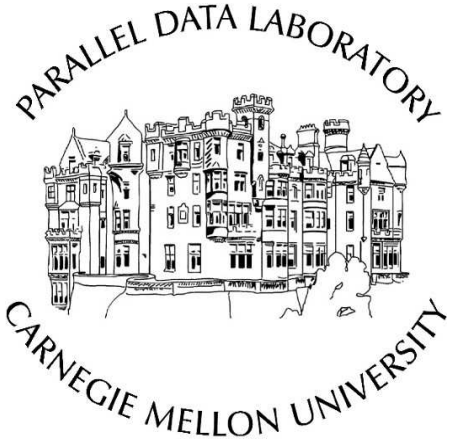

YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores

Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López and Garth Gibson
Carnegie Mellon University

CMU-PDL-11-111
May 2011

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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 Los Alamos National Laboratory under contract number 54515-001-07, 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, LSI, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware) for their interest, insights, feedback, and support.

Keywords: Scalable cloud databases, benchmarking, performance debugging, HBase, IcyTable, YCSB, weak consistency, bulk inserts, pre-splits, server-side filtering, fine-grained access control

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 a set of extensions, called YCSB++*, 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 deferring 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 DoD IcyTable, developed to emphasize high ingest rates and fine-grained security.*

1 Introduction

Large-scale table stores, such as BigTable [10], Dynamo [15], HBase [3] and Cassandra [4, 31], are becoming increasingly important for Internet services. To meet the desired scalability and availability requirements, these table stores were designed to be more simple and lightweight than traditional relational databases [38, 41]. Today, these table stores are used both by application services and by critical systems infrastructure. Applications ranging from business analytics to scientific data analysis rely on table stores [9, 40] and the next version of GoogleFS, called Colossus, stores all file system metadata in BigTable [18].

This growing adoption, coupled with spiraling scalability and tightening performance requirements, has motivated 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 are being extended with new mechanisms for consistency, concurrency, data partitioning, indexing, and query analysis.

Features to speed up ingest-intensive workloads are required for many applications that generate and store petabytes of data in a table store at very high speeds [40]. Typically data is ingested in a table using iterative insertions or bulk insertions. Iterative insertions add new data through "insert" or "update" operations and they are optimized using numerous techniques that include buffering at the clients, disabling logs [1, 35], relying on fast storage devices [43], and using indexing structures optimized for high-speed inserts [21–23, 36]. Bulk insertions load existing data-sets by converting them from their current storage format to the format of the respective table store. Proposals to speed-up bulk insertion in scalable table stores include using optimization frameworks to pre-split partitions [42] and running Hadoop jobs to parallelize data loading [2, 7].

Another new feature in table stores is the ability to run distributed computations directly on table store servers instead of clients. Google's BigTable coprocessors allow arbitrary application code to run directly on tablet servers even when the table is growing and expanding over multiple servers [9, 13]. HBase plans to use a similar technique for server-side filtering and fine-grained access control [25, 26, 29]. This server-side execution model, inspired from early work in parallel databases [16], is designed to drastically reduce the amount of data shipped to the client and significantly improve performance, particularly of scan operations with an application-defined filter.

With the rising importance and profusion of table stores, it is natural for a benchmarking framework to be developed. Yahoo! Cloud Serving Benchmark (YCSB) is a great framework for measuring the basic performance of several popular table stores including HBase, Voldemort, Cassandra and MongoDB [12]. 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 random or sequential row insertions, lookups or deletions, YCSB lacks support for benchmarking advanced functionality perceived to be important and increasingly supported by table stores.

Our goal is to extend the scope of table store benchmarking to more complex features and optimizations than are supported by YCSB. We seek a systematic approach to benchmark advanced functionality in a scalable and distributed manner. The design and implementation of our techniques require no changes to the table stores being evaluated; however, the abstraction layer adapting a table store to a benchmarking crafted API for specific advanced functions may be simple or complex, depending on the capabilities of the underlying table store.

Table 1 summarizes the contributions of our paper. The first contribution is a set of benchmarking techniques to measure and understand five commonly found 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++, in

Extensions to the YCSB framework	Observations in HBase and IcyTable
Ingest-intensive workload extensions	
External Hadoop tool that loads all data in a table store file in a format used by the table store servers	Bulk insertion delivers highest data ingest rate of all ingestion techniques, but the servers may end up re-balancing the data regions
Pre-splits supported by a new workload executor for key range partitioning and API extensions to send keys to appropriate partitions	Ingest throughput of HBase increases by 20% but if range partitioning is not known a priory the HBase servers may incur re-balancing and merging overhead
Offloading functions to the DB servers	
New workload executor that generates "deterministic" data to allow setting appropriate filters and DB client API extensions to send filters to servers	Server-side filtering benefits HBase and IcyTable only when the client scans a enough data (more than 10 MB) to mask the overhead of network transfer and disk reads
Fine grained access control (Detailed in Appendix A)	
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 [3] but IcyTable does)	IcyTable'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 wit the data)
Distributed testing using multiple YCSB client nodes	
ZooKeeper-based barrier synchronization for multiple YCSB clients to coordinate the start and end of different tests	Distributed testing benefits multi-client, multi-phase testing (used to evaluate weak consistency and table pre-splits)
Distributed event notification using ZooKeeper to understand the cost (measured as read-after-write latency) weak consistency	Both HBase and IcyTable 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

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 IcyTable case studies.

the YCSB framework. Our final contribution is the experience of analyzing these features in two table stores derived from Google BigTable, HBASE [3] and IcyTable.

2 YCSB++ Design

This section presents an overview of HBase, IcyTable and YCSB, followed by the design and implementation of advanced functionality benchmarking techniques in the current YCSB framework.

2.1 Overview of HBase and IcyTable

HBase and IcyTable are BigTable-like scalable semi-structured table stores that store data in a multi-dimensional sorted map where keys are tuples of the form {row, column, timestamp}. HBase is developed as a part of the open-source Apache Hadoop project [3, 24] and IcyTable is developed for the U.S. Department of Defense. Both are written in Java and layered on top of the Hadoop distributed file System (HDFS) [27]. They support efficient storage and retrieval of structured data, including range queries, and allow using IcyTable tables as input and output for MapReduce jobs. Other features in these systems include automatic load-balancing and partitioning, data compression and server-side user-defined function such as regular expression filtering. In addition, IcyTable also supports fine-grained security labels for tables, columns and individual cells. To avoid terminology differences in HBase and IcyTable, the rest of the paper will use the terminology from the Google BigTable paper [10].

At a high-level, each table is indexed as a B-tree where all records are stored on the leaf nodes called *tablets*. An installation of these table stores consists of *tablet servers* running on all nodes in the cluster. Each tablet server is responsible for handling requests for several tablets. A tablet consists of rows in a contiguous range in the key space and is stored as one or more files stored in HDFS. Different table stores represent these files in their custom formats (BigTable uses an SSTable format, HBase uses an HFile format and IcyTable uses a RegionFile format); we will refer to them as *store files*. Both HBase and IcyTable provide columnar abstractions that allow users to group a set of columns in a *locality group*. By storing a locality group in its separate store file in HDFS, these systems can perform scans efficiently without reading excess data (from other other columns). Both 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 managed by one tablet server. To access a table, clients contact the master to get the location of the tablet servers. All inserts are sent to this server and buffered in memory; this in-memory buffer corresponding to each tablet is called a *memstore*. When this memstore fills up, the tablet server flushes it 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. All store files are a part of the same tablet managed by a tablet server. Once a tablet exceeds its threshold size, the tablet server splits the overflow 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 keep a bounded number of store files, a periodic *major compaction* operation combines merges all the store files in a region into one new store file. All files are stored in HDFS and these table stores rely on HDFS for durability and availability of data.

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 [12]. At a high level, YCSB consists of the following components, shown as the light boxes in Figure 1.

The **workload executor** module is responsible for loading the test data and generating operations that will be issued by DB Clients to the table stores. The default workload provided by YCSB issues a series of basic operations including reads, updates, deletes and scans. In YCSB, a "read" may be reading a single row or scanning a series of consecutive rows and an "update" may either insert a new row or update an existing one. Operations are issued one at a time per thread and their distributions are based on proportions specified in the **workload parameter file** for a benchmark

YCSB allows users to specify the type of benchmark using a set of **workload parameter files**. Each parameter file includes properties that identify the types of operations to perform and their distributions. YCSB distribution includes five default workload files (called Workloads A, B, C, D, E) that generate read-intensive, update-intensive and scan-intensive workloads (or a mix of these).

DB Clients are wrapper classes around the APIs of various table stores. Currently YCSB supports HBase, Cassandra, MongoDB and Voldemort; in this work, we added a new client for the IcyTable table store. For a given table store, its DB client converts a 'generic' operation issued by the workload executor to an operation specific for the table store under test. For example, for an HBase cluster, if the workload executor generates a `read()` operation, the HBase DB client issues a `get()` operation to the HBase servers.

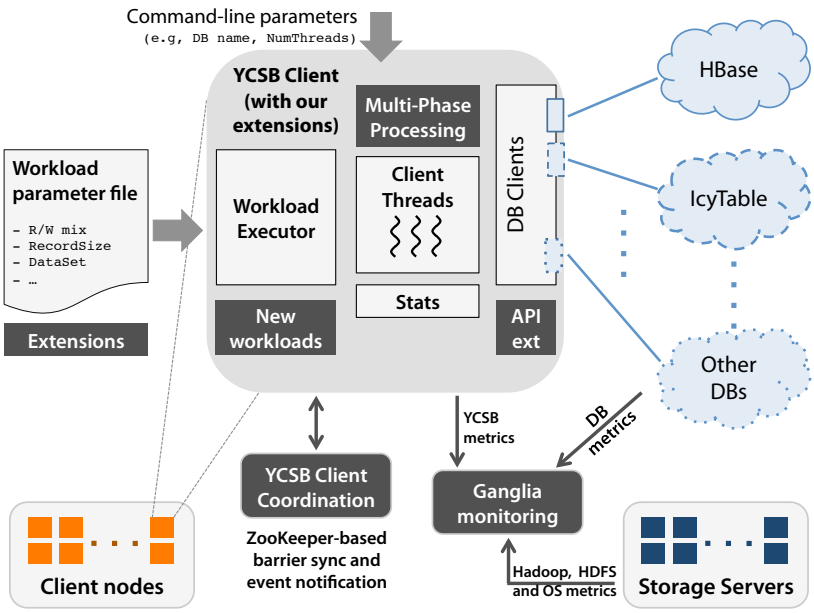

Figure 1. YCSB++ functionality testing framework. Unshaded boxes show modules in YCSB v0.1.3 [12] and dark gray boxes show our extensions.

To start executing a benchmark, YCSB uses the **client threads** module to start multiple threads that all call the workload executor to issue operations and then report the measured performance to the **stats** module. Users specify the number of work generating threads, the table store being evaluated and the workload parameter file as command line parameters.

2.3 New extensions in YCSB++

YCSB's excellent modular structure made it natural for us to integrate testing of additional functionality as YCSB extensions. In this section, we show how YCSB++ implements functionality extensions in different YCSB modules, denoted by dark shaded boxes in Figure 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 originally designed to run on a single node and this one instance of YCSB, even with hundreds of threads, may be insufficient to effectively test large deployments of cloud table stores. YCSB++ controls execution of different workload generator instances through distributed coordination and event notification using Apache ZooKeeper. ZooKeeper is a service that provides distributed synchronization and group membership services, and is widely used by open-source cloud table stores, including HBase and IcyTable [5, 28]

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: status flag, ZooKeeper server address, barrier-sync variable, and 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++, it creates a "barrier" directory in its namespace. Whenever a new YCSB++ client starts, it joins the barrier by contacting ZooKeeper server that in turn creates a new entry, corresponding to this client's identifier, in

the "barrier" directory. Number of entries in a "barrier" directory indicates the number of clients that have joined the barrier. If all clients have joined, ZooKeeper sends a callback to these clients to begin executing their workloads; if not, YCSB++ clients block and wait for more clients to join. Upon finishing the test (or a phase of a test), YCSB++ clients notify the ZooKeeper about leaving the barrier.

Weak consistency – Table stores provide high throughput and high availability by eliminating expensive features, such as ACID properties, of traditional relational databases. Based on the **CAP** theorem, many table stores tolerate network **P**artitions and provide high **A**vailability by giving up on strong **C**onsistency guarantees [8, 20]. Most systems offer "loose" or "weak" consistency semantics, such as eventual consistency [15, 45], 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., actions taken on receiving stale data). YCSB++ measures the time lag before a different client can successfully observe a value that was most recently written by other clients.

To evaluate the time to consistency, YCSB++ uses the aforementioned `ZKCoordination` module's producer-consumer abstraction for asynchronous directed coordination between multiple clients. YCSB++ clients interested in benchmarking weak consistency specify three properties in the workload parameter file: status flag to check if a client is a producer or a consumer, ZooKeeper server address, and a reference to the shared queue data-structure in ZooKeeper. Synchronized access to this queue is provided by ZooKeeper: like the barrier abstraction, 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 are inserting or updating 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 and, on receiving a notification from ZooKeeper about new elements, remove a key from the queue to read it from the table store. If the read fails, "producers" 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 set of keys for read-after-write measurements. The "read-after-write" time lag for key K is the difference from the time a "producer" inserts K until the first time a "consumer" can successfully read that key from the table store server; we only report the lag for keys that needed more than one read attempt.

Table pre-splitting for fast ingest – Recall that both HBase and IcyTable 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 the tablet during splits and, until the split finishes, refuses any operation (including a read) addressed to this tablet. One way to avoid this splitting overhead is to pre-split a table into multiple key ranges based on a priori knowledge of the workload.

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

Bulk loading using Hadoop – To add massive data-sets, table stores rely on high throughput, specialized bulk loading tools [2]. YCSB++ supports bulk loading using an interface that is different from YCSB's API for normal insert operations. We built an external tool that directly processes the incoming data, stores it in an on-disk format specific to the table store and notifies the servers about the existence of this new data-set (files). YCSB++ uses a Hadoop application for the processing and storing phase of bulk load and a custom `import()` API call for notifying

the table store about the data files. Table store servers make this data-set available to users after successfully updating internal data-structures used to reference these new data files.

To facilitate better understanding of bulk load operations, our Hadoop tool runs a multi-phase task that generates the data, sorts it, partitions it and finally stores it in appropriate table store file formats. The generated data can also be used for other YCSB benchmarks.

Server-side filtering – Current YCSB distribution contains basic support for simple filtering that allows users to create tests that request reads or scans from only a single column. To evaluate the effects of filtering, YCSB++ provides support to control the length of a scan and to manipulate locality groups. Having precise control over scan length makes it easier to reason about the amount of data that must be processed for each request for the performance gains of filtering to outweigh its costs.

We added a new property that signals the DB clients to store each column in a separate locality group (if the table store supports this feature).[1] YCSB++ includes four advanced server-side filters that are currently not supported in YCSB: filters that return keys that match a specified pattern, filters that return columns that match a specified pattern, filters that return columns whose value matches a specified pattern, and filters that return entire rows for keys where a specified column has a value that matches a specified pattern.

Filters are specified in the workload parameter files, detected and configured by the workload generator modules, and implemented by the DB clients for HBase and IcyTable. YCSB++ also enhanced the DB client API to allow implementations to optionally support these filters. By default, both HBase and IcyTable, use regular expressions to specify patterns, but our extensions allow users to other alternate schemes supported by a table store.

2.4 Performance monitoring in YCSB++

There are many tools for cluster-wide monitoring and visualization such as Ganglia [34], Collectd [11], and Munin [44]. These tools are very powerful for large scale data gathering, transport, and visualization. While such tools make it easy to view application-agnostics metrics such as aggregate CPU load in a cluster, to investigate the performance of cluster-wide table stores servers we require application-specific performance monitoring and analysis. Rather than simply extracting virtual memory statistics for the sum of all processes running on a cluster, what we really want is the aggregate memory usage of a MapReduce task, or a history of HBase's compaction operations.

Using Ganglia as a base, we built a custom monitoring tool, named Otus [37], for YCSB++. On each cluster host, Otus runs a daemon process that periodically collects metrics from different table store components including tablet servers, master node, HDFS and data nodes. Users can aggregate and analyze the collected data through a tailored web-based visualization system. Otus's metrics are stored in a central repository that users can access for post-processing and analysis.

For the purpose of YCSB++ benchmarks, Otus collects metrics from the operating system, the table store and the YCSB++ clients. The OS-level resource utilization for individual table store 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. Table store-specific metrics, such as the number of tablets and store files, provide information about the inner workings of these systems. Otus currently supports metrics from HBase and IcyTable. Extending support for another table store involves writing a Python script to extract its metrics. In addition, we extended the YCSB client to periodically send the performance metrics to Otus. Through the web interface, a user can query and display metrics for the table store components pertaining to a benchmarking experiment.

[1] Ideally, we would like fine-grained control over locality groups and support for sparse inserts that do not specify values for all columns.

By storing the collected data in a central repository and providing a flexible web interface to access the data, users can rely on fine-grained time series of different metrics coming from different layers in the system and establish relations between the way various components behave. We used Otus to observe and analyze the behavior of table stores while our benchmarking tests executed. Figure 2 is an example of output from Otus showing a combined simultaneous display of three metrics collected during an experiment: HDFS data node traffic, tablet server CPU utilization and the number of store files in the system.

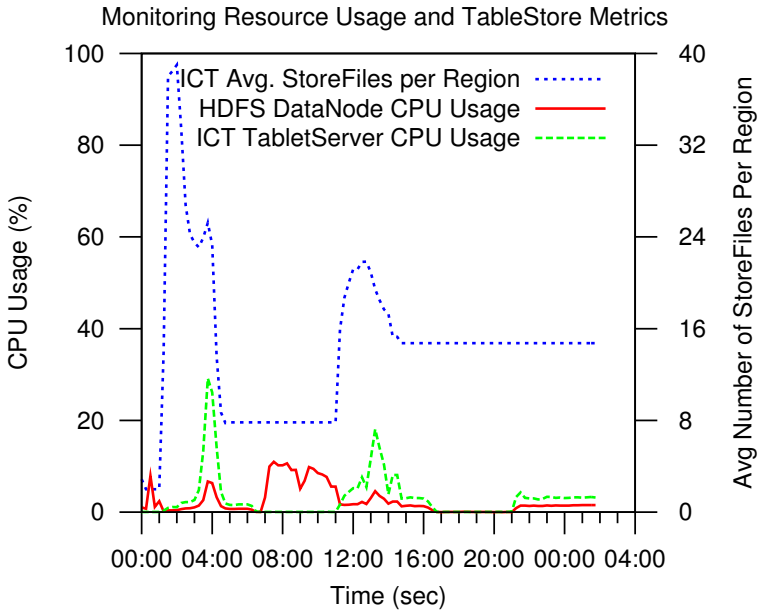

Figure 2. Combined display of metrics collected with Otus.

3 Analysis

All our experiments are performed on the 64-node "OpenCloud" cluster at CMU. Each node has a 2.8 GHz dual quad core CPU, 16 GB RAM, 10 Gbps Ethernet NICs, 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 Arista 7124S 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 IcyTable were running on this HDFS configuration with one master and 6 region servers – a configuration similar to the original YCSB [12].

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

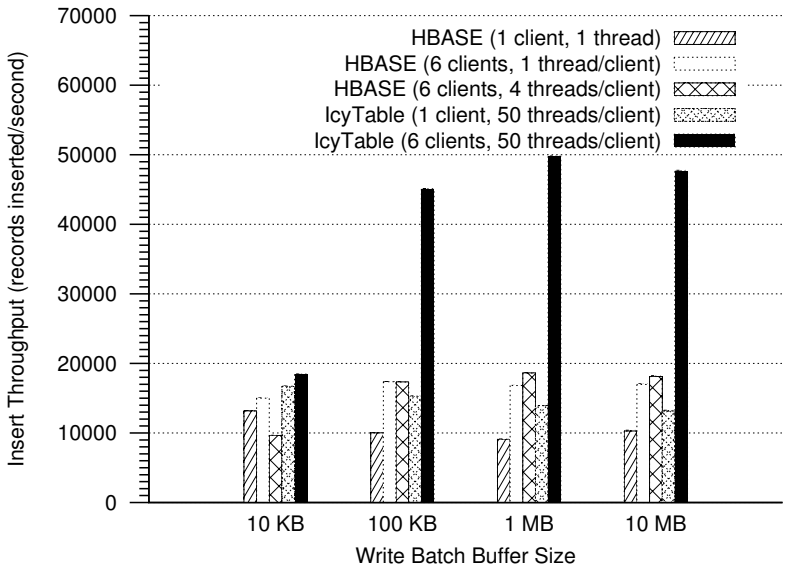

Figure 3. Effect of batch size on insert rate.

3.1 Effect of batch writing

Both HBase and IcyTable coalesce application writes in a client-side buffer before sending them to a server. Batch writing helps improve the throughput by avoiding a round-trip latency for each write. To measure the effect of batch size, we configured both 6-node HBase and IcyTable clusters layered on a HDFS cluster (with two other nodes acting as master). We used 6 YCSB++ clients, on separate machines, that each inserts 9 million rows in a single table using 50 threads for IcyTable and, at most, 4 threads for HBase.[2]

Figure 3 reports the insert throughput, i.e., number of rows inserted per second, for different sizes of the batch-writing buffer. To measure the baseline performance, we configured YCSB++ to run 50 threads with one only client when testing IcyTable, and at this configuration, IcyTable does not observe any performance gain from using larger batch sizes. Using our Otus monitors, we discovered the insert rate is limited because the YCSB++ client is running at almost 100% CPU utilization and cannot generate enough work to keep a 6-node IcyTable cluster busy; Figure 4 shows this for an experiment when the client batch size was 100 KB (we observed the same phenomenon for all other batch sizes).

In our next experiment on IcyTable, we used six YCSB++ clients, each with 6 threads. Figure 3 shows that increasing the batch size from 10 KB to 100 KB doubles the throughput. However, insert throughput does not improve, and roughly stays the same, for higher batch sizes. Our monitoring information for this experiment, shown in Figure 5, shows that for batch sizes of 100KB or more all six IcyTable servers are close to saturation, operating at more than 80% CPU utilization. (Note that this graph shows the average CPU utilization on a six node cluster.)

Figure 3 also shows HBase performance with a single YCSB++ client with one thread, 6 YCSB++ clients with one thread and 4 threads each. Unlike IcyTable, HBase does not benefit from different batch sizes. We also observed that HBase is less robust than IcyTable at handling large number of client threads.

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

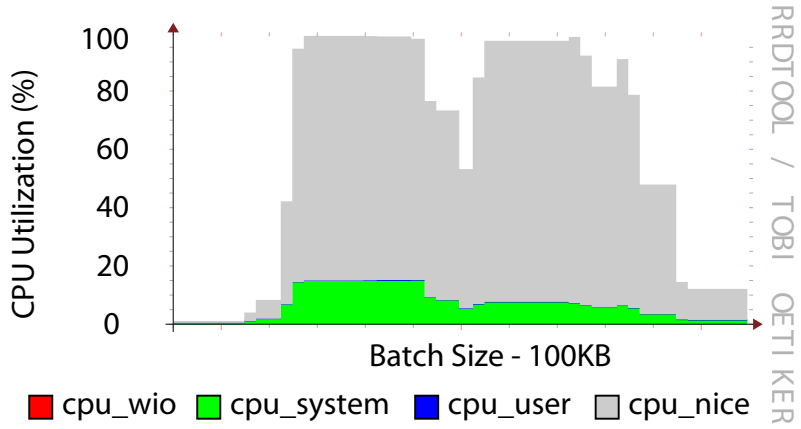

Figure 4. A single client inserting records in a 6-node IcyTable cluster becomes CPU limited resulting underutilized servers and low overall throughput (see Figure 3).

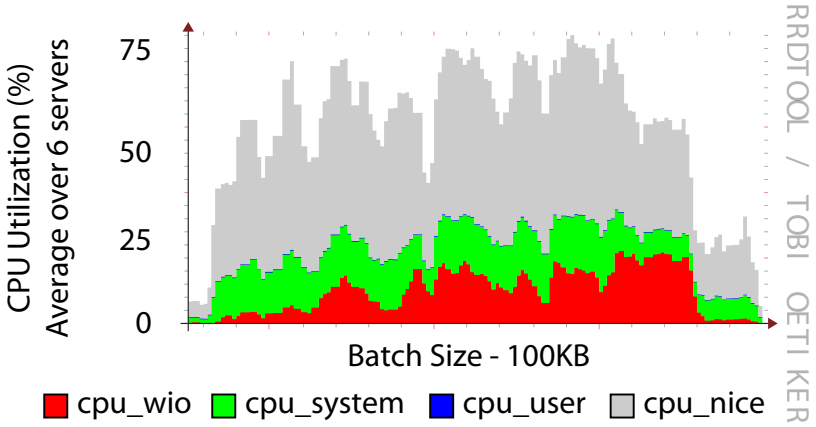

Figure 5. IcyTable servers are saturated when 300 threads insert records using a 100 KB batch (see Figure 3).

3.2 Weak consistency due to batch writing

One key side-effect of batching is that any newly written objects are not written to the server until the batch is full or the time-out on the buffer expires. Although both HBase and IcyTable support strong consistency [3], such delayed writes violate the read-after-write consistency expected by many applications; that is, a client may fail to read the most recently written object.

To understand this cost of batch writing, we used ZooKeeper-based producer-consumer abstraction with a two YCSB++ client setup. C_1 inserts 1 million records in one table and randomly samples 1% of these inserts to be enqueued at the ZooKeeper server, while C_2 tries to read the keys inserted in the ZooKeeper queue. We use four different batch sizes (10 KB, 100 KB, 1 MB and 10 MB) to measure the "read-after-write" time lag observed in both HBase and IcyTable.

Figure 6 shows a cumulative distribution of the time lag as observed by C_2. This data excludes the time lag of any keys that are read successfully the first time C_2 tries to do so. Out of 10,000 keys inserted in ZooKeeper, less that 1% keys experience a lag for 10 KB batch in both HBase and IcyTable, 1.2% and 7.4% of the keys experience a time lag for a 100 KB batch in IcyTable and HBase respectively, 14% and 17% for a 1 MB batch, and 33% and 23% for a 10 MB batch in

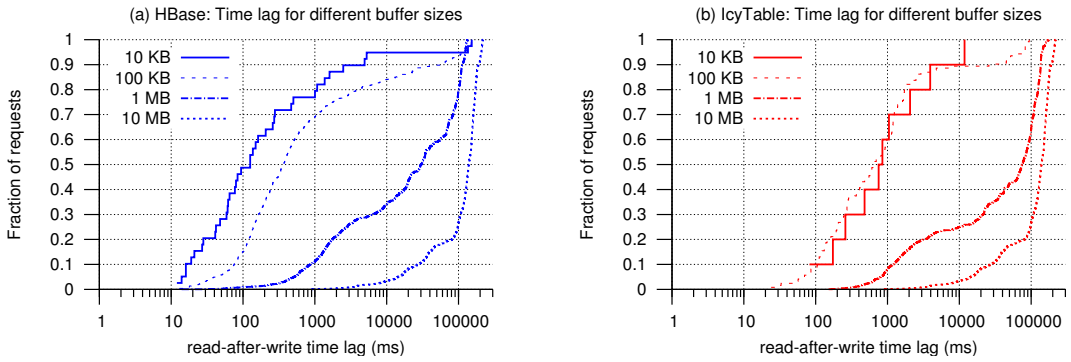

Figure 6. CDF of read-after-write time lag for different batch sizes

IcyTable and HBase respectively. This happens because the smallest batch fills up quickly and is flushed to the server more often, while the largest batch takes longer to fill up.

This phenomenon also explains the distribution of the time lag for the different batch sizes in Figure 6. For the smallest batch size (10KB), HBase has a median lag of 100 ms and and at most 150 seconds of lag, while IcyTable has an order of magnitude higher median (about 900 ms) and an order of magnitude lower maximum lag (about 10 seconds). However, the time lag for both systems is similar for all higher batch sizes; for the largest batch size (10 MB), the median lag is 140 seconds and maximum lag is 200 seconds. Interestingly, we observe that large batches may cause some keys to be visible more than 100 seconds after they were written by other clients. In summary, with large batched writes, the programmer must be prepared to cope with read-after-write time lags in the order of minutes.

3.3 Table pre-splitting

For ingest heavy workloads, if the key distribution is known a priori, YCSB++ can enable clients to pre-split a table such that the key ranges (and corresponding tablets) are well balanced across all tablet servers. This allows a table store to avoid splitting tablets during the ingest phase, which often results in high ingest performance. However, the number of tablets (or key ranges) that are created after a pre-split has a tradeoff: fewer tablets may result in a load imbalance while many tablets may increase load (memory and CPU) on tablet servers. Moreover, in the latter case, having more tablets incurs higher memory pressure from large number of memstores leading to less memory per memstore and frequent minor compactions.

As Table 2 shows, there are six phases in our pre-splitting illustrative experiment. Various

Phase	Workload
Pre-load	Pre-load 6M rows in range $[0, 12B]$
Pre-split	Pre-split tablet $[0, 72M]$ evenly
Load	Load 48M rows in range $[0, 72M]$
Measurement 1	Half update and half read for 4 minutes with 600 ops/s target
Sleep	Sleep for 5 minutes
Measurement 2	Same as measurement one

Table 2. Different phases in pre-split experiment

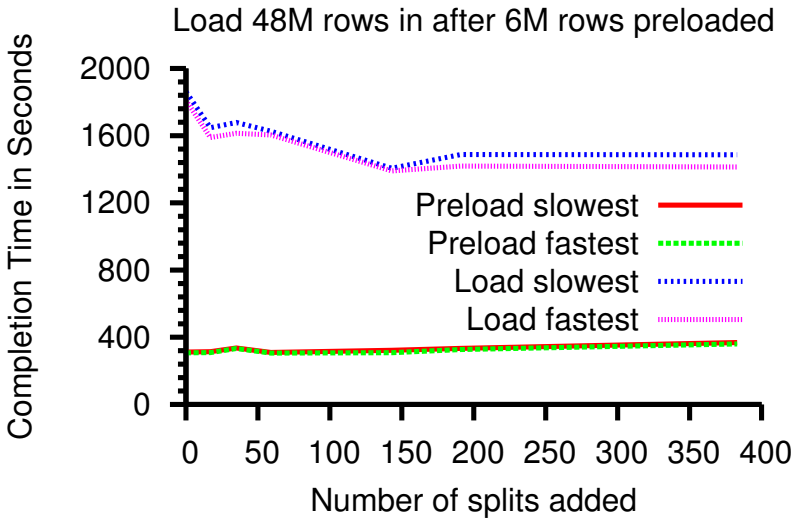

Figure 7. Load test with pre-splits

number of pre-splits are added after pre-load phase to understand their effect on the load phase. Keys are distributed evenly across the whole range for every phase. In the pre-split phase, we split uniformly in the range $[0, 72M]$. There are two measurement phases with low target operation throughput to measure the latency of read operations. There is a sleep phase in between to let the system be idle and trigger compactions.

We use three clients to run the workload against IcyTable. Each tablet server holds $1GB$ for memstore files. Figure 7 shows the shortest and longest completion time among three clients for pre-load and load phases with different number of pre-splits added to the table. Pre-load phase is always operated on an empty table, so it takes about the same amount of time. More pre-split helps reducing the completion time from at least 1810 seconds to 1486 seconds, which is about 20% improvement. With more pre-splits, the throughput of inserts is higher in the beginning because every server processes insert requests. As the table grows, more splits are triggered and the load is balanced across all the servers. With 17, 35 and 59 pre-splits, we observe a 10% improvement in completion time. 143 pre-splits are enough and because no more splits occurred in the load phase, we observer another 10% improvement in performance.

We also observed that that the read operations during the second measurement phase had uniformly low latencies of approximately 7 ms. During the first measurement phase, immediately following the load, however, read latencies were sometimes as high as 1500 ms or more. To explain this phenomenon we consulted our Otus monitoring tool for metrics recorded during this run. Figure 8 shows the instantaneous read latency (as extracted from YCSB++) and the number of running concurrent minor and major compactions (as extracted from IcyTable) over the course of the first measurement phase for the 143 pre-split run. We note that the presence of a high number of major and minor compactions is correlated to peaks in read latencies, which gradually drop off as the compactions complete. This is another example of how using our Otus monitoring tool helped us form a hypothesis linking benchmark performance to the internal state of the table store.

If the distribution of pre-loaded data matches the distribution of future data, and there are already enough tablets to load balance the servers, little benefit is expected from pre-splitting. When we changed the pre-load distribution to match the load distribution and repeated our experiments, we saw benefits as small as 7%. In this case, bulk load may still be effective because it may use

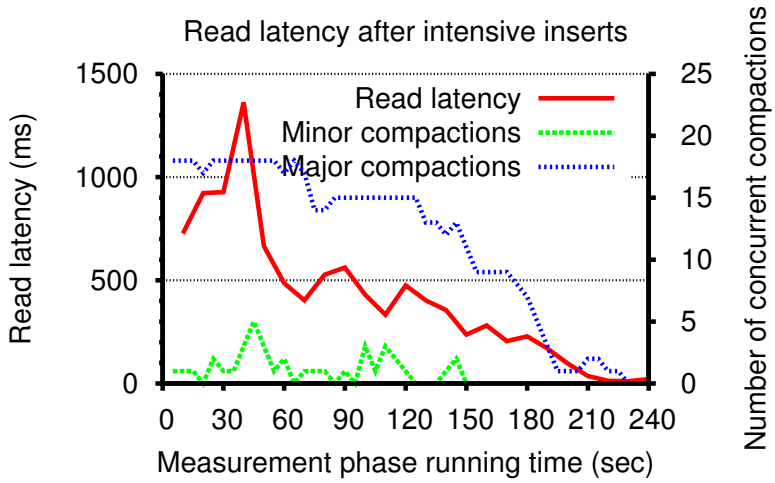

Figure 8. Read latency after intensive inserts

more efficient code path for the construction of store files.

3.4 Bulk loading using Hadoop

Recall that YCSB++ uses an external Hadoop/MapReduce application to test bulk insertions in HBase and IcyTable. Experiments to test the bulk loading mechanisms in HBase and IcyTable are modeled after the pre-splitting experiments in Section 3.3. Each experiment is divided into six phases shown in Figure 10. Phase (1), *MR pre-load*, executes the data generation MapReduce job to create 6 M rows with 10 cells per row. Phase (2), *Pre-load import*, refers to the adoption of the generated files by the tablet servers. Phase (3), *R/U 1*, executes the read/update workload for measurement purposes. Phases (4) & (5), *MR for load* and *Load import*, comprise the generation of 48 M additional rows for bulk loading. Phases (6) & (8) correspond to two additional R/U workload executions for latency measurement, separated by a 5 minute pause in Phase (7). The R/U phases are set up to place low load on the system. We measured the request latency as observed by the clients.

We ran three test cases using this section: (1) HBase with splits, (2) HBase without splits, and (3) IcyTable. In other words, some runs of the test against HBase experience dynamic rebalancing (splits and major compactions), and some do not. The results are shown in Figures 9–13.

HBase with splits: In this experiment, the first R/U phase has a query latency under 10 ms as shown in Figure 9(a). Phases R/U 2 and R/U 3 experience much higher latency around 100 ms,

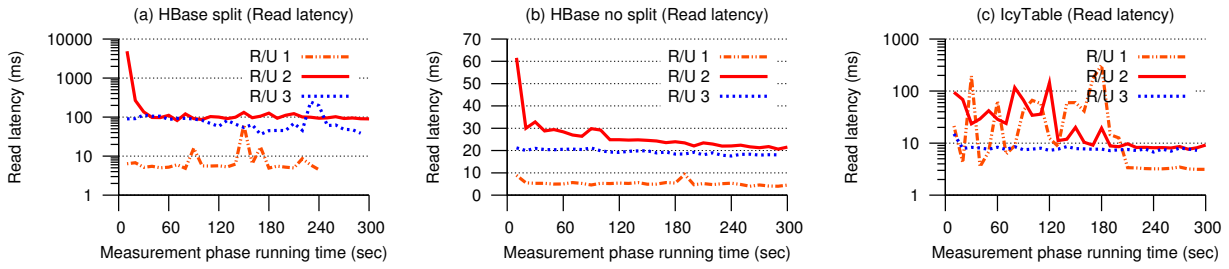

Figure 9. Read latency during the measurement phases for the bulk load experiments

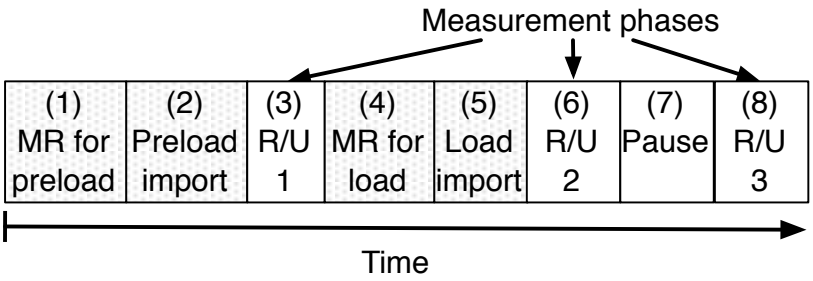

Figure 10. Six phase experiment used to understand the bulk load feature in HBase and IcyTable

Scenario		Rows 10^6	MR min:sec	Load sec	Total min:sec
HBase split	Preload	6	1:09	7	1:16
	Load	48	7:43	277	12:20
HBase nosplit	Preload	6	1:27	6	1:34
	Load	48	7:01	27	7:27
IcyTable	Preload	6	1:16	3	1:19
	Load	48	4:55	5	5:00

Table 3. Bulk load running times

and as high as 5000 ms at the beginning of second R/U phase. This high latency in R/U 2 can be explained by the time needed to read the newly bulk loaded data to warm up the cache in the tablet servers. The subsequent high latency (above 100 ms / query) is due to the work performed by the tablet servers to split tablets and major compact a large number of store files. Figure 11 shows the number of store files and tablets in the table, as well as the compaction rate on a timeline for the duration of the experiment.

HBase without splits: In this scenario, no splits occur throughout the duration of the experiment. This is confirmed by the tablet count and number of store files metrics shown in Figure 12. All the loaded files belong to a single tablet, thus all the queries are processed by a single tablet server. Compared to the previous case with split and compaction activity, the query latency during the R/U phases remains relatively low during, between 5 and 30 ms, as shown in Figure 9(b).

We also observe that IcyTable implements a more aggressive split/compaction policy. Figure 13 shows the number of store files and compactions during the experiment. IcyTable starts splitting regions to balance the load as soon as the initial set of files is loaded. It also aggressively performs compactions as new write operations are performed. Figure 9(c) shows that the query latency increases following a bulk load, both in phases R/U 1 and R/U 2. The re-balancing and compaction work is performed in approximately 3 minutes, which affected the response time for R/U 1 and R/U 2. Phase R/U 3 exhibits low query latency as the system has already finished performing the background work.

3.5 Server-side filtering

By default, a row read or scan in YCSB from table stores such as HBase or IcyTable returns all columns to the client, potentially much more than the application may be interested in. Using server-side filtering, clients can pass a filter the server and possibly receive only the desired data. Consequently, this may reduce the network and computation overhead at the client.

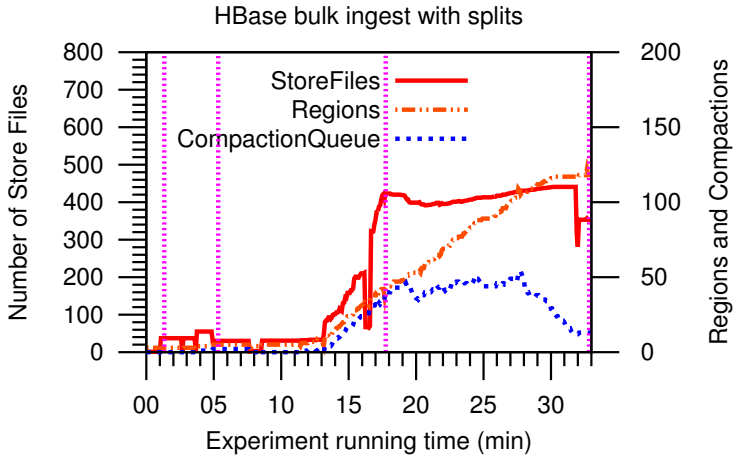

Figure 11. HBase bulk ingest with splits.

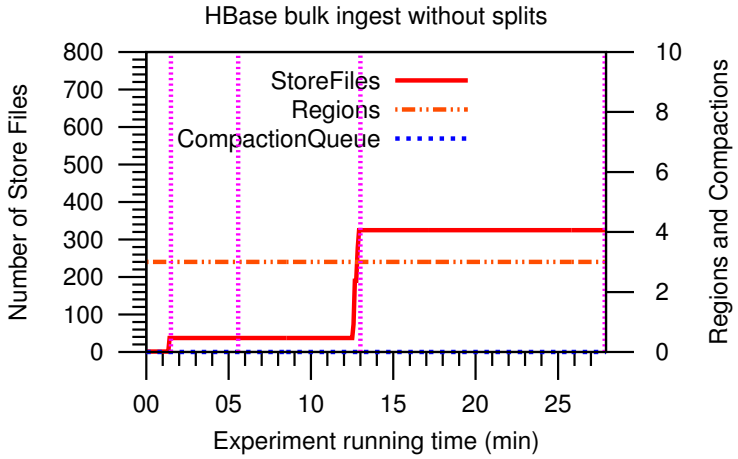

Figure 12. HBase bulk ingest without splits.

To benchmark the effects of filtering, we created a dataset with 100 times as much data per row as the standard YCSB configuration in 10 times as many fields (one hundred total). On this dataset, we ran a custom workload consisting of scans of constant length ranging from 1 to 1000, and measured throughput (number of rows read per second) at the client for both HBase and IcyTable with and without filtering enabled.

Figure 14 shows the results of filtering on IcyTable. We observe that for smaller scan lengths filtering does not provide any benefit – in fact, it degrades the performance. This behavior results from the implementation of IcyTable tablet servers. For scan requests, IcyTable uses a *scanner batch* which is the amount of data returned by the tablet server and the IcyTable servers return the requested data to the client only when the batch fills up. If the amount of data returned by a scan request that does not fill up the batch size, the IcyTable servers do "extra" work that degrades the performance. Figure 14 also shows that, for the unfiltered case, the collection is relatively easy as the server just has to read directly from the tablet. In the filtered case, however, the tablet server processes each row, eliminates all columns except the unfiltered one, adds the result to the batch and then processes the next row. Recall that in this set-up there are one hundred columns in

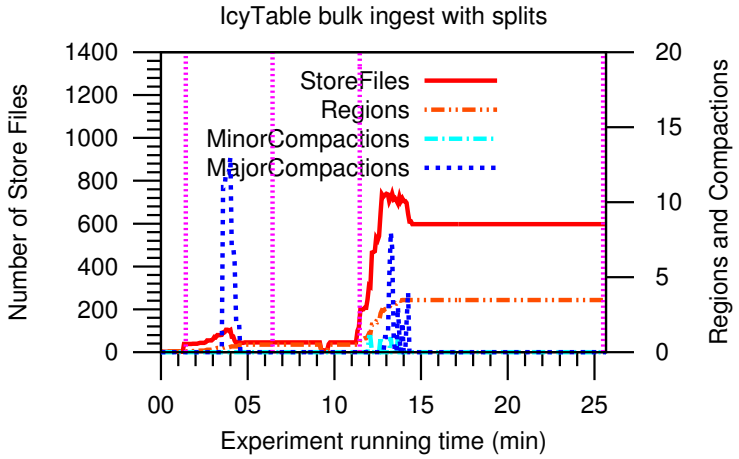

Figure 13. IcyTable bulk ingest.

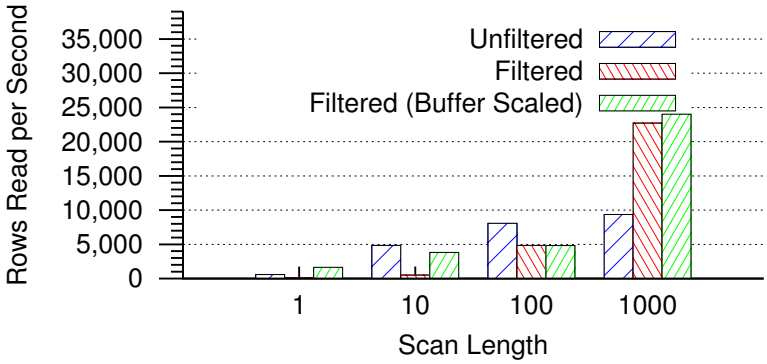

Figure 14. Performance of basic filtering on IcyTable for varying scan lengths

each row and filtering out all but one of them. As a result, in the filtered case, the tablet server is reading one hundred times as many rows as the unfiltered case and performing processing on them.

The third configuration in Figure 14 is the "filtered (buffer scaled)" case which sets the scanner batch size to a value more appropriate to the number of columns and rows we are interested in. We set the batch size to the number of records requested times 1000 bytes (the size of an entry in our set-up). Servers process an amount of data proportional to what the client actually is interested in and this results in a performance improvement for all cases. Our analysis shows that while server side filtering may be a tremendous win, the effects of server side overhead for techniques, such as prefetching, may potentially be magnified if one imposes server side operations in an ad-hoc, ill-informed manner.

To understand why filtering helps, let us examine the cluster metrics during the period of time corresponding to the run when scan length was set to 1000 scan length run in IcyTable. Let us examine the CPU load on the client as shown in Figure 15–left. During the first, filtering-disabled, segment of the run the client is busy processing records. Without filtering, all columns are being examined and copied into a buffer before being returned from YCSB++. With larger data size and higher number of columns to examine, the overhead piles up. On the other hand in the second, filtering enabled segment of the run when the client only has to examine a single column (1/100th the date of the unfiltered case), CPU load is comparatively light.

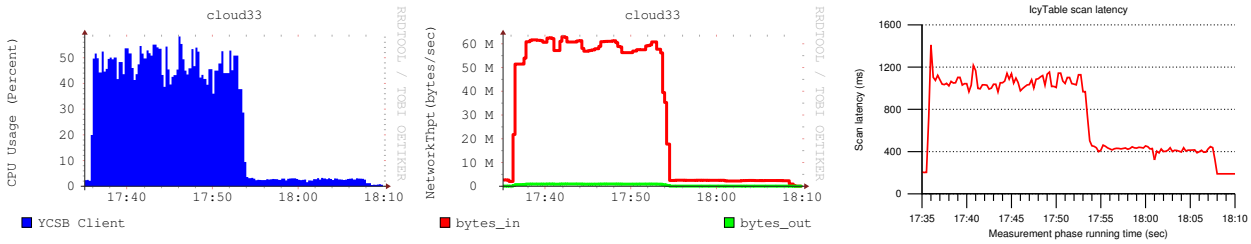

Figure 15. CPU load, network traffic, and scan latency on the YCSB++ client during the period of 1000-record scans. Filtering is enabled at 17:53

Network traffic, in Figure 15–center, shows the amount of data sent to the client. Since we saturated the network during our experiments (which use a 10GigE NIC), this confirms our hypothesis that filtering can have a linear effect on network traffic. Both network overhead and CPU processing impose a penalty on the latency of each scan operation, as shown in Figure 15–right, which drops the throughput. It is notable that the performance benefits were not directly proportional to the CPU overhead differences as there are still some constant computation overheads as well as network speed considerations and RTTs.

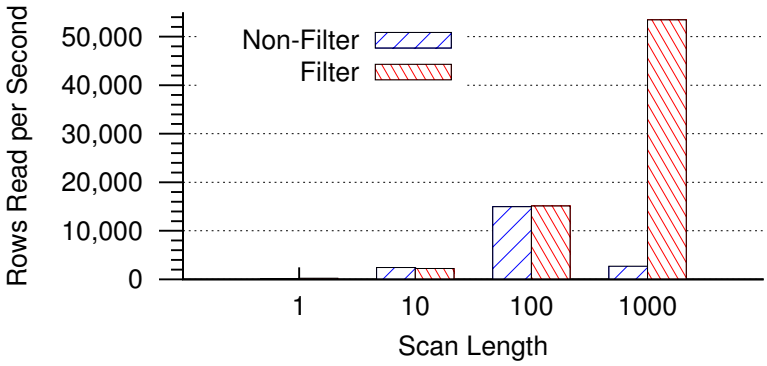

Figure 16. Performance of basic filtering on HBase for varying scan lengths

We performed the same experiment on HBase and Figure 16 shows the results of our analysis. Note that HBase does not require manipulation of the batch size because it performs less aggressive prefetching than IcyTable.

3.6 Fine-grained access control

Because only IcyTable supports fine grained access control, we could not perform a comparison with HBase.[3] However, YCSB++ has been extended to enable testing the cost and benefits of access control implementation in IcyTable. Appendix A has details about our extensions to YCSB and preliminary analysis.

[3] HBase has proposed to add security features in future releases [29]

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 and implement them to enhance the YCSB framework [12], which currently supports whole-system performance benchmarking for different workloads. All advanced features discussed in this paper and, in general, added to table stores are inspired from decades of research in traditional databases. We focus 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 [32, 46]. 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 [32]. Another study found that Amazon SimpleDB's eventually consistent model may cause users to experience stale reads and inter-item inconsistencies [33, 46]. Unlike our approach to measure the time required for first successful read, the SimpleDB study also checked if subsequent reads also returned stale values [46]. In contrast to SimpleDB's eventual consistency, both HBase and IcyTable provide strong data consistency if batch writing is disabled at their respective clients. Orthogonal approaches to understand weak consistency includes theoretical models [30], algorithmic properties [6, 17].

Ingest-intensive optimizations. Since many open-source table stores rely on systems services provided by the Hadoop framework, they rely of running an external Hadoop job that process and stores massive data-sets in a tabular form understood by the table store servers. This approach is common among users of HBase table store that uses a Hadoop job to insert data [2, 7]. An alternate approach adopted by the PNUTS system is to use an optimization-based planning phase before inserting the data [42]. This phase allows the system to gather statistics about the data-set that may lead to efficient splitting and balancing. In contrast to Hadoop based bulk load tool that we use, PNUTS-like planned approach may be useful for pre-splitting optimizations that rely on the distribution of keys in the data-set. Because our work is focused on benchmarking, the YCSB++ workload generator can be modified to include a planning phase (along with several heuristics) than can generate a range of dynamically changing key distributions to better understand the tradeoffs of using table pre-splits.

Server-side filtering. Shipping functions to the server is an old idea that has been studies in different forms including active disks (in a single-node setting) [39], MapReduce (in cloud computing) [14] and key-value stores (in wide-area networks) [19]. Because 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 [9, 13, 25, 26]. YCSB++ approach to testing server-side filtering has much narrower focus on regular expression based filters than the general abstractions proposed by HBase [25, 26].

5 Conclusion

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

Benchmarking tools like YCSB help with basic, single-phase workload testing of the core create-

read-update-delete interfaces, and we initially constructed multi-phase tests of advanced features by scripting combinations of YCSB tests. However, the extensibility of YCSB allows us to integrate our testing into YCSB++, a distributed multi-phase YCSB with an extended abstract table API for pre-splitting, bulk loading, server side filtering, and cell-level access control lists.

To enable 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, the network and CPU usage of these service, yields a powerful tool for understanding and improving scalable table store systems.

References

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

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

[3] HBase: The Hadoop Database. http://hadoop.apache.org/hbase/.

[4] The Apache Cassandra Project. http://cassandra.apache.org/hbase/.

[5] ZooKeeper. http://zookeeper.apache.org.

[6] Aiyer, A. S., Anderson, E., Li, X., Shah, M. A., and Wylie, J. J. Consistability: Describing usually consistent systems. In *HotDep 2008*.

[7] Barbuzzi, A., Michiardi, P., Biersack, E., and Boggia, G. Parallel bulk Insertion for large-scale analytics applications. In *Proceedings of the Workshop on Large-Scale Distribued Systems and Middleware (LADIS '10)* (2010).

[8] Brewer, E. A. Towards robust distributed systems. Keynote at ACM PODC 2000 (Portland OR).

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

[10] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A Distributed Storage System for Structured Data. In *USENIX OSDI* (2006).

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

[12] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *ACM SOCC 2010*.

[13] Dean, J. Designs, Lessons and Advice from Building Large Distributed Systems. Keynote at LADIS Workshop 2009 - http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf.

[14] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI* (2004).

[15] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. In *ACM SOSP 2007*.

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

[17] Fekete, A., and Ramamritham, K. Consistency Models for Replicated Data. In *Replication (LNCS 5959), 2010*.

[18] Fikes, A. Storage Architecture and Challenges (Jun 2010). Talk at the Google Faculty Summit 2010.

[19] Geambasu, R., Levy, A. A., Kohno, T., Krishnamurthy, A., and Levy, H. M. Comet: An Active Distributed Key-Value Store. In *USENIX OSDI 2010*.

[20] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *ACM SIGACT News, Vol 33, Issue 2* (June 1989).

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

[22] Graefe, G. Partitioned B-trees: A user's guide. In *BTW 2003*.

[23] Graefe, G., and Kuno, H. Fast Loads and Queries. In *Replication (LNCS 6830), 2010*.

[24] Hadoop. Apache Hadoop Project. http://hadoop.apache.org/.

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

[26] HBase-1002. HBase Coprocessors. http://hbaseblog.com/2010/11/30/hbase-coprocessors/.

[27] HDFS. The Hadoop Distributed File System: Architecture and Design.

[28] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC 2010*.

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

[30] Kraska, T., Hentschel, M., Alonso, G., and Kossmann, D. Consistency Rationing in the Cloud: Pay only when it matters. In *VLDB 2009*.

[31] Lakshman, A., and Malik, P. Cassandra - A Decentralized Structured Storage System. In *LADIS Workstop 2009*.

[32] Li, A., Yang, X., Kandula, S., and Zhang, M. CloudCmp: Comparing Public Cloud Providers. In *IMC 2009*.

[33] Liu, H. The cost of eventual consistency. http://huanliu.wordpress.com/2010/03/03/%EF%BB%BFthe-cost-of-eventual-consistency/.

[34] Massie, M. L., Chun, B. N., and Culler, D. E. The Ganglia distributed monitoring system: Design, implementation and experience.

[35] MySQLTips. Minimally Logging Bulk Load Inserts into SQL Server. http://www.mssqltips.com/tip.asp?tip=1185&home.

[36] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The log-structured merge-tree (LSM-tree). In *ACTA INFORMATICA 33(4), 1996*.

[37] Ren, K., López, J., and Gibson, G. A. Otus: Resource Attribution in Data-Intensive Clusters. In *MapReduce Workshop, 2011*.

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

[39] Riedel, E., Faloutsos, C., Gibson, G. A., and Nagle, D. Active Disks for Large-Scale Data Processing. In *IEEE Computer, June 2001.*

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

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

[42] Silberstein, A., Cooper, B. F., Srivastava, U., Vee, E., Yerneni, R., and Ramakrishnan, R. Efficient Bulk Insertions into a Distributed Ordered Table. In *ACM SIGMOD 2008.*

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

[44] v. G. Pohl, and Renner, M. *Munin: Graphisches Netzwerk-und System-Monitoring.* opensource press, 2008.

[45] Vogels, W. Eventually Consistent. In *ACM Queue, 6(6)* (Oct. 2008).

[46] Wada, H., Fekete, A., Zhao, L., Lee, K., and Liu, A. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective. In *CIDR 2011.*

Appendix A: Fine-grained access control

Cloud table stores are often used in a shared environment where multiple users and applications manipulate the same table even if each user or application may not always manipulate the same column family. In BigTable, a table and a column family form a unit of access control [10] and HBase proposes to support this model [29]. In this paper, the model is referred as a *schema-level* access control since an access control list is stored and checked at the schema/metadata level. This *schema-level* access control is not sufficient for all users; in an environment where entries in the same column require different levels of security clearance finer-grain access control is needed. One possible model for a fine-grain access control is proposed by IcyTable where an individual access control list is associated with each cell in a column. This model is referred to as *cell-level* access control since an access control list is stored along with each cell. In IcyTable, this fine-grain access control is used only for read permissions while write permissions are still limited to a *schema-level* access control.

YCSB++ supports both the *schema-level* and *cell-level* access control models. This is done by extending the DB client API to pass credentials and an access control list from YCSB client and workload executor to the table stores. We have implemented a *security* extension to YCSB's core workload executor to set credentials for read operations and *schema-level* access control at the beginning of each benchmark while a *cell-level* access control is passed to the DB client before each insert operation.

5.1 Fine-grained access control

While a fine-gain access control in IcyTable can offer a greater control over who can access a particular cell, the overhead for supporting this fine-grain access control may be significant since each access control list is added to every cell in a table. The overhead comes in two ways. First, more bytes are sent over a network and stored in disks at servers to support access control lists. Secondly, there is a computational overhead to process an access control list while inserting and to verify credentials when a cell is accessed. This type of overhead may depend on the number of entries (i.e. users and groups) in an access control list.

Row Key	Column Family	Column
≈ 12 bytes	3 bytes	6 bytes
Access Control	**Value**	**Timestamp**
≈ 100 bytes	2 bytes	8 bytes

Table 4. Table setting: a size of each column

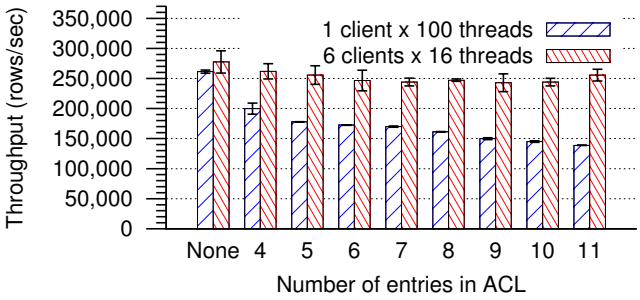

Figure 17. Insert Throughput. *Throughput decrease as the number of clauses increase in case where CPU is a limiting resource.*

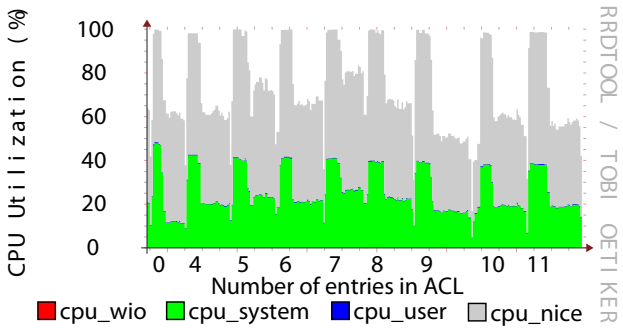

Figure 18. YCSB++ client CPU utilization. *A single client inserting records becomes CPU limited resulting in lower throughput as the number of entries in each access control list increases.*

To understand the overhead of additional bytes, we designed an experiment where the length of an access control list is much larger than the combined length of the key and value. Table 4 shows the settings used in this experiment.

To measure the computational overhead with the effects of space overhead, we constructed access control lists such that the total length of each access control list stays the same while each entry within the list becomes smaller. Each access control list consists of three segments of entries. First a series of entries whose combined size is fixed while their number increases. By increasing their number while decreasing the size of each entry this segment allows us to increase the total number of entries in the access control list without increasing its size. The second segment is two fixed entries that appear on every access control list. We use these entries to test our credentials. Since we are interested in measuring overhead for this experiment we configured the credentials and these entries such that all scans would succeed. Finally, the last entry in the access control list is the name of the row so that all access control lists for the column are different from each other.

Two benchmarks are used to measure the security overhead: A benchmark for insert operations and another for scan operations. For insert operations, only one record is inserted per YCSB request. However, there is a buffer inside the database client to buffer these operations before sending them to the server. In the case of IcyTable, a client is configured to buffer up to 100KB before sending requests to a server. For scan operations, each operation scans 1,000 records with a random starting key. From the results presented in Section 3, we decided to pre-split a table into 24 tablets before starting the insert experiment so that all servers would receive even load from the start. A table is empty before insertion with only single column. For each round of the benchmark, 48 million rows are inserted into the table and 320 million rows are scanned. We used two settings for YCSB++ clients: a single client with 100 threads and 6 clients using our coordination extensions with 16 threads each (96 threads in total).

Insert – Figure 17 reports the insert throughput, i.e. the number of records inserted per second, for varying numbers of entries in each access control list. A value of zero entries means that no security was used. An average of three runs with a standard deviation shown. In the case of a single client setting, we observe that the throughput decreases as the number of entries in each access control list increases from a 24% throughput reduction in the 4-entry access control list setting to a 47% throughput reduction in the 11-entry access control list setting. Using our Otus monitor, Figure 18 shows that the YCSB++ client is running at almost 100% CPU utilization. So as the number of entries in each access control list increases, more computation is required to process the additional entries, reducing the insert throughput. Once 6 clients are used, the insert throughput does not decrease as the number of entries in each access control list increased although the average CPU utilization of all clients increased from 18% in a no security setting and 22% in a 4-entry setting to 29% in a 11-entries setting.

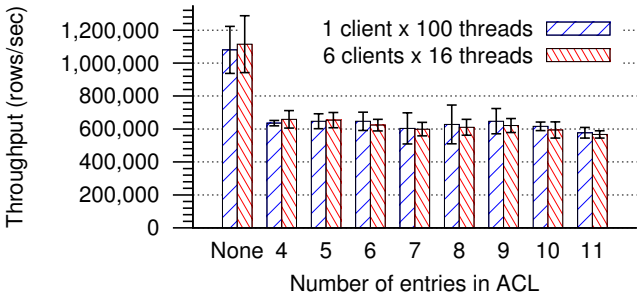

Figure 19. Scan Throughput. *A significant drop once a fine-gain security is used.*

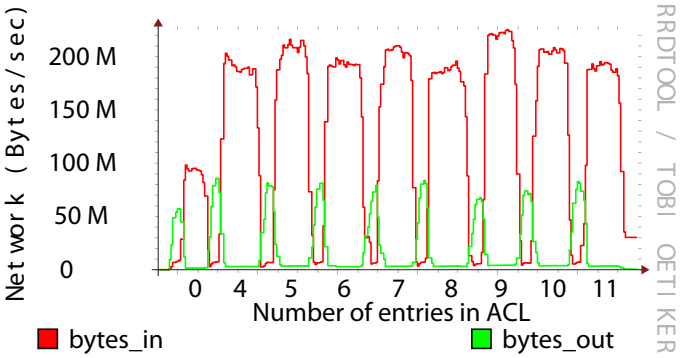

Figure 20. Aggregate network traffic of 6 YCSB++ clients. *Significantly more bytes are sent from* IcyTable *servers to* YCSB++ *client while scanning records with access control lists.*

Scan – Figure 17 reports the scan throughput, i.e., number of records scanned per second, for different number of entries in each access control list. Again, a value of zero means no security used. An average of three runs with a standard deviation shown. For both a single client setting and 6-clients setting, we observe a significant drop (40% - 50%) in a scan throughput once a fine-gain access control is used while an increase in the number of entries in each access control list does not form any obvious overhead. Using Otus, Figure 20 shows that significant more data is sent from a server to a client once a security is used. IcyTable opts to send a whole access control list back to a client, although in many situations an access control list yields no benefit to the client once an access is allowed.