YCSB++ : Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores

Swapnil Patil[1], Milo Polte[1], Kai Ren[1], Wittawat Tantisiriroj[1], Lin Xiao[1],
Julio López[1], Garth Gibson[1], Adam Fuchs[2], Billie Rinaldi[2]

[1]*Carnegie Mellon University*, [2]*National Security Agency*
http://www.pdl.cmu.edu/ycsb++/

ABSTRACT

Inspired by Google's BigTable, a variety of scalable, semi-structured, weak-semantic table stores have been developed and optimized for different priorities such as query speed, ingest speed, availability, and interactivity. As these systems mature, performance benchmarking will advance from measuring the rate of simple workloads to understanding and debugging the performance of advanced features such as ingest speed-up techniques and function shipping filters from client to servers. This paper describes YCSB++*, a set of extensions to the Yahoo! Cloud Serving Benchmark (YCSB) to improve performance understanding and debugging of these advanced features.* YCSB++ *includes multi-tester coordination for increased load and eventual consistency measurement, multi-phase workloads to quantify the consequences of work deferment and the benefits of anticipatory configuration optimization such as B-tree pre-splitting or bulk loading, and abstract APIs for explicit incorporation of advanced features in benchmark tests. To enhance performance debugging, we customized an existing cluster monitoring tool to gather the internal statistics of* YCSB++*, table stores, system services like HDFS, and operating systems, and to offer easy post-test correlation and reporting of performance behaviors.* YCSB++ *features are illustrated in case studies of two BigTable-like table stores, Apache HBase and* Accumulo*, developed to emphasize high ingest rates and fine-grained security.*

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*performance evaluation*; H.2.4 [**Database Management**]: Systems—*distributed and parallel databases*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools, diagnostics*

Keywords

Scalable Table Stores, Benchmarking, YCSB, NoSQL

SOCC '11, October 27–28, 2011, Cascais, Protugal

1. INTRODUCTION

The past few years have seen an emergence of large-scale table stores that are more simple and lightweight, and provide higher scalability and availability than traditional relational databases [11, 46]. Table stores, such as BigTable [12], Dynamo [17], HBase [27] and Cassandra [1, 33], are an intrinsic part of Internet services. Not only are these stores used by data-intensive applications, such as business analytics and scientific data analysis [8, 45], but they are also used by critical systems infrastructure; for example, the next generation Google file system, called Colossus, stores all file system metadata in BigTable [20].

This growing adoption, coupled with spiraling scalability and tightening performance requirements, has led to the inclusion of a range of (often re-invented) optimization features that significantly increase the complexity of understanding the behavior and performance of the system. Table stores that began with a simple table model and single-row transactions have extensions with new mechanisms for consistency, bulk insertions, concurrency, data partitioning, indexing, and query analysis.

A key functionality enhancement for applications that continuously capture petabytes into a table is to increase the speed of ingest [45]. Typically data is ingested in a table using iterative insertions or bulk insertions. Iterative insertions add new data through single row "insert" or "update" operations that are often optimized using techniques such as client-side buffering, disabling logs [35, 43], relying on fast storage devices [49], and indexing structures optimized for high-speed inserts [23–25, 38]. Bulk loads bypass the regular insertion code path by converting existing datasets from their external storage format to the format of the native table store so that insertion bypasses the normal insert code path. Proposals to speed up bulk loading include using optimization frameworks to pre-split partitions [47] and running Hadoop jobs to parallelize data loading [5, 28].

Another useful feature is the ability to run distributed computations directly on data stored at table store servers instead of clients. BigTable co-processors allow arbitrary application code to run directly on tablet servers even when the table is growing and expanding over multiple servers [8, 15]. HBase plans to use a similar technique for server-side filtering and fine-grained access control [30, 32, 40]. Such a server-side execution model, inspired from early work in parallel databases [18], is designed to drastically reduce the amount of data shipped to the client. This significantly improves performance, particularly of scan operations with an application-defined filter.

Extensions to the YCSB framework	Observations in HBase and ACCUMULO
Distributed testing using multiple YCSB client nodes	
ZooKeeper-based barrier synchronization for multiple YCSB clients to coordinate start and end of different tests	Distributed setup benefits multi-client, multi-phase testing (to evaluate weak consistency and table pre-splits)
Distributed event notification using ZooKeeper to understand the cost (measured as read-after-write latency) of weak consistency	Both HBase and ACCUMULO support strong consistency, but using client-side batch writing for higher throughput results in weak consistency with higher read-after-write latency as batch sizes increase
Ingest-intensive workload extensions	
External Hadoop tool that formats data to be inserted into a format used natively by the table store servers	Bulk insertion delivers the highest data ingest rate of all ingestion techniques, but the servers may end up doing expensive load-balancing
A new workload executor for externally pre-splitting the key space into variable-sized and fixed-size ranges.	Ingest throughput of ACCUMULO increases by 20% but if range partitioning is not known a priori the servers may incur expensive re-balancing and merging overhead
Offloading functions to the DB servers	
New workload executor that generates "deterministic" data to allow use of appropriate filters and DB client API extensions to send filters to servers	Server-side filtering benefits HBase and ACCUMULO only when the client scans enough data (more than 10 MB) to mask network and disk I/O overhead
Fine grained access control	
New workload generator and API extensions to DB clients to test both schema-level and cell-level access control models (HBase does not support access control [27] but ACCUMULO does)	ACCUMULO's access control increases the size of the table and may reduce insert throughput (if client CPU is saturated) or scan throughput (when server returns ACLs with the data) in proportion to controls imposed

Table 1: ***Summary of contributions – For each advanced functionality that* YCSB++ *benchmarks, this table describes the techniques implemented in YCSB and the key observations from our HBase and* ACCUMULO *case studies.***

The profusion of table stores calls for developing effective benchmarking tools, and the Yahoo! Cloud Serving Benchmark (YCSB) has answered this call successfully. YCSB is a great framework for measuring the basic performance of several popular table stores including HBase, Voldemort, Cassandra and MongoDB [14]. YCSB has an abstraction layer for adapting to the API of a specific table store, for gathering widely recognized performance metrics and for generating a mix of workloads. Although it is useful for characterizing the baseline performance of simple workloads, such as single-row insertions, lookups or deletions, YCSB lacks support for benchmarking advanced table store functionality. Advanced features make a table store attractive for a wide range of use cases, but their complex interactions can be very hard to benchmark, debug and understand, especially when a store exhibits poor performance.

Our goal is to extend the scope of table store benchmarking in YCSB to support complex features and optimizations. In this paper, we present a systematic approach to benchmark advanced functionality in a distributed manner, and implement our techniques as extensible modules in the YCSB framework. We do not modify the table stores under evaluation, but the abstraction layer adapting a specific table store to a benchmarking oriented API for a specific advanced function may be simple or complex, depending on the capabilities of the underlying table store.

Table 1 summarizes the key contributions of this paper. The first contribution is a set of benchmarking techniques to measure and understand five advanced features: weak consistency, bulk insertions, table pre-splitting, server-side filtering and fine-grained access control. The second contribution is implementing these techniques, which we collectively call YCSB++, as extensible modules in the YCSB framework. Our final contribution is the experience of analyzing these features in two table stores, HBASE [27] and ACCUMULO, both inspired by BigTable and exhibiting most or all of YCSB++ features.

2. YCSB++ DESIGN

In this section, we present an overview of table stores, including HBase and ACCUMULO, followed by the design and implementation of advanced functionality benchmarking techniques in YCSB++.

2.1 Overview of table stores

HBase and ACCUMULO are scalable semi-structured table stores that store data in a multi-dimensional sorted map where keys are tuples of the form {row, column, timestamp}. Both are inspired by Google's BigTable system [12]. HBase is being developed as a part of the open-source Apache Hadoop project [26, 27] and ACCUMULO is being developed by the U.S. National Security Agency.[1] Both are written in Java and layered on top of the Hadoop distributed file system (HDFS) [6]. They support efficient storage and retrieval of structured data, including range queries, and allow using tables as input and output for MapReduce jobs. Other

[1]An open-source release of ACCUMULO has been offered to the Apache Software Foundation.

features in these systems pertinent to YCSB++ include automatic load-balancing and partitioning, data compression and server-side user-defined function such as regular expression filtering. To avoid confusion from terminology differences in HBase and ACCUMULO, the rest of this paper uses terminology from the Google BigTable paper [12].

At a high-level, each table is indexed as a B-tree in which all records are stored in leaf nodes called *tablets*. An HBase or ACCUMULO installation consists of *tablet servers* running on all nodes in the cluster, and each tablet server handles requests for several tablets. A tablet consists of rows in a contiguous range in the key space and is represented (on disk) as one or more files stored in HDFS. Each table store represents these files in their respective custom formats (BigTable uses an SSTable format, HBase uses an HFile format and ACCUMULO uses an RFile format) which we will refer as *store files*. In all cases, store files are sorted, indexed, and used with bloom filters to make negative lookups faster [12]. Both HBase and ACCUMULO provide columnar abstractions that allow users to group a set of columns into a *locality group*. Each locality group is stored in its separate store file in HDFS; this enables efficient scan performance by avoiding excess data fetches (from other columns) [48]. These table stores use a master server that manages schema details and assigns tablets to tablet servers in a load-balanced manner.

When a table is first created, it has a single tablet, the root of the B-tree, managed by one tablet server. Inserts are sent to an appropriate tablet server guided by the cached state about non-leaf nodes of the B-tree. The leaf tablet server logs mutation operations and buffers all requests in an in-memory buffer called *memstore*. When this memstore fills up, the tablet server flushes recently written entries to create a store file in HDFS; this process is called *minor compaction*. As the table grows, the memstore fills up again and is flushed to create another store file. Reads not specified in a memstore may have to search many store files for the requested entries. This use of multiple store files representing mutations from a particular time period is inspired by the classic log-structured merge tree (LSM-tree) [38]. Once a tablet exceeds a threshold size, the tablet server splits the overflowing tablet (and its key range) by creating a new tablet on another tablet server and transferring the rows that belong to the key range of the new tablet. This process is called a *split*. A large table may have large number of tablets and each tablet may have many store files. To control the number of store files that may be accessed to service a read request, *major compaction* operations are used to merge store files into fewer store files. All files are stored in HDFS and these table stores rely on HDFS for durability and availability of data.

2.1.1 *Additional features in* ACCUMULO

The design and implementation of ACCUMULO has several features that are different from other open-source table stores. Perhaps the most unique feature in ACCUMULO is the *iterator framework* that embeds user-programmed functionality into the different LSM-tree stages. Figure 1 shows how iterators fit in the tablet server architecture of ACCUMULO and enable in-situ processing during otherwise necessary I/O operations. For example, iterators can operate during minor compactions by using the memstore data as input to generate on-disk store files comprised of some transformation of the input such as statistics or additional indices.

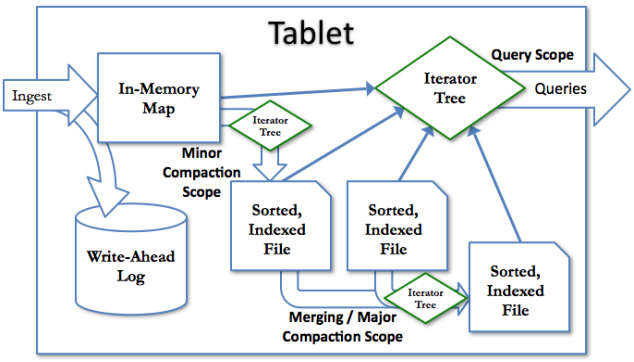

Figure 1: ***Design of the*** ACCUMULO ***tablet server and its use of iterators for bulk and stream processing.***

Iterators read one or more sorted sequences of key-value pairs and output an ordered stream of key-value pairs. The input stream can be transformed in various ways depending on the context of transformation used by an iterator. The context of an iterator is the degree of commonality among the key-value pairs that it uses as input for a given operation. For example, a version context is defined as a set of key-value pairs that share the same row and column but have different timestamps and values. A versioning iterator operates within that version context and pares down the input set to the N key-value pairs with the most recent timestamps. An aggregating iterator also operates within a version context, and it replaces all key-value pairs in the set with a new key-value pair whose value is an aggregate function (e.g. sum) of the key-value pairs in the context.

ACCUMULO can also create iterator trees by chaining different iterators together such that the output from one iterator serves as the input to another iterator. These iterators can be organized as a hierarchy, comprising of parent and child iterators that can create a user-defined data processing pipeline to support stream processing, incremental bulk processing, and partitioned join operations. Iterators can perform the basic operations of a query language, such as selection, projection, and set intersection and union within a partition. Trees of these types of iterators are used to implement scalable information retrieval systems with highly expressive query languages. Other user-defined iterators can be used to efficiently encode complex, online statistical aggregation functions that are crucial for big-data analytics.

Another feature unique to ACCUMULO is fine-grained cell-level access control. To the best of our knowledge, ACCUMULO is the only table store that provides cell-level access control by associating an access control list (ACL) with every cell. This is different from Bigtable, which uses table-level and column family-level access control mechanisms [12]. HBase proposes to support a coarse-grained schema-level access control mechanism that will store and check the ACLs only at a schema (or metadata) level [30]. Currently, ACCUMULO uses cell-level access control only for reads and additional schema-level access control for both read and write operations. To support cell-level ACLs, ACCUMULO uses a key specifier comprised of the tuple {row, column family, column qualifier, visibility, timestamp} where the visibility portion is an encoded and-or tree of authorizations.

The authors of ACCUMULO report that it has been demon-

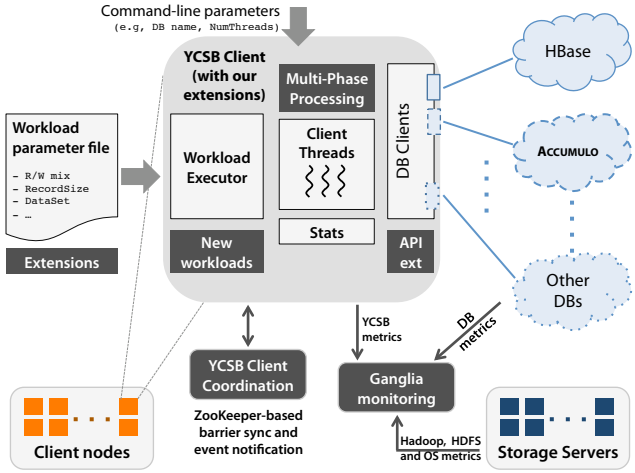

Figure 2: YCSB++ ***functionality testing framework – Light colored boxes show modules in YCSB v0.1.3 [14] and dark shaded boxes show our new extensions.***

strated across diverse hardware configurations and multiple levels of scale and, in a variety of usability tests, it has been successful at handing very complex data-sets with high-speed, efficient ingest and concurrent query workloads. An open-source release of Accumulo has been offered to the Apache Software Foundation.

2.2 YCSB background

The Yahoo! Cloud Serving Benchmark (YCSB) is a popular extensible framework designed to compare different table stores under identical synthetic workloads [14]; the different modules in YCSB are shown as light boxes in Figure 2.

The **workload executor** module loads test data and generates operations that will be specialized and issued by a **DB client** to a table store. The default YCSB workload issues mixes of basic operations including reads, updates, deletes and scans. In YCSB, read operations may `read()` a single row or `scan()` a range of consecutive rows and update operations may either `insert()` a new row or `update()` an existing one. Operations are issued one at a time per client thread and their distributions are based on parameters specified in the **workload parameter file** for a benchmark. The YCSB distribution includes five default workload files (called *Workloads A, B, C, D and E*) that generate specific read-intensive, update-intensive and scan-intensive workloads.

The current YCSB distribution provides **DB client** modules with wrappers for HBase, Cassandra [1], MongoDB [2] and Voldemort [3]; YCSB++ adds a new client for Accumulo. For a given table store, its DB client converts a 'generic' operation issued by the workload executor to an operation specific for that table store. In an HBase cluster, for example, if the workload executor generates a `read()` operation, the HBase DB client issues a `get()` operation to the HBase servers.

YCSB starts executing a benchmark using a pool of **client threads** that call the workload executor to issue operations and then report the measured performance to the **stats** module. Users can specify the size of the work generating thread pool, the table store being evaluated and the workload parameter file as **command line parameters**.

2.3 Extensions in YCSB++

YCSB's excellent modular structure makes it natural for us to integrate advanced functionality testing mechanisms as YCSB extensions. Our YCSB++ extensions are shown as dark shaded boxes in Figure 2.

2.3.1 *Parallel testing*

The first extension in YCSB++ enables multiple clients, on different machines, to coordinate start and end of benchmarking tests. This modification is necessary because YCSB was designed to run on a single node and just one instance of YCSB, even with hundreds of threads, may limit its ability to test large deployments of table stores effectively. YCSB++ controls execution of different workload generator instances through distributed coordination and event notification using Apache ZooKeeper, a service that provides distributed synchronization and group membership [29, 52]. ZooKeeper is already used in HBase and Accumulo deployments.

YCSB++ implements a new class, called `ZKCoordination`, that provides two abstractions – barrier-synchronization and producer-consumer – through ZooKeeper. We added four new parameters to the workload parameter file: a status flag, the ZooKeeper server address, a barrier-sync variable, and the size of the client coordination group. The status flag checks whether coordination is needed among the clients. Each coordination instance has a unique barrier-sync variable to track the number of processes entering or leaving a barrier. ZooKeeper uses a hierarchical namespace for synchronization and, for each barrier-sync variable specified by YCSB++, creates a corresponding "barrier" directory in its namespace. Whenever a new YCSB++ client starts, it joins the barrier by contacting the ZooKeeper server that in turn creates a new entry, corresponding to the client's identifier, in the barrier directory. The number of entries in a barrier directory indicates the number of clients that have joined the barrier. If all the clients have joined the barrier, ZooKeeper sends these clients a callback message to start executing the benchmark; if not, YCSB++ clients block and wait for more clients to join. After the test (or one phase) completes, YCSB++ clients notify ZooKeeper about leaving the barrier.

2.3.2 *Weak consistency*

Table stores provide high throughput and high availability by eliminating expensive features, particularly the strong ACID transactional guarantees found in traditional relational databases. Based on the **CAP** theorem, some table stores tolerate network **P**artitions and provide high **A**vailability by giving up on strong **C**onsistency guarantees [7, 22]. Systems may offer "loose" or "weak" consistency semantics, such as eventual consistency [17, 50], in which acknowledged changes are not seen by other clients for significant time delays. This lag in change visibility may introduce challenges that programmers may need to explicitly handle in their applications (i.e., coping with possibly stale data). YCSB++ measures the time lag from one client completing an insert until a different client can successfully observe the value.

To evaluate this time to consistency, YCSB++ uses asynchronous directed coordination between multiple clients enabled by the producer-consumer abstraction in the aforementioned `ZKCoordination` module. YCSB++ clients interested in benchmarking weak consistency specify three properties in the workload parameter file: a status flag to

check if a client is a producer or a consumer, the ZooKeeper server address, and a reference to a shared queue data-structure in ZooKeeper. Synchronized access to this queue is provided by ZooKeeper: for each queue, ZooKeeper creates a directory in its hierarchical namespace and adds (or removes) a file in this directory for every key inserted in (or deleted from) the queue. Clients that insert or update records are "producers" who add keys of recently inserted records in the ZooKeeper queue. The "consumer" clients register a callback on this queue at start-up. On receiving a notification from ZooKeeper about new elements, "consumers" remove a key from the queue then read it from the table store. If the attempt to read this key fails, the "consumer" will put the key back on the queue and try reading the next available key. Excessive use of ZooKeeper for inter-client coordination may affect the performance of the benchmark; we avoid this issue by sampling a small fraction (1%) of the inserted keys for read-after-write measurements. The "read-after-write" time lag for key K is the difference from the time a "consumer" first tries to read the new key until the first time it successfully reads that key from the table store server; we only report the lag for keys that needed more than one read attempt. We did not measure the time from "producer" write to "consumer' read in order to avoid cluster-wide clock synchronization challenges.

2.3.3 Table pre-splitting for fast ingest

Recall that both HBase and Accumulo distribute a table over multiple tablets. Because these stores use B-tree indices, each tablet has a key range associated with it and this range changes when a tablet overflows to split into two tablets. These split operations limit the performance of ingest-intensive workloads because table store implementations lock a tablet during splits and migrate a large amount of data from one tablet server to another on a different machine. During this migration, servers refuse any operation (including reads) addressed to the tablet undergoing a split (until it finishes). One way to reduce this splitting overhead is to split a table when it is empty or small into multiple key ranges based on a priori knowledge, such as key distributions, of the workload; we call this pre-splitting the table.

YCSB++ adds to the DB clients module a pre-split function that takes split points as input and invokes the servers to pre-split a table. To enable pre-splits in a benchmark, YCSB++ adds a new property in the workload parameter files that can specify either a list of variable-size ranges in the key space or a number of fixed-size partitions to divide the key space.

2.3.4 Bulk loading using Hadoop

To efficiently add massive data-sets, various table stores rely on specialized, high-throughput tools and interfaces [28]. In addition to the normal insert operations, YCSB++ supports the use of these specialized bulk load mechanisms. YCSB++ invokes an external tool that directly processes the incoming data, stores it in an on-disk format native to the table store, and notifies the servers about the existence of the new and properly formatted files through an `import()` API call. Table store servers make the newly loaded data-set available after successfully updating internal data-structures.

Developers can create bulk loader adaptors for particular table stores by providing specific implementations for two YCSB++ components: data transformation and import operation adaptors. For the data transformation component, YCSB++ expects a Hadoop application for partitioning, potentially sorting, and storing the data in the appropriate format. The implementation of the import operation loads the formatted data using the specific interface for the particular table store. YCSB++ also implements a generic Hadoop data generator for bulk load benchmarks that can be extended and adapted to a particular store by implementing the corresponding output format adaptor in the tool.

2.3.5 Server-side filtering

Server-side filtering offloads compute from the client to the server, possibly reducing the amount of data transmitted over the network and amount of data fetched from disk. In order to reduce the amount of data fetched from disk, YCSB++ includes the ability to break columns into locality groups. Since locality groups are often stored in separate files by tables stores, filters that test and return data from only some locality groups do less work [48]. YCSB++ takes a workload parameter causing each column to be treated as a single locality group.

There are a wide range of server-side filters that could be supported and scalable table stores filtering implementations are often not as expressive as SQL. For YCSB++ we define four server-side filters, exploiting regular expressions for "pattern" parameters, that are significantly different and are supported in both HBase and Accumulo. The first filter returns the entire row if the value of the row's key matches a specified pattern, the second filter returns the entire row if the value of the row's entry for a specified column name matches a specified pattern, the third filter returns the row's key and the {column name, cell value} tuple where column name matches a specified pattern, and the fourth filter returns the row's key and the {column name, cell value} tuple where any column's entry value matches a specified pattern.

Each table store's DB client implements these four filters in whatever manner is best supported by the table store under test. That is, if the table store does not have an API capable of function shipping the filter to the server, it could fetch all possibly matching data and implement the filter in the DB client.

2.3.6 Access control

Table stores support different types of access control mechanisms, including none at all, checks applied at the level of the entire table, checks applied conditionally to each column, column family or locality group, or checks applied to every cell. Checks applied only to the entire table or specific column sets are said to be schema-level access controls, while checks applied to every cell are said to be cell-level access controls. HBase developers are working on schema-level access control, although the main release of Hbase has no security [30]. Accumulo implements both, using schema-level access control on all accesses and cell-level access controls on read accesses.

YCSB++ supports tests that specify credentials for each operation and access control lists (ACLs) to be attached to schema or cells. The DB client code for each table store implements operations specific to a credential used or an ACL set in the manner best suited to that table store. The goal of YCSB++ access control tests is to evaluate the performance consequences of using access control; our tests ex-

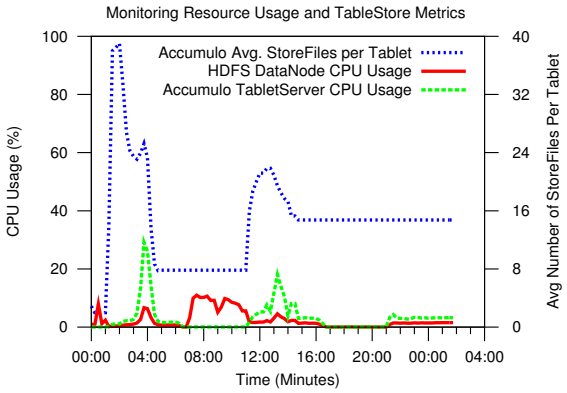

Figure 3: ***An example of combining different logical aggregates in Otus graphs.***

aggerate the use of ACLs relative to table data to make performance trends more noticeable, not because we believe that this heavy use of ACLs is common.

2.4 Performance monitoring in YCSB++

There are many tools for cluster-wide monitoring and visualization such as Ganglia [37], Collectd [13], and Munin [39]. These tools are designed for large scale data gathering, transport, and visualization. They make it easy to view application-agnostic metrics, such as aggregate CPU load in a cluster, but they lack support for application-specific performance monitoring and analysis. For example, virtual memory statistics for the sum of all processes running on a node or cluster are typically recorded, but we think a more useful approach is to report aggregate memory usage of a MapReduce task separate from that used by tablet servers, HDFS data servers and other non-related processes.

YCSB++ uses a custom monitoring tool, called Otus [41], that was built on top of Ganglia. Otus runs a daemon process on each cluster node that periodically collects metrics from the node's OS, from different table store components such as tablet servers and HDFS data nodes, and from YCSB++ itself. All collected metrics are stored in a central repository; users can process and analyze the collected data using a tailored web-based visualization system.

In Otus, OS-level resource utilization for individual processes is obtained from the Linux `/proc` file system; these metrics include per-process CPU usage, memory usage, and disk and network I/O activities. By inspecting command-line invocation data from `/proc` and aggregating stats for process groups derived from other invocations, Otus differentiates logical functions in a node. Table store related metrics, such as the number of tablets and store files, are extracted directly from the table store services to provide information about the inner workings of these systems. Otus can currently extract metrics from HBase and Accumulo, and adding support for another table store involves writing Python scripts to extract the desired metrics in whatever manner that table store uses to dynamically report metrics [41]. We also extended the YCSB++ stats module to periodically send (using UDP) performance metrics to Otus.

By storing the collected data in a central repository and providing a flexible web interface to access the benchmark data, users can obtain and correlate fine-grained time series information of different metrics coming from different service layers and within a service. Figure 3 shows a sample output from Otus that combines simultaneous display of three metrics collected during an experiment: HDFS data node CPU utilization, tablet server CPU utilization and the number of store files in the system.

3. ANALYSIS

All our experiments are performed on sub-clusters of the 64-node "OpenCloud" cluster at CMU. Each node has a 2.8 GHz dual quad core CPU, 16 GB RAM, 10 Gbps Ethernet NIC and four Seagate 7200 RPM SATA disk drives. These machines were drawn from two racks of 32 nodes each with an Arista 7148S top-of-the-rack switch. Both rack switches are connected to an Force10 4810 head-end switch using six 10 Gbps uplinks each. Each node was running Debian Lenny 2.6.32-5 Linux distribution with the XFS file system managing the test disks.

Our experiments were performed using Hadoop-0.20.1 (that includes HDFS) and HBase-0.90.2 which use the Java SE Runtime 1.6.0. HDFS was configured with a single dedicated metadata server and 6 data servers. Both HBase and Accumulo were running on this HDFS configuration with one master and 6 region servers – a configuration similar to the original YCSB paper. [14]. The test data in these table stores was stored in table that used the default YCSB schema where each row is 1 KB in size and comprises of ten columns of 100 bytes each; this schema was used for all experiments except server-side filtering (in Section 3.5) and access control (in Section 3.6).

The rest of this section shows how YCSB++ was used to study the performance behavior of advanced functionality in HBase and Accumulo. We use the Otus performance monitor (Section 2.4) to understand the observed performance of all software and hardware components in the cluster.

3.1 Effect of batch writing

Both HBase and Accumulo coalesce application writes in a client-side buffer before sending them to a server because batching multiple writes together improves the write throughput by avoiding a round-trip latency in sending each write to the server. To understand the benefits of batching for different write buffer sizes, we configure two 6-node clusters, one for HBase and other for Accumulo, that are both layered on an HDFS instance. We use 6 separate machines as YCSB++ clients that insert 9 million rows each in a single table; the YCSB++ clients for Accumulo use 50 threads each, while the YCSB++ clients for HBase use 4 threads each.[2]

Figure 4 shows the insert throughput (measured as the number of rows inserted per second) with four different batch sizes. All numbers are an average of two runs with negligible variance. Results are most dramatic for Accumulo, where more than a factor of two increase in insert throughput can be obtained with larger write batching, but HBase also sees almost a factor of two increase with large batch size when the offered load from the client is large.

Graphs like Figure 4 are useful to the developers of a ta-

[2]HBase, when configured with 50 threads per client, was unable to complete the test successfully without crashing any server during the test.

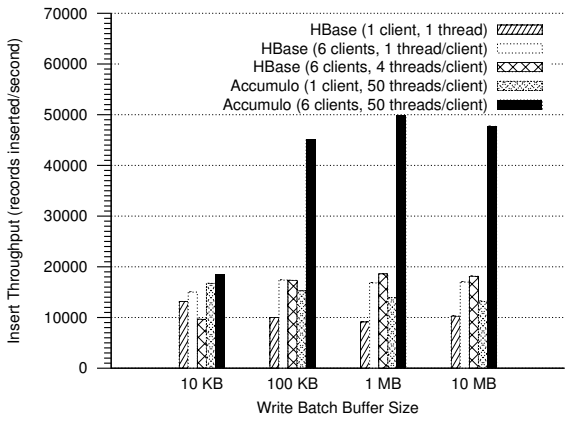

Figure 4: ***Effect of batch size on insert rate in a 6-node HBase and*** ACCUMULO ***cluster.***

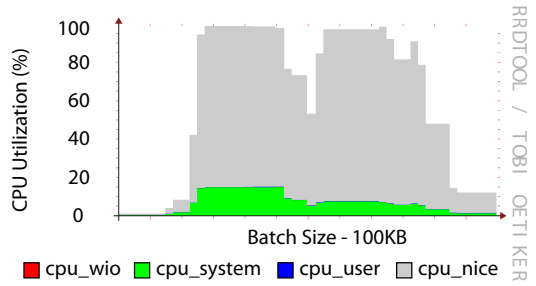

Figure 5: ***A single client inserting records in a 6-node*** ACCUMULO ***cluster becomes CPU limited resulting in underutilized servers and low overall throughput (Figure 4).***

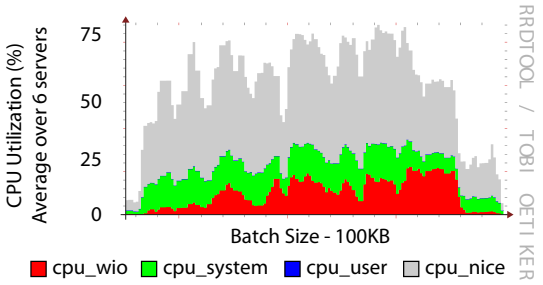

Figure 6: ***Six* Accumulo *servers begin to saturate when 300 threads (spread on six clients) insert records at maximum speed using a 100 KB batch buffer (Figure 4).***

ble store both to confirm that a mechanism such as batch writing achieves greater insert throughput and to point out where other effects impact the desired result. For example, HBase with 10KB batches sees lower throughput at higher offered load and ACCUMULO with 1 client and 50 threads aggregate sees slightly decreasing throughput with larger batches. Figure 5 begins to shed light on the latter situation; 9 million inserts of 1 KB rows in batches of 10 KB (10 rows) to 10 MB (10,000 rows) fully saturates the client most of the time, so little throughput can be gained from more efficiency in the server or lower per insert latency. In fact, the two periods of significant decrease in utilization in the client suggests looking more deeply at non-continuous processes in the server (such as tablet splitting and major compactions of store files, which, for example, are seen to be large sources of slowdown in Section 3.4).

Consider the most significant throughout change in Figure 4, ACCUMULO with high offered load sees its throughout increase from near 20,000 rows per second to over 40,000 rows per second when the batch size goes from 10 KB to 100 KB then sees only small increases for larger batches. Figures 6 shows how the server CPU utilization with 100 KB batches is approaching saturation, reducing the benefit of larger batches from both increasing the client efficiency at generating load and increasing the server efficiency at processing load to only increasing throughput with increased server efficiency.

3.2 Weak consistency due to batch writing

Although batching improves throughput, it has an important side-effect: data inconsistency. Even for table stores like HBase and ACCUMULO that support strong consistency, newly written objects are locally buffered and are not sent to the server until the buffer is full or a time-out on the buffer expires. Such delayed writes can violate the read-after-write consistency expected by many applications, i.e. a client, who is notified by another client that some write has been completed, may fail to read the data written by that operation.

We evaluate the cost of batch writing using the producer-consumer abstraction in YCSB++ with a 2-client setup. Client C_1 inserts 1 million rows in an empty table, randomly selects 1% of these inserts and enqueues them at the ZooKeeper server. The second client C_2 dequeues keys inserted in the ZooKeeper queue and attempts to read the rows associated with those keys. We estimate the "read-after-write" time lag as the time difference between when C_2 first attempts to read a key and when it first successfully reads that key. This under-estimates by the time from write at C_1 to dequeue at C_2 and over-estimates by the time in the ZooKeeper queue of the last unsuccessful read, but neither of these should be more than a few milliseconds.

Figure 7 shows a cumulative distribution of the estimated time lag observed by client C_2 for different batch sizes. This data excludes the (zero) time lag of keys that are read successfully the first time C_2 tries to do so. Out of the 10,000 keys that C_2 tries to read, less than 1% keys experience a non-zero lag when using a 10 KB batch in both HBase and ACCUMULO. The fraction of keys that experience a non-zero lag increases with larger batch sizes: 1.2% and 7.4% of the keys experience a lag for a 100 KB batch size in ACCUMULO and HBase respectively, 14% and 17% for a 1 MB batch size, and 33% and 23% for a 10 MB batch size. This fraction of keys that see non-zero lag increases with batch size because smaller batches fill up more quickly and are flushed to the server more often, while larger batches take longer to fill and are flushed less often.

For the developer or administrator of the table store, these tests give insight into the expected scale of delayed creates. For the smallest batch size (10 KB), HBase has a median lag of 100 ms and a maximum lag of 150 seconds, while ACCUMULO has an order of magnitude higher median (about 900 ms) and an order of magnitude lower maximum lag (about

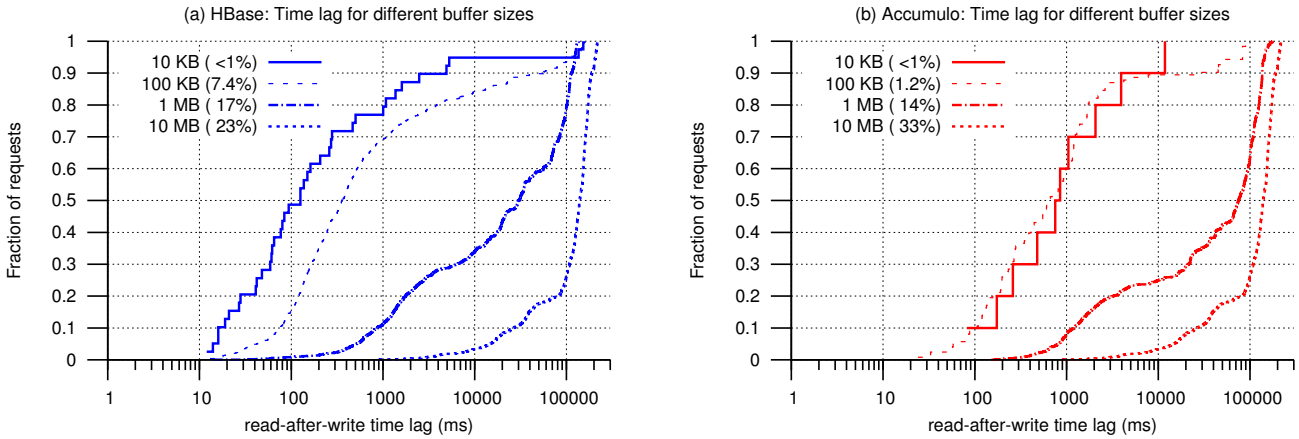

Figure 7: ***CDF of non-zero read-after-write time lag estimations for different batch sizes. The parentheses show the fraction of tests that show non-zero time lag.***

10 seconds). However, the time lag for both table stores is similar for all larger batch sizes; the largest batch size (10 MB), for example, has a median lag of approximately 140 seconds and a maximum lag of approximately 200 seconds for both HBase and ACCUMULO.

For programmers of services that use table stores, it is important to observe that large batches may cause some keys to be visible more than 100 seconds after they were written by other clients. That is, with large batched writes, programmers must be prepared to cope with read-after-write time lags in the order of minutes.

3.3 Table pre-splitting

Both HBase and ACCUMULO rely on a distributed B-tree that grows incrementally by splitting its leaf nodes (tablets) as the table grows. Because throughput degrades during tablet splits, these table stores provide an alternate interface to pre-split a table before it has much data. The goal for pre-splitting is (1) to migrate less data during ingest phase, and (2) to engage more tablet servers earlier on in an ingest-heavy workload.

We extended YCSB++'s DB client API to offer an interface to pre-split a key range into N equal sub-ranges. The idea is that N predicts the future size of the tablets covering the key range in question. If N is too small, extra splits beyond the pre-splits will be done and not all tablet servers will be engaged early in the ingest work. If N is too large, tablets and their minor compactions will be numerous and small, leading to complex interactions with the major compaction policies.

ACCUMULO has a general interface for synchronously pre-splitting the tablets covering a range at specific key values. This is fast because multiple tablets in ACCUMULO can share the same (immutable) store files. In HBase, tablet pre-splitting is deferred until the next major compaction on that tablet, and then the tablet is divided into exactly two new tablets, optimizing the work of splitting and major compaction together. Unfortunately, when the YCSB++ DB client code invokes many pre-splits in HBase and the corresponding major compactions, which is an uncommon workload for HBase, it becomes unstable. The results in this section are all taken from ACCUMULO experiments.

Phase	Phase Name	Workload
(1)	Pre-load	Pre-load 6 million rows with keys uniform in range $[0, 12 \times 10^9]$
(2)	Pre-split	Pre-split the key range $[0, 72 \times 10^6]$ into fixed-size partitions
(3)	Load	Load 48 million rows with keys uniform in range $[0, 72 \times 10^6]$
(4)	R/U Measurement 1	50% read and 50% update operations for 4 minutes with target throughput of 600 ops/sec (light load)
(5)	Pause	Sleep for 5 minutes
(6)	R/U Measurement 2	Same as Phase (4)

Table 2: ***Six-phase experiment used to study the effects of pre-splitting in HBase and*** ACCUMULO.

It is tempting to evaluate ingest speed by the time until the last client returns from submitting the last row to the table store, but this underestimates the churn the table store may continue to experience as it splits and compacts tablets after the clients think ingest is complete. One way to measure this churn is through a light load of query and update operations after the inserts are done; YCSB++ enables this through multiple phase tests, using its multi-client coordination techniques to synchronize all client threads on the current phase.

The six phases used in our pre-split experiments are described in Table 2. Phase (5) is a 300 second idle period designed to encourage a table store waiting for an idle period to do its pending work, so phase (6) repeats the light query and update load to expose the impact of such pending work. In these experiments, we use three YCSB++ clients and reduce the main memory each tablet server uses for memstore to 1 GB to engage compaction work more frequently.

Figure 8 shows the duration of Phase (1), which loads 6 million rows with keys in the $[0, 12 \times 10^9]$ range into an empty table, and Phase (3), which loads 48 million rows with keys in the $[0, 72 \times 10^6]$ range into a table that is pre-split (in Phase (2)) into different numbers of fixed-size partitions. This figure reports the "slowest" and the "fastest" completion times of the three YCSB++ clients used for this experiment. After pre-splitting the key range $[0, 72 \times 10^6]$ into 17 equal-

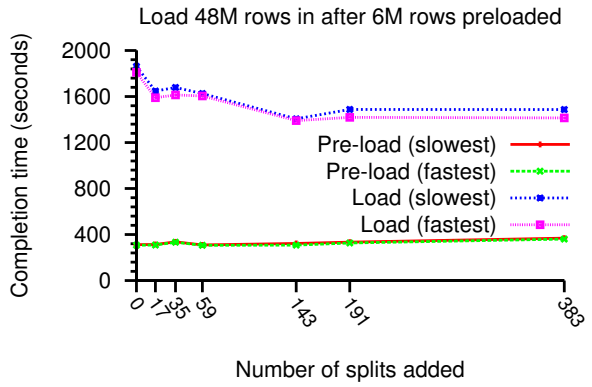

Figure 8: ***Effect of pre-splitting the*** $[0, 72 \times 10^6]$ ***key range in an*** ACCUMULO ***table into into varying number of equal-sized ranges on completion time of ingest-intensive workloads.***

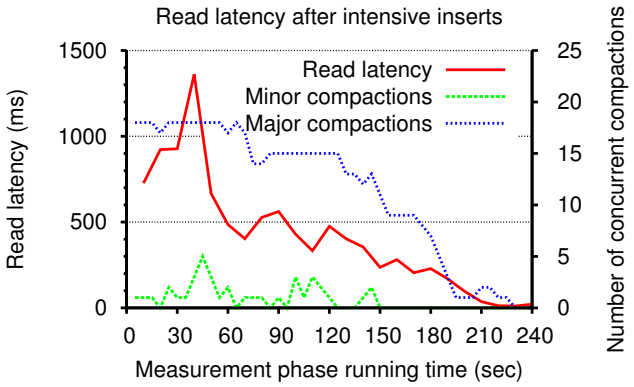

Figure 9: ***Understanding read latencies in*** ACCUMULO ***after ingest-intensive workloads and the correlations with compactions on its servers.***

sized partitions the duration of Phase (1) is reduced from from about 1,800 seconds to 1,600 seconds, and after pre-splitting it into 143 or more partitions the duration reduces even further to about 1,500 seconds – a 20% improvement in completion time. Pre-splitting the $[0, 72 \times 10^6]$ key range into less than 143 tablets does not inhibit most further splits because the split threshold is 256 MB in store files. But as little as 17 pre-splits ensures that all tablet servers are engaged during the subsequent insert phases.

The duration of insert phases is only part of the effort expended by table stores. Phases (4), (5) and (6) explore the behavior after the insert phases. In Phase (6), read latencies are all about 7-10 ms. However, read latency for operations in Phase (4), which happens immediately after the table attains 54 million rows, is dependent on compactions happening concurrently in the table store. Figure 9 shows the behavior during Phase (4) for the case when the range $[0, 72 \times 10^6]$ is pre-split into 143 equal-size ranges; this figure plots read latency with the number of compaction operations on the tablet servers (collected by the Otus performance monitor). In the first 60 seconds of this measurement phase, the read latency is always more than 500 ms and as high as 1,500 ms. These slow operations correlate with a large number of major compactions that keep tablet servers busy. As the measurement phase progresses and the compactions that the tablet servers want to do complete, read latencies start to decrease. After about 200 seconds, when the tablet servers are no longer performing any compactions, monitored read operations are taking about 7 ms, which corresponds to our observed read latencies for operations performed much later in Phase (6). This large variance in response time suggest that the policies and mechanisms of compactions, like defragmentation and cleaning in log-structured file systems [44], is important future work for table store developers.

3.4 Bulk loading using Hadoop

YCSB++ uses an external Hadoop/MapReduce (MR) tool to benchmark bulk inserts in HBase and ACCUMULO. Similar to the previous section on pre-splitting tables, we analyze the performance of bulk insertions using an eight-phase experiment shown in Table 3. The big difference between the experiments of Section 3.3 and those in this section is the replacement of an iterative call to insert one row many times with a MapReduce job that formats all data to be inserted into a native format, stored as on-disk store files, and one call to adopt these store files in a table in the store. For both HBase and ACCUMULO, our formatting tool generated store files for 36 tablets, enough to reliably load balance work to 6 tablet servers, but for HBase, a limit on the size of the native store files to be imported led us to generate 8 store files per tablet.

Phase	Phase Name	Workload
(1)	MR Pre-load	Format 6 million rows with uniform key distribution into the table store native format using a Hadoop/MapReduce job
(2)	Pre-load import	Import the on-disk data files (created in the previous phase) into an empty table in the table store
(3)	R/U Measurement 1	50% read and 50% update operations for 5 minutes with target throughput of 600 ops/sec
(4)	MR Load	Format 48 million new rows with uniform key distribution into the table store native format using a Hadoop/MapReduce job
(5)	Load import	Import the on-disk data files (created in the previous phase) into the table created in Phase (2)
(6)	R/U Measurement 2	Same as Phase (3)
(7)	Pause	Sleep for 5 minutes
(8)	R/U Measurement 3	Same as Phase (3)

Table 3: ***Eight-phase experiment used to understand bulk loading in HBase and*** ACCUMULO.

For both table stores, the time to format a bulk load is much faster than the time to insert one row at a time; this is observed from comparing the durations in Figure 8 to durations from start to P2 and from end of P3 to end of P5 in Figure 11. And since the adoption of native store files is also very fast, more interesting issues are observed in the measurement phases. Figure 10 shows the read latencies during all three measurement phases in this eight-phase experiment and Figure 11 reports the number of store files, tablets and compactions across all phases, but we distinctly highlight

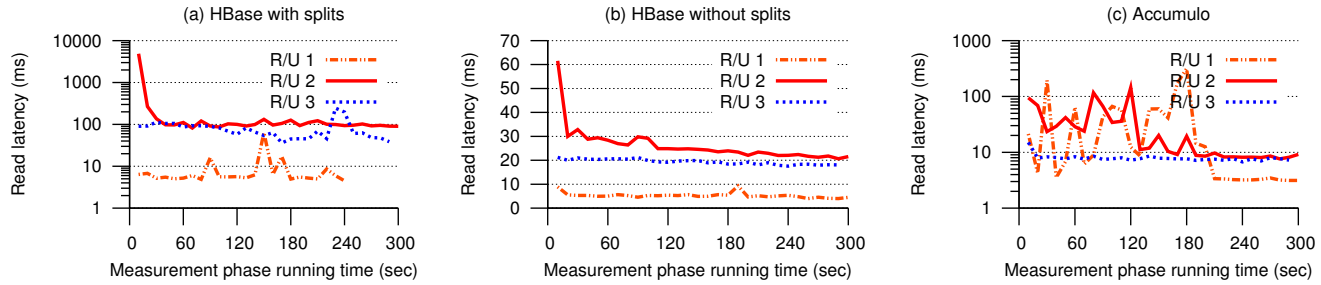

Figure 10: *Read latency during the three measurement phases in our eight-phase bulk load experiments*

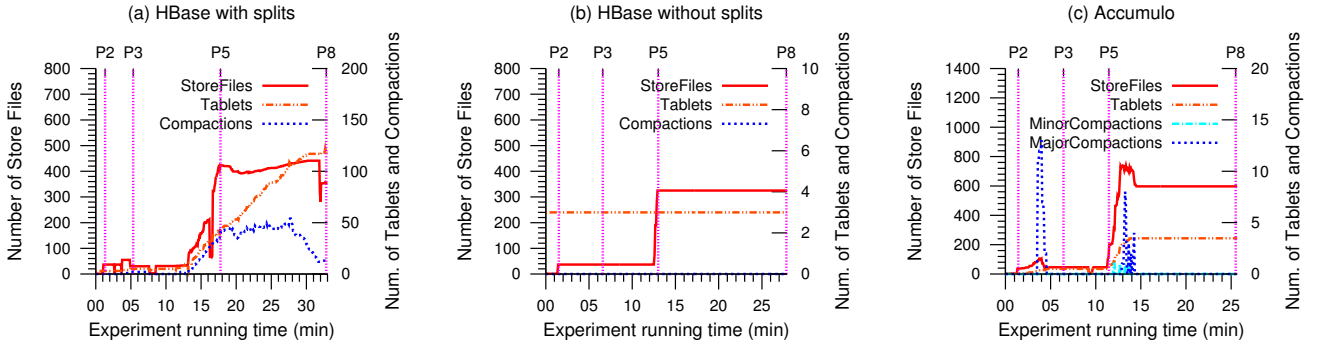

Figure 11: *Measuring the number of store files, tablets and concurrent compactions relative to the ends of Phase (2), Phase(3), Phase (5) and Phase (8) during our eight-phase bulk load experiments.*

the end of Phase (2), (3), (5) and (8). All experiments were run three times. For ACCUMULO all three runs were similar and one run in shown in the graphs. For HBase two runs were similar and one very different; we show one of each.

The two different types of runs experienced when using HBase are shown in Figure 10 and 11, (a) when HBase decides to split during the run, or (b) when HBase does not do any splits during the run. Figure 11(b) confirms that splits and compactions are not happening in this run and the read response times are constant and low (20-30 ms after all data has been loaded). In fact, all the data is attached to the same tablet and served by one tablet server. Splits induce a lot of work and interference with the measurement workload, so the Y-axes of Figures 10(a) and on the right in Figure 11(a) change by more than an order of magnitude. The read response time immediately after all data has been inserted peaks at 5,000 ms and does not drop below 100 ms until about 12 minutes after the insertion is complete. One might conclude that the insertion takes at least 12 minutes longer than just building and inserting the store files, or close to 25 minutes, almost as long the fastest pre-split experiment's insertion and post-insertion compactions took (in Section 3.3).

When this test is run on the ACCUMULO table store, splitting and compacting is more aggressive and consistent. The interference to read response time is larger in the first measurement phase, but by 3 minutes into the second measurement phase, the splitting and compaction is done, allowing the entire load to be complete in less than one-third of the time of HBase run experiencing splits. This experiment emphasizes the importance of the policies managing splits and compactions to performance.

3.5 Server-side filtering

By default, read or scan operations in YCSB return all the columns associated with the respective row(s). For a large table with thousands of columns, clients may get much more data (than what they are interested in) resulting in high data processing and network transfer overheads. Filtering at the tablet servers helps minimize this overhead by not returning irrelevant data to the client.

To understand the effectiveness of server-side filtering, we use a YCSB++ test to create a data-set that has 100 times more data per row than the data-set used in prior tests: each row has 10 times as many cells (total of 100 cells) and each cell is 10 times larger in size (total of 1 KB). Moreover, each column has a dedicated locality group. This test's workload issues scan requests for one cell from each of 1, 10, 100, or 1000 rows at a randomly selected row key (this is the third type of filter described in Section 2.3.5 that returns the row's key and the {column name, cell value} tuple). We refer to these values as the "scan length" and report the client-perceived scan throughput in terms of number of rows received by the client every second. All results in this section are computed as an average of three runs and have very small variance.

Figure 12 shows that server-side filtering in ACCUMULO drastically improves client throughput only for a scan length of 1,000 rows. In fact, for all smaller scan-lengths, filtering performs much worse than without server-side filtering – a phenomenon that arises from ACCUMULO's scan implemen-

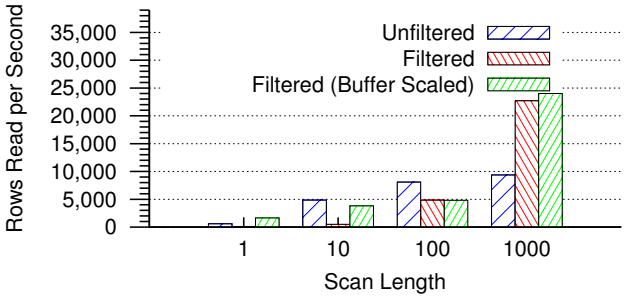

Figure 12: ***Performance of server-side filtering in*** ACCUMULO ***for varying scan lengths in terms of rows.***

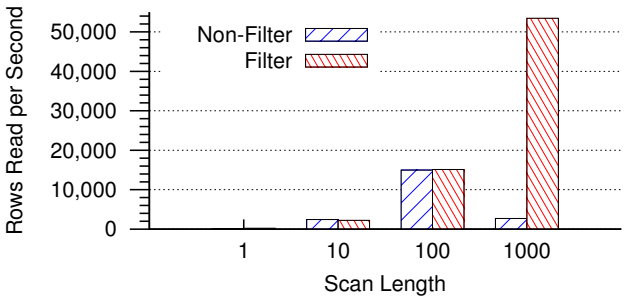

Figure 13: ***Performance of server-side filtering in HBase for varying scan lengths.***

tation. ACCUMULO uses a `scanner` object to return results of a scan operation. A `scanner` object, by default, can hold 1,000 rows. An ACCUMULO tablet server returns a `scanner` object to the client only when the object is filled (1,000 rows). Consequently, even if a scan request wants only a single row, which is the case for scan length 1 in Figure 12, the tablet servers will continue to scan table data until the `scanner` object is filled. As a result, server-side filtering exacerbates load on the servers, especially for scan lengths of 1 and 10 rows, because the server has to read and filter more rows to fill the single `scanner` object that was requested.

Instead of using `scanner` objects with the default size, we modified ACCUMULO's DB client API in YCSB++ to allow our test to size `scanner` objects to the expected scan length. This modification, titled in Figure 12 as "filtered (buffer scaled)", decreases the unneeded scanning load on the servers and results in a significant improvement for smaller scan lengths of 1 and 10 rows.

We repeat this experiment to study server-side filtering in HBase. Figure 13 shows that HBase, similar to ACCUMULO, does not benefit from server-side filtering when scan lengths are smaller than 100 rows, but filtering improves the throughput by 10 times for scan length of 1,000 rows. We also observe that HBase does not require batch size manipulation because it performs less aggressive prefetching than ACCUMULO.

3.6 Benchmarking access control

Because only ACCUMULO supports fine grained access control, we could not perform a comparison with HBase.[3] However, YCSB++ enables testing of the costs associated with fine-grained access control in ACCUMULO.

Attributes of each cell	Attribute size
Row Key	12 bytes
Column family	3 bytes
Column	6 bytes
ACL	100 bytes
Value	2 bytes
Timestamp	8 bytes

Table 4: ***Sizes of attributes associated with each cell used for ACL benchmarking.***

Although fine-grained mechanisms like cell-level access control provide great flexibility for data security, they come at the cost of additional performance overhead: the first overhead stems from higher disk and network traffic for each access and the second overhead stems from computationally verifying credentials on each access. Both of these overheads are dependent on the size of the ACLs in terms of number of users and groups.

Although ACCUMULO contains optimizations for ACLs that are frequently reused, we setup experiments to benchmark the worst-case performance by using a unique ACL for each key and by making the size of the ACL three times larger than the rest of the cell itself, as shown in Table 4. For this experiment, we use two benchmarks – an insert workload that writes 48 million single-cell rows in an empty table and a scan workload that scans 320 million rows. Two different client configurations – one with a single client with 100 threads and other with six clients with 16 threads each – generate load on a 6-node ACCUMULO cluster. We report an average of three runs (and standard deviation) for each configuration.

Figure 14 shows the insert throughput, measured as the number of rows inserted per second, for different numbers of entries in each ACL (while the total size of the ACLs is constant). A value of zero entries means that no security was used. When the workload uses a single client with 100 threads, we observe that the throughput decreases with increasing number of entries in each ACL: in comparison to not using any access control, throughput drops by 24% with 4 entries in the ACL and by as much as 47% with an 11-entry ACL. This happens because the single YCSB++ client is running at almost 100% CPU utilization (as shown in Figure 15) and increasing the number of entries in each ACL leads to increased computation overhead. However, using six YCSB++ clients with 16 threads each, reduces the insert throughput only by about 10%, even when there are 11 entries in the ACL.

Figure 16 shows the scan throughput, measured as the number of rows scanned per second, for varying number of entries in each ACL. Unlike the insert throughput, we observe that the scan throughput is not affected by using different client configurations. However, in both cases, the scan throughput drops by about 45% once fine-grained ACLs are invoked and remains same for different number of entries in an ACL. In Figure 17, Otus performance monitoring shows that this degradation results from a four-fold increase in the amount on data sent from the ACCUMULO tablet servers to

[3]HBase plans to add security in future releases [30]

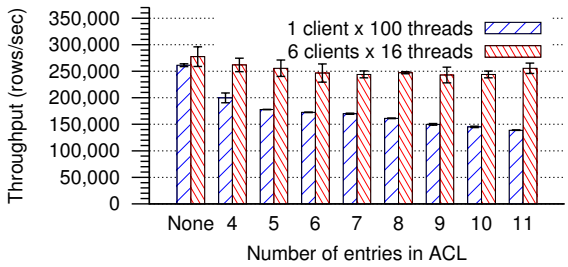

Figure 14: *Insert throughput decreases with increasing number of ACL clauses when the CPU is a limiting resource.*

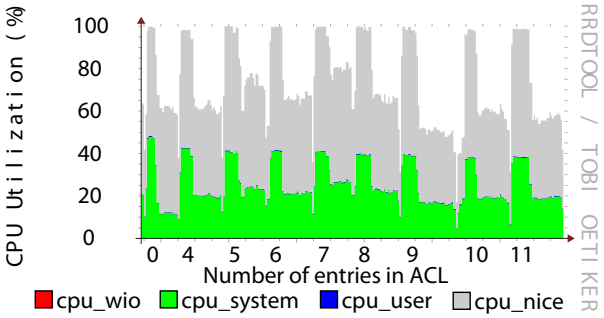

Figure 15: *Single* **YCSB++** *client inserting records is CPU limited resulting in lower throughput as the numbers of entries in each access control list increases.*

the clients, even though the clients do not always have use for the ACLs after they are allowed data access.[4] In these experiments, we see that extremely complex fine-grained access controls can impact performance significantly but "only" by a factor of two.

4. RELATED WORK

To the best of our knowledge, this is the first work to propose systematic benchmarking techniques for advanced functionality in table stores. All advanced features of table stores discussed in this paper are inspired by decades of research and implementation in traditional databases. We focus this section on work in scalable and distributed table stores.

Weak consistency: Various studies have measured the performance impact of weaker consistency semantics used by different table stores and service providers [34, 51]. Using a first read-after-write measurement similar to YCSB++, one study has found that 40% of reads return inconsistent results when issued right after a write [34]. Other studies found that Amazon SimpleDB's eventually consistent model may cause users to experience stale reads and inter-item inconsistencies [36, 51]. Unlike our approach which measures the time to the first successful read, the SimpleDB study also checked if subsequent reads returned stale values [51]. In contrast to SimpleDB's eventual consistency which stems

[4]Some users of ACCUMULO use ACLs to convey interesting information to the reader.

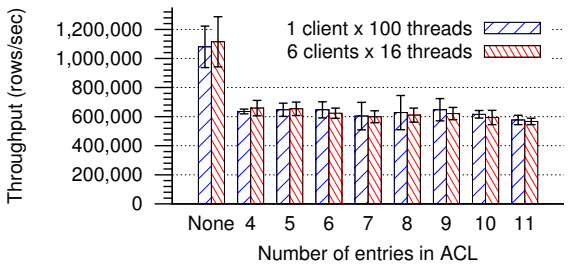

Figure 16: *For both client configurations, scan performance drops by about 45% when* **Accumulo** *enables per-cell access control and the throughput is not affected by the number of entries in the ACL.*

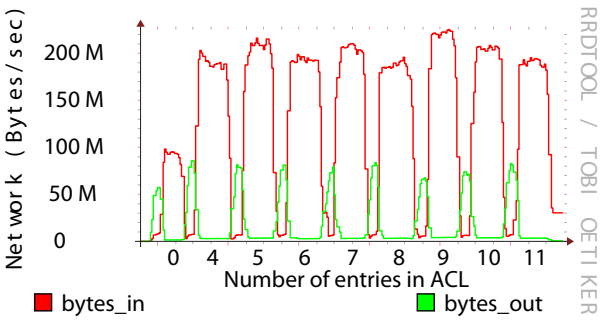

Figure 17: *Significantly more bytes are sent from* **Accumulo** *servers to* **YCSB++** *clients while scanning records with access control lists.*

from divergent replicas, both HBase and ACCUMULO experience weak consistency only when batch writing is enabled at the clients. Orthogonal approaches to understand weak consistency include theoretical models [31] and algorithmic properties [4, 19].

Ingest-intensive optimizations: The use of an external Hadoop job to format and store massive data-sets in a native tabular form understood by the table store servers has been exploited for HBase [5, 28]. An alternate approach adopted by the PNUTS system is to use an optimization-based planning phase before inserting the data [47]. This phase allows the system to gather statistics about the data-set that may lead to efficient splitting and balancing. Such an approach could be used to complement the Hadoop-based bulk load tool used in YCSB++.

Server-side filtering: Function shipping in databases has long been used in parallel database [18] and this idea has appeared in active disks (in a single-node setting) [42], MapReduce (in cloud computing) [16] and key-value stores (in wide-area networks) [21]. Because the Hadoop/MapReduce framework is built on the premise of collocating compute and data, both HBase and BigTable have proposed the use of co-processors to allow application level code to run on the tablet servers [8, 15, 32, 40]. The YCSB++ approach to testing server-side filtering focuses on regular expression based filters rather than the general abstractions proposed by HBase [32, 40].

	Voldemort Key-Value Store [3]	Apache Cassandra [1]	MongoDB [2]
Weak consistency	Eventual consistency semantics resulting from divergent replicas	Eventual consistency semantics resulting from divergent replicas	Default mode: strong data consistency (can support weak consistency for high performance)
Bulk loading	No support for bulk loading	Provides an interface for bypassing the RPC marshalling process and directly bulk loading into Cassandra's memtable format [9]	Custom bulk-load commands to import data from different file formats; no API support to directly create MongoDB data files
Table pre-splitting	Uses consistent hashing-based data partitioning that randomly pre-splits hash ranges across all server nodes (in an *a priori* manner); users cannot control this partitioning	Allows an pre-splitting ranges using different partitioner modes (using both the key and the hash of the key); but to avoid hot-spots, the node tokens may need to be constantly adjusted	Allows pre-splitting based on a continuous key-range, followed by load-balancing the pre-split ranges
Server-side filtering	Simple key-value data model with no filtering support	Filtering based on column names and key values; no support for user-defined matching for value-based filtering	Enables SQL-style "`where`" clause filtering on the servers
Fine-grained ACLs	No access control support	Extensible authorization to control read/writes to a column family (no cell-level ACLs [10])	No access control support (only simple user authentication for the DB)

Table 5: ***Applicability of* YCSB++ *extensions for other scalable stores supported by the original YCSB distribution [14].***

5. CONCLUSION AND FUTURE WORK

Scalable table stores started with simple data models, lightweight semantics and limited functionality. Today, they feature a variety of performance optimizations, such as batch write-behind, tablet pre-split, bulk loading, and server-side filtering, as well as enhanced functionality, such as per-cell access control. Coupled with complex deferring and asynchronous online re-balancing policies, these optimizations have performance implications that are neither assured nor simple to understand, and yet are important to the goals of high ingest rate, secure scalable table stores.

Benchmarking tools like YCSB [14] help with basic, single-phase workload testing of the core create-read-update-delete interfaces, but lack support for benchmarking and performance debugging advanced features. In this work, we extended YCSB's modular framework to integrate support for advanced feature testing. Our tool, called YCSB++, is a distributed multi-phase YCSB with an extended abstract table API for pre-splitting, bulk loading, server side filtering, and applying cell-level access control lists.

For more effective performance debugging, YCSB++ exposes its internal statistics to an external monitor, like Otus, where they are correlated with statistics from the table store under test and system services like file systems and MapReduce job control. Collectively comparing metrics of internal behaviors of the table store (such as compactions), the benchmark phases, and the network and CPU usage of each service, yields a powerful tool for understanding and improving scalable table store systems.

Although we evaluated YCSB++ with only two table stores, HBase and Accumulo, which have multi-dimensional distributed sorted map structures, we believe that YCSB++ can be used to test other table stores as well. Table 5 shows the applicability of different YCSB++ extensions to three other table stores, Voldemort, Cassandra and MongoDB, that were originally supported by YCSB.

Testing weak consistency is one feature that is important for all three table stores. Although weak consistency in HBase and Accumulo stems from client-side buffering, we were able to adapt YCSB++ to measure the read-after-write time lag for Voldemort and Cassandra where eventual, weak consistency can result from divergent replicas. Some features like table pre-splitting and fine-grained access control are not common; for example, pre-splitting is relevant only if the underlying distributed index supports incremental growth.

YCSB++ is publicly available under the Apache License at `http://www.pdl.cmu.edu/ycsb++/` for researchers and developers to use it to benchmark their table store deployments and extend it to facilitate testing of other advanced features.

Acknowledgements

The work in this paper is based on research supported in part by the National Science Foundation under award CCF-1019104, by the Betty and Gordon Moore Foundation, by the Qatar National Research Fund under award number NPRP 09-1116-1-172, and by grants from Google and Yahoo!. We also thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware) for their interest, insight, feedback, and support.

References

[1] Apache Cassandra. http://cassandra.apache.org/.

[2] MongoDB. http://www.mongodb.org/.

[3] Project Voldemort: A distributed database. http://project-voldemort.com/.

[4] A. S. Aiyer, E. Anderson, X. Li, M. A. Shah, and J. J. Wylie. Consistability: Describing usually consistent systems. In *Proc. of the 4th Workshop on Hot Topics in Syetms Dependability (HotDep '2008)*, San Diego, CA, December 2008.

[5] A. Barbuzzi, P. Michiardi, E. Biersack, and G. Boggia. Parallel bulk Insertion for large-scale analytics applications. In *Proc. of the 4th ACM SIGOPS/SIGACT International Workshop on Large Scale Distributed Systems and Middleware (LADIS '2010)*, Zurich, Switzerland, July 2010.

[6] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/r0.16.4/hdfsdesign.html.

[7] E. A. Brewer. Towards robust distributed systems. Keynote at the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC '2000) on July 19, 2000 in Portland OR.

[8] M. Cafarella, E. Chang, A. Fikes, A. Halevy, W. Hsieh, A. Lerner, J. Madhavan, and S. Muthukrishnan. Data Management Projects at Google. *SIGMOD Record*, 37(1), 2008.

[9] Cassandra. Cassandra's Binary Memtable. http://wiki.apache.org/cassandra/BinaryMemtable.

[10] Cassandra. Cassandra's Extensible Authentication/Authorization. http://wiki.apache.org/cassandra/ExtensibleAuth.

[11] R. Cattell. Scalable SQL and NoSQL Data Stores. http://www.cattell.net/datastores/Datastores.pdf.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '2006)*, Seattle, WA, November 2006.

[13] Collectd: The system statistics collection daemon. http://collectd.org/.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM Symposium on Cloud Computing (SOCC '2010)*, Indianapolis, IN, June 2010.

[15] J. Dean. Designs, Lessons and Advice from Building Large Distributed Systems. Keynote at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS '2009) on October 11, 2009 - http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '2004)*, San Francisco, CA, December 2004.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP '2007)*, Stevenson, WA, October 2007.

[18] D. J. Dewitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6), 1992.

[19] A. Fekete and K. Ramamritham. Consistency Models for Replicated Data. In *Replication*, volume 5959 of *Lecture Notes in Computer Science*, 2010.

[20] A. Fikes. Storage Architecture and Challenges. Talk at the Google Faculty Summit 2010 on July 29, 2010.

[21] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An Active Distributed Key-Value Store. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '2010)*, Vancouver, Canada, October 2010.

[22] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

[23] G. Graefe. Partitioned B-trees: A user's guide. In *Proc. of the 10th Conference on Database Systems for Business, Technology and Web (BTW '2003)*, Leipzig, Germany, February 2003.

[24] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1), 2006.

[25] G. Graefe and H. Kuno. Fast Loads and Queries. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems II*, volume 6380 of *Lecture Notes in Computer Science*, 2010.

[26] Hadoop. Apache Hadoop. http://hadoop.apache.org/.

[27] HBase. Apache HBase. http://hbase.apache.org/.

[28] HBase. HBase - Bulk Loads in HBase. http://hbase.apache.org/docs/r0.89.20100621/bulk-loads.html.

[29] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of the 2010 USENIX Annual Technical Conference (USENIX ATC '2010)*, Boston, MA, June 2010.

[30] E. Kootz. The HBase Blog – Secure HBase: Access Controls. http://hbaseblog.com/2010/10/11/secure-hbase-access-controls/.

[31] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *Proc. of the VLDB Endowment*, 2(1), 2009.

[32] M. Lai. HBase Coprocessors. http://hbaseblog.com/2010/11/30/hbase-coprocessors/.

[33] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. In *Proc. of the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS '2009)*, Big Sky, MT, October 2009.

[34] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proc. of the 9th ACM SIGCOMM Conference on Internet Measurement (IMC '2009)*, Chicago, IL, November 2009.

[35] Lily. Bulk Imports in Lily. http://docs.outerthought.org/lily-docs-current/438-lily.html.

[36] H. Liu. The cost of eventual consistency. http://huanliu.wordpress.com/2010/03/03/the-cost-of-eventual-consistency/.

[37] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation And Experience. *Parallel Computing*, 30(7), 2004.

[38] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), 1996.

[39] G. Pohl and M. Renner. *Munin: Graphisches Netzwerk-und System-Monitoring*. Open Source Press, 2008.

[40] A. Purtell. Coprocessors: Support small query language as filter on server side. https://issues.apache.org/jira/browse/HBASE-1002.

[41] K. Ren, J. López, and G. Gibson. Otus: Resource Attribution in Data-Intensive Clusters. In *Proc. of the 2nd International Workshop on MapReduce and its Applications (MapReduce '2011)*, San Jose, CA, June 2011.

[42] E. Riedel, C. Faloutsos, G. Gibson, and D. Nagle. Active Disks for Large-Scale Data Processing. *IEEE Computer*, 34(6), 2001.

[43] G. Robidoux. Minimally Logging Bulk Load Inserts into SQL Server. http://www.mssqltips.com/tip.asp?tip=1185.

[44] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1), August 1992.

[45] SciDB. Use Cases for SciDB. http://www.scidb.org/use/.

[46] M. Seltzer. Beyond Relational Databases. *Communications of the ACM*, 51(7), 2008.

[47] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient Bulk Insertions into a Distributed Ordered Table. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '2008)*, Vancouver, BC, Canada, June 2008.

[48] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, , and S. Zdonik. C-Store: A Column Oriented DBMS. In *VLDB*, 2005.

[49] TokuTek. Fractal Tree Indexing in TokuDB. http://tokutek.com/technology/.

[50] W. Vogels. Eventually Consistent. *ACM Queue*, 6(6), 2008.

[51] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective. In *Proc. of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '2011)*, Asilomar, CA, January 2011.

[52] ZooKeeper. Apache ZooKeeper. http://zookeeper.apache.org/.