to maintain these properties of YCSB while extending it to apply to a new class of systems.

A previous extension of YCSB called YCSB++ [30] is designed to evaluate non-transactional access to distributed key-value stores. It adds functionality to enable bulk loading of data into HBase and Accumulo in the form of an extended API. It also allows operations like B-Tree splits to be performed on the database indexes to simulate application usage. Other useful features include the ability to launch YCSB clients from multiple nodes and to coordinate readers and writers to simulate complex read-after-write application scenarios.

Important work from UCSB [31], [32] has extended YCSB for evaluating novel systems that support transactions. These measurements modify the DB client and encapsulate each database operation in a transaction within the methods of this class. This is suitable to measure the performance of the system but it does not provide an ability to measure the new tiers (the transactional overhead of individual database operations, nor is the consistency measured) that we propose.

While not building on YCSB, there have been other researchers that have offered benchmarks for consistency properties in data platforms. In the context of a traditional (centralised) database, Shasha and Bonnet [33] measured the number of read operations that do not return the correct latest data. For clouds, Wada et al [34] measured the probability of returning stale values, as a function of how much time had elapsed between the latest write and the read. Our approach here, of measuring the extent to which the data has deviated from a consistent state, follows that used by Fekete et al [35] for a centralised database. A different approach to measure consistency is found in Zellag and Kemme [36] where the execution trace is captured, and the non-serializable executions are detected by cycles in the dependency graph.

A survey and taxonomy for approaches to measuring and monitoring consistency is given by Bermbach and Kuhlenkamp [37].

VII. Future work

We are working on additional workloads that will target specific anomalies that are observed at various transaction isolation levels [26] and develop measures to quantify these. We will run these against our client coordinated transaction library and distributed key-value store as well as publicly available cloud services like Google Cloud Storage (GCS) and Windows Azure Storage (WAS).

We intend to release the source code for these workloads and the enhancements made to the YCSB+T framework. We will explore the possibility of incorporating them into the main YCSB source tree so that the greater community can benefit. We will also explore ways to integrate with YCSB++ for

its distributed client execution, coordination and monitoring capabilities that are useful for running web-scale simulations against web-scale transactional NoSQL key-value stores.

VIII. Conclusions

We have presented YCSB+T, an extension of the Yahoo! Cloud Services Benchmark (YCSB), with the ability to wrap multiple database operations into transactions that introduces a validation stage that executes after running the workload.

Further, we described a workload called the Closed Economy Workload (CEW) that is used to evaluate the performance of a data store using that consists of read and read-modify-write operations that simulate an application scenario and later validates the consistency of the data and quantifies the anomalies if detected.

Our benchmark is suitable to performance test systems providing transactions in cloud-based NoSQL systems. We use it to measure the performance and validate the correctness of our own key-value store and our client coordinated transaction protocol and library that provides transactional access to it.

YCSB+T can be used to perform an apples-to-apples comparison between competing data storage solutions and enables the application developer to define a workload that simulates the application closely. The workload validation stage can be used to validate the consistency guarantees of the system and quantify the anomalies detected.

YCSB+T is completely backward compatible with YCSB enabling existing benchmark code to run without any modification enabling existing users to easily migrate their benchmarks.

References

[1] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, pp. 9–11, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1364782.1364786

[2] "Amazon S3 API Reference," 2011. [Online]. Available: http://docs.amazonwebservices.com/AmazonS3/latest/API/

[3] F. Chang *et al.*, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1145/1365815.1365816

[4] B. F. Cooper, R. Ramakrishnan *et al.*, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, pp. 1277–1288, August 2008. [Online]. Available: http://dx.doi.org/10.1145/1454159.1454167

[5] W. Vogels, "Eventually consistent," *Queue*, vol. 6, pp. 14–19, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1466443.1466448

[6] B. Calder *et al.*, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *SOSP'11*, 2011, pp. 143–157. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043571

[7] "Google Cloud Storage," 2013. [Online]. Available: https://developers.google.com/storage/docs/overview

[8] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI'10*, 2010, pp. 1–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924961

[9] S. Das *et al.*, "G-Store: a scalable data store for transactional multi key access in the cloud," in *SoCC '10*, 2010, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807157

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

[11] J. J. Levandoski *et al.*, "Deuteronomy: Transaction support for cloud data," in *CIDR '11*.

[12] J. Baker *et al.*, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," in *CIDR*, Jan. 2011, pp. 223–234.

[13] J. C. Corbett, J. Dean *et al.*, "Spanner: Google's globally-distributed database," in *OSDI '12*, 2012, pp. 251–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387905

[14] Transaction Processing Performance Council, "TPC-C Benchmark Specification," http://www.tpc.org/tpcc, 2005.

[15] ——, "TPC-E Benchmark Specification," http://www.tpc.org/tpce, 2007.

[16] B. F. Cooper, A. Silberstein *et al.*, "Benchmarking cloud serving systems with YCSB," in *SoCC '10*, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[17] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 7–. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[18] G. DeCandia *et al.*, "Dynamo: amazon's highly available key-value store," in *SOSP '07*, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[19] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[20] J. Rao *et al.*, "Using paxos to build a scalable, consistent, and highly available datastore," *Proc. VLDB Endow.*, vol. 4, pp. 243–254, January 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1938545.1938549

[21] W. Lloyd *et al.*, "Don't settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS," in *SOSP '11*, Oct. 2011.

[22] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *USENIX ATC'12*, 2012, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342842

[23] M. Brantner, D. Florescu *et al.*, "Building a database on S3," in *SIGMOD '08*, 2008, pp. 251–264. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376645

[24] R. Escriva, B. Wong *et al.*, "Warp: Multi-Key Transactions for Key-Value Stores," United Networks, LLC, Tech. Rep., 05 2013. [Online]. Available: http://hyperdex.org/papers/warp.pdf

[25] D. B. Lomet, A. Fekete *et al.*, "Unbundling transaction services in the cloud." in *CIDR '09*, 2009. [Online]. Available: http://dblp.uni-trier.de/db/conf/cidr/cidr2009.html#FWZ09

[26] H. Berenson *et al.*, "A critique of ANSI SQL isolation levels," in *SIGMOD '95*, 1995, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/223784.223785

[27] F. Junqueira *et al.*, "Lock-free transactional support for large-scale storage systems," in *IEEE/IFIP DSN-W '11*, June 2011, pp. 176 –181.

[28] A. Dey, A. Fekete, and U. Röhm, "Scalable transactions across heterogeneous nosql key-value data stores," *Proc. VLDB Endow.*, vol. 6, no. 12, pp. 1434–1439, Aug. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2536274.2536331

[29] J. Gray, *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[30] S. Patil, M. Polte *et al.*, "YCSB++: benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 9:1–9:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038925

[31] A. J. Elmore, S. Das *et al.*, "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *SIGMOD Conference*, T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds. ACM, 2011, pp. 301–312.

[32] S. Patterson, A. J. Elmore *et al.*, "Serializability, not serial: concurrency control and availability in multi-datacenter datastores," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1459–1470, Jul. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2350229.2350261

[33] D. Shasha and P. Bonnet, *Database tuning: principles, experiments, and troubleshooting techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[34] H. Wada, A. Fekete *et al.*, "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective," in *CIDR*, 2011, pp. 134–143.

[35] A. Fekete, S. Goldrei *et al.*, "Quantifying isolation anomalies," in *Proceedings of Very Large Databases (VLDB'09)*, 2009, pp. 467–478.

[36] K. Zellag and B. Kemme, "How *consistent* is your cloud application?" in *Proceedings of ACM Symposium on Cloud Computing (SoCC'12)*, 2012, p. 6.

[37] D. Bermbach and J. Kuhlenkamp, "Consistency in distributed storage systems - an overview of models, metrics and measurement approaches," in *International Conference on Networked Systems (NETYS'13)*, 2013, pp. 175–189, published in Springer LNCS 7853.