Booz | Allen | Hamilton

Benchmarking the Apache Accumulo Distributed Key–Value Store

[Version 2.1; 10/6/2014]

Ranjan Sen <sen_ranjan@bah.com>

Andrew Farris <farris_andrew@bah.com>

Peter Guerra <guerra_peter@bah.com>

Table of Contents

Table of Figures

List of Tables

1 ABSTRACT

With the rapid growth and diversity of distributed key–value stores, there is a growing interest in benchmarking and understanding the performance characteristics of these storage systems. In this paper, we present the results of benchmarking the Apache Accumulo scalable key–value store using a series of stress tests that execute large-scale graph operations. These tests are readily available with the Accumulo open source distribution but have not been reported in any earlier benchmark studies. We ran these benchmark tests on two different target environments: a cluster of physical machines optimized for large-scale analytics and a cluster of inexpensive virtual machines (VM) executing on a publicly available compute cloud. As a counterpoint to these benchmarks, we also provide the results of running the Yahoo! Cloud Service Benchmark (YCSB) in the same environments.

1.1 Keywords

Scalable table stores; Benchmarking; NoSQL; Apache Accumulo

1.2 Executive Summary

Distributed computing has grown in recent years as a result of the increased availability and maturity of open source tools that put cost-effective parallel computing in the hands of many organizations. Open source platforms such as Apache Hadoop and Apache HBase allow organizations to deploy sophisticated parallel computing algorithms by leveraging commodity hardware or cloud infrastructure such as Amazon Web Services (AWS). These platforms allow organizations to manage data at a scale that was previously impossible.

As companies and government organizations look to this newer technology, there are few benchmark studies as to what the upper limits of scale or true linear scalability of these platforms are. There is anecdotal evidence and some studies but fewer scientific studies that look at the upper limits of the processing power of these platforms.

Because Accumulo is widely used in U.S. Federal Government systems, we chose to study the scalability of this platform by performing numerous ingest and query tests to determine how much data can be processed. This paper describes the methods we used to benchmark Accumulo on two platforms: EMC^2 Analytics Workbench (AWB), which shows scalability on bare-metal servers, and Amazon Elastic Compute Cloud (Amazon EC2) instances to show scalability on virtualized infrastructure.

Our tests included ingest and query tests as well as measures using YCSB++. We took constant performance measures on Accumulo running on 100, 300, 500, and 1,000 nodes to determine whether the data processing was truly linear and to determine how much data could be processed using this platform.

During the testing, we observed high ingest rates in Accumulo, reaching a maximum throughput of more than 150 million records per second for a presplit table on 1,000 nodes of the EMC^2 AWB. Average throughput can also be near 100 million records per second for presplit tables. We also saw improvement in ingest rates for non-presplit tables after a sufficiently large number of tablets are created and distributed across the cluster.

In the Walker and Batch Walker tests, we found that latencies were generally in the tens of milliseconds and microseconds range, respectively, for both presplit and non-presplit tables. In mixed tests, we found more stable performance for the presplit table, where both Ingest and Walker tests were running at the same time. For non-presplit tables, the latencies increased as more ingest occurred.

The scalability is almost linear in the case of presplit tables, because more test loads are introduced in proportion to the size of the test cluster. This is not quite so for a non-presplit tables.

The performance and scalability figures are generally better than most distributed table stores for which benchmark data is available. The presplit table tests provide the best-case scenario; the non-presplit table cases provided the worst-case scenario. In practice, we expect performance to lie within these two extreme cases. Our tests were based on random generation of row references. In a weighted scheme of row reference, we expect better performance for the non-presplit table.

Although this study is not intended as a bake-off against different technologies, it is expected that the numbers expressed here prove the viability of Accumulo as a scalable key–value store for large quantities of data and that the platform is linearly scalable at the petabyte range and possibly beyond.

2 INTRODUCTION

Distributed key–value stores, also known as *scalable table stores,* provide a lightweight, cost-effective, scalable, and available alternative to traditional relational databases [1, 2]. Today, scalable table stores such as BigTable [3], Amazon DynamoDB [4], HBase [5], Apache Cassandra [6], Voldemort [7], and Accumulo [8] are becoming an essential part of Internet services. These tools are used for high-volume, data-intensive applications, such as business analytics and scientific data analysis [9, 10]. In some cases, they are available as a cloud service, such as Amazon SimpleDB [11] and Microsoft Azure SQL Database [12] as well as application platforms such as Google App Engine [13] and Yahoo! Query Language [14].

Benchmarks have been used in comparing performance of computing resources ranging from hardware systems such as CPUs, disks, and networks to software systems, including databases and application servers. Such benchmarks include the general-purpose Transaction Processing Council (TPC) benchmarks [15], benchmarks for e-commerce systems such as SPECweb [16,17], as well as other benchmarks for cloud computing [18]. The criteria for a successful benchmark are relevance to an application domain, portability to allow benchmarking of different systems, scalability to support benchmarking of larger systems, and simplicity so that results are understandable [19]. The types of application we consider here are those that provide data-serving services in which a user reads and writes to a database. Individual benchmarks exist for several scalable table stores, including BigTable [20], Cassandra [21], and Hypertable [22]. A standard benchmark known as the YCSB [23] and its extension, known as YCSB++ [24], have been developed to apply a uniform set of tests to multiple scalable table stores. YCSB has been used in comparing different systems such as Cassandra, HBase, PNUTS [24], and sharded MySQL [25]. YCSB++ [26] compares the advanced features of HBase and Accumulo, such as server-side programming and the fine-grained data access control the latter provides.

Many applications in this class use multidimensional indexed data structures, such as graphs [27], to construct and maintain attribute relationships. The published benchmarks for scalable table stores do not address operations on multidimensional data structures. The Accumulo open source distribution contains a set of "continuous" stress tests that can be used as standard benchmarks for addressing applications that use indexed structured graphs.

This paper presents the use of the Accumulo "continuous" test suite to perform benchmark testing of Accumulo. It addresses operations on indexed data structures in addition to standard workloads on two representative large compute cluster environments. We do not compare performance of different table store products but rather investigate the development of a suitable benchmark that exposes and highlights the performance characteristics of Accumulo.

In what follows, we highlight the basic architecture and features of Accumulo, a distributed key–value store based on the system described in Google's BigTable paper. We identify benchmarks that highlight its performance characteristics and present results of running the benchmarks on the AWB [28] and on Amazon EC2 [29]. We then present the results of running the benchmark tests and concluding remarks.

2.1 Introducing Apache Accumulo

Accumulo is based on the Google BigTable design but includes several unique features that distinguish it from other, existing BigTable implementations. One feature is the Accumulo iterator framework, which embeds user-programmed functionality into different Log-Structured Merge Tree (LSM-tree) stages [30]. Accumulo also provides cell-level security that enables fine-grained data access control.

An Accumulo cluster runs on a pool of machines and operates over the Apache Hadoop Distributed File System (HDFS) [31]. Data is manipulated in Accumulo using a client application programming interface (API) or is used as input or output for Hadoop MapReduce jobs [31]. An Accumulo cluster consists of a single master server and many tablet servers, write-ahead logger servers, a garbage collector process, monitoring service, and many clients. The Accumulo master server is responsible for managing and coordinating tablet server activities, including responding to failures. The master uses Apache ZooKeeper as a coordination service [34] for managing and distributing tablets across the tablet servers. The master server handles table creation, alteration, and deletion requests from clients. It coordinates startup, graceful shutdown, and recovery of changes in write-ahead logs when tablet servers fail.

An Accumulo table is a sparse, multidimensional, sorted map of key–value pairs. The multidimensional keys, as shown in Figure 2-1, consist of a row ID, column key, and timestamp. Column keys consist of separate column family, qualifier, and visibility elements. All elements of the key and value are stored as bytes except for the timestamp, which is stored as a 64-bit integer, equivalent to the Java long type. Tables are sorted in ascending order by the row ID and column keys and in descending order by the timestamps. As such, Accumulo tables can be thought of as being indexed by the full key, including row ID, column family, column qualifier, visibility, and timestamp. Accumulo column families are dynamic and do not need to be defined at the time of table creation. Column families can be organized into locality groups to control how data is stored on disk. This allows data that is accessed together to be stored separately from other data in the same table.

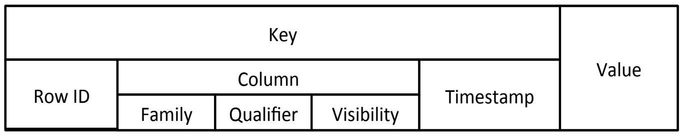

Figure 2-1: Apache Accumulo Row Structure

(The key consists of a row ID, column family, qualifier and visibility, and timestamp.)

Each table is partitioned into tablets, which are distributed to the tablet servers. Each tablet represents a range of rows from a table and is assigned to a single tablet server. A single row as defined by a row ID is never split across multiple tablets. When using a table design that employs large rows (e.g. a single row ID has a large number of columns), a tablet can contain a single row. A tablet server typically manages many tablets. It receives writes to these tablets from clients, persists writes to a write-ahead log, sorts new key–value pairs in memory, periodically flushing sorted key–value pairs to new files in HDFS. The tablet server also responds to read and scan requests from clients, forming a merge-sorted view of all keys and values from each of the tablet's files and the sorted in-memory store. A tablet server uses multiple write-ahead logger servers to write updates to local on-disk storage distributed across multiple physical machines within the cluster.

When a write operation arrives at a tablet server, it is written into a write-ahead log, and then inserted into the in-memory store, known as the *MemTable.* When the MemTable reaches a certain size, the tablet server writes out the sorted key–value pairs to a file in HDFS, called an *RFile.*

When a read operation arrives, the tablet server performs a binary search in the MemTable and the in-memory indexes associated each RFile in HDFS to find the relevant values. If clients are performing a range scan, many ordered key–value pairs are returned to the client from the MemTable and RFiles via a merge-sort process.

To manage the number of RFiles per tablet, the tablet server periodically performs minor and major compactions of the files in a tablet server. *Compactions* combine one or more RFiles into newer, likely larger files in HDFS. During compactions, deleted key–value pairs are permanently removed from the store. The garbage collector process removes any unnecessary files asynchronously. Changes are made atomically on a per-row basis on the tablet server. The reads are optimized to quickly retrieve the value associated with a given key and to efficiently return ranges of consecutive keys and their associated values. Methods to efficiently retrieve several ranges of rows simultaneously are available. Figure 2-2, taken from S. Patil, et. Al (2011) [26], offers a schematic diagram about the relationship between the different storage components.

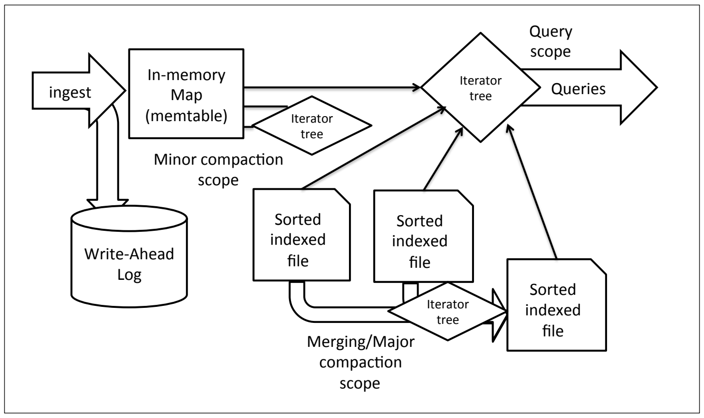

Figure 2-2: Apache Accumulo Tablet Storage Components

Accumulo provides table features such as locality groups, constraints, bloom filters, and iterators. *Locality groups* enable sets of column families to be stored separately on disk to allow clients to scan over columns that are frequently used together efficiently. *Constraints* are policies applied on mutations at data insert time. When the bloom filter is enabled, Accumulo uses a memory-based data structure to determine whether a file contains a given key before opening the file. *Iterators* provide a modular mechanism for adding functionality to be executed by tablet servers when scanning or compacting data. This allows users to efficiently summarize, filter, and aggregate data. Versioning iterators use the timestamps within a key and can combine several keys based on their timestamp values. Filtering iterators return only key–value pairs that satisfy the filter logic. Aggregating iterators are applied across the values associated with any keys that share row ID, column family, and column qualifier. This is similar to the Reduce step in MapReduce, which applies some function to all the values with a particular key.

Indices are used to support lookups via more than one attribute of an entity. Consider a main table whose rows are key–value pairs that represent these entities. To build an index to the rows based on their values, it is possible to construct an index table with the value as the row ID and the columns as the row IDs of the main table, as shown in **Figure 2-3**. Accumulo tables can support any number of columns without specifying them beforehand [32]. Lookups are done by scanning the index table first for occurrences of the desired values in the columns specified.

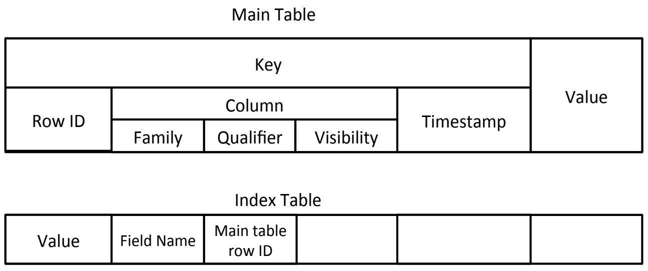

Figure 2-3: Apache Accumulo Table Structures

Accumulo is well suited to storing entities and their attributes, especially if the attributes are sparse. It is often useful to join several datasets together on common attributes within the same table, which allows for the representation of graphs with nodes and their attributes and connections to other nodes. Iterators can be configured to add the value of all mutations applied with the same key (designating a pair of nodes). Similarly, document-partitioned indexing to look for records that match a set of given criteria can be achieved. Accumulo supports shared indexing, where the matches of multiple search terms are performed at the tablet servers, thus avoiding costly network traffic. Analytics applications can be developed using MapReduce and iterators in conjunction with Accumulo tables [32].

Accumulo tables can be used as the source and destination of MapReduce jobs. Aggregating iterators may be used as replacement for the Reduce phase of MapReduce jobs when it is not necessary to process entire datasets. In this scenario, each tablet server hosts one or more parts of a table as tablets while multiple instances of an iterator are run in parallel across many tablet servers. In this configuration one cannot necessarily know when all values from a range have been observed. This disallows calculating functions such as the average of values of all the rows visited by the scan within the iterator itself. However, many applications can benefit from this feature. Using Accumulo iterators in this way enables efficient and fast lookups and updates of distinct ranges of information using random access patterns, in a manner complementary to the purely sequential access model of MapReduce.

To increase throughput when reading frequently accessed entries, Accumulo employs a block cache. The block cache can be configured on a per-table basis, and all tablets hosted on a single tablet server share a single resource pool. The block cache may have a significant effect on alleviating hot spots as well as reducing query latency.

High-speed ingest into tables is achieved by presplitting a table into multiple tablets and adding data to these tablets in parallel. This avoids a delay in parallelism that results in waiting for tablets to split when they reach the per-tablet size limit. The split threshold is controlled by the table configuration parameter table.split.threshold [32].

A *presplit table* is a table that has already been split into tablets, each of which stores a distinct sorted range of the rows distributed in the cluster. If we store X rows in the table, we can create X/p tablets, where the i-th tablet, $0 \leq i \leq \frac{X}{p} - 1$, will have row range of $\{ip + 1, ip + 2 \ \dots (i + 1)p\}$. The rows of the table have row IDs $\{1, 2 \dots X\}$. If there are N nodes in the cluster then, for an even distribution of tablets, each node will host on average p/N tablets.

Accumulo is capable of scaling to high rates of ingest. In many applications, scalability depends on both the number of tablet servers and the number of ingest clients sending data into the system. It is often the case that a single ingest client is unable to sustain a rate of ingest that fully uses server capacity. In operational settings where high rates of ingest are paramount, clusters are often configured to dedicate some number of machines solely to running ingest clients.

Accumulo implements a security mechanism known as *cell-level security.* Every key–value pair has its own security label, stored under the column visibility element of the key, that is used to determine whether a given user meets the security requirements to read the value. This enables data of various security levels to be stored within the same row and users of varying degrees of access to query the same table while preserving confidentiality. Production systems can integrate with public key infrastructure and designate client code in the query layer to negotiate to authenticate users and retrieve authorization tokens. This requires users to specify only the information necessary to authenticate them to the system.

3 BENCHMARKING DISTRIBUTED TABLE STORES

Benchmarks need to be relevant to the application domain where the table store technology is used. They should be scalable to allow benchmarking larger systems; portable so that they can be used for different products; and simple, easy to run, and repeat.

The performance evaluation of BigTable [20] addressed two criteria: performance and scale. The operations of interest were sequential and random reads and writes on tables of known sizes. The workload was controlled to be 1 GB of data per tablet server. In sequential read or writes, row keys from a sorted range were used. In random reads and writes, the rows used were generated randomly. Multiple clients were used in load generation. In the Scaling test, the number of tablet servers was increased along with a proportional increase in load. Based on this work, a benchmark was developed for Hypertable that used Zipfian distribution to generate workloads of random inserts and sequential scans [22]. Netflix developed the write-oriented benchmark stress tool for Cassandra in Amazon EC2 instances (M1 Extra Large—4 CPU, 15 GB of RAM, and 4x400 GB disk), where the writes to disk were performed but the reads were limited to memory [29]. Their test also included scalability tracking and was run across three availability zones in the Amazon cloud for 96 instances in each. With 60 client instances, this test generated 1.1 million writes per second and was complete in 2 hours, creating a 7.2 GB table of records.

The YCSB is a standard data-serving benchmark focusing on portability and extensibility. The benchmark provides a number of adapters that allow it to be run against a variety of different database systems. YCSB provides two tiers of benchmarks: performance and scalability. The performance tier focuses on the latency of requests when the database is under load. There is an inherent tradeoff between latency and throughput: As load increases, latency increases due to contention for hardware resources such as disk, CPU, and network. Application designers must choose system configurations that provide an acceptable balance of latency and throughput for their application. To understand this relationship, the benchmark measures query response time (latency) in relation ingest rate (throughput) while increasing the amount of data ingested to induce system load. To measure scaling, latency is measured while increasing the number of machines in the system in addition to proportionally increasing dataset size and throughput. In scalable system, latency should remain constant despite the increase in dataset size, throughput and number of machines. The results of using YCSB in a small cluster for four widely used systems—Cassandra, HBase, PNUTS, and a simple shared MySQL implementation—are available [23].

The YCSB benchmark consists of a workload generator and a workload configurator. The workload generator includes a base core workload class that activates operations, which can be customized via the workload configurators to carry out insert, update, or read operations using different statistical distributions on target databases. The benchmarks can be run with different database configuration parameters and the results of throughput and latency obtained. The core workload class can be extended to control the environment and activate operations that expose additional performance characteristics.

The workloads in the core package are applied to the same basic application type. In this application, there is a table of records, each with a variable number of fields. Each record is identified with a string-type primary key. The values of fields are a random string of ASCII characters of known length. Each operation against the data store is randomly chosen to insert a record, to update a record by replacing the value of one field, to read a randomly chosen field or all fields of a record, or to scan records starting at a randomly chosen record key. The number of records to scan is randomly chosen. The distribution of scan lengths is chosen as part of the workload. So, the scan method takes an initial key and the number of records to scan.

The workload client randomly decides which operation to perform (insert, update, read, or scan), which record to read or write, how many records to scan, and so on; random distributions govern these decisions. The distributions can be uniform, Zipfian, latest, or multinomial. In a uniform distribution, an item is chosen uniformly at random, with each choice being equally likely. In Zipfian distribution, some records are more likely to be chosen than others. For example, in the case of an insert operation, a uniform distribution models a scenario in which the actual insertion of a record does not change its probability of choice for insertion. In the case of Zipfian distribution, a record may have a measure of popularity, where high popularity makes its more probable to be inserted than lower-popularity options. The latest distribution is similar to the Zipfian, except that the more likely records are the most recently inserted ones. In the multinomial distribution, the probabilities of choice for each item can be specified. For example, we may assign a probability of 0.95 to the read operation and a probability of 0 to scan or insert. The distribution is assigned to two main choices: which operation to perform and which record to read and write. The core workload package consists of five combinations based on the ratio of these operations and distribution for selection of records. These workloads model different application scenarios and are presented in **Table 3-1**.

Workload	Workload Characteristics	Application Scenario	Performance Evaluation
Workload A (update heavy)	50% read, 50% update, with Zipfian distribution of operations	Session store recording recent actions in a user session (read and update).	Latency vs. throughput for both read and update operations
Workload B (read heavy)	95% read, 5% update, with Zipfian distribution	Photo tagging, where adding a tag is an update operation; most operations are to read tags	Read and update latencies vs. throughputs (same as above but read heavy)
Workload C (read only)	100% read, with Zipfian distribution	User profile cache, where profiles are constructed elsewhere	Results similar to those of workload B
Workload D (read latest):	95% read, 5% insert with distribution, where the latest transaction repeats more	User status updates (people want to read the latest status)	Similar result as workload B
Workload E (short ranges):	95% scan, 5% insert, with Zipfian distribution	Reads the first record in scan/uniform to determine the number of records to scan, corresponding with applications of threaded conversation; each scan is for the posts in a given thread	Was run for HBase, Cassandra, and PNUTS, with ranges up to 100 records

Table 3-1: The Five Workloads in YCSB

Scalability tests were configured and run on clusters that had different number of servers, varying the size of workload proportionally. Six server-class computers and multiple multicore computers to run up to 500 threads of the YCSB client program. The database used consisted of 120 million 1-KB records for a total size of 120 GB of data. Read operations retrieved an entire record; update operations modified one of the fields.

The YCSB++ is an extension of YCSB that provides multiclient-generated workloads, coordinated by ZooKeeper server to follow interaction patterns and experimental strategies to investigate advanced features of table stores [26]. This helps emulate usage scenarios of multiple coordinated operations and provide realistic workloads on the tables. Some of these scenarios include multiple phases of table operations by different clients. The five advanced features examined were weak consistency, bulk insertions, table presplitting, server-side filtering, and fine-grained access control. The tests were run for HBase and Accumulo and exposed functional and performance characteristics not exposed in a systematic way before.

The weak consistency test measures the time lag from one client completing an insert until a different client can successfully observe the value. Table presplitting helps reduce ingest time. To add presplitting, it uses a new workload property. Server-side filtering offloads compute cycles from the client to the server and is expected to reduce network overhead. Four different server-side filters have been defined. For fine-grained security, it supports tests that specify credentials for each operation and access control list (ACL) to be attached to schemas or cells.

The performance of server-side programming and the cost of cell-level access control in Accumulo were tested [26]. Worst-case performance estimation was obtained by using a unique ACL for each key and a larger size for the ACL. Large ACLs were generated in the iterators on the server side, so network overhead between the client and tserver related to the size of the ACL was factored out of measurements. Two workloads were used: an insert workload of 48 million single-cell row inserts into an empty table and a scan workload of 320 million rows scans. One client that had 100 threads and six clients that had 16 threads were used for a six-node cluster. When the client with 100 threads was used, insert throughput decreased while increasing the number of entries in each ACL. When using only six 16-thread clients, the throughput decrease was much less significant compared to using no ACLs, even with a large number of entries for the ACLs.

Multiphase experiments involving multiple clients and table presplitting to avoid online splitting overhead during continuous ingest were conducted. Six million rows of an Accumulo table were split into various numbers of partitions, and completion times were found. At a certain split size in this experiment, Accumulo carried out no further splitting automatically during ingest.

Similar experiments were conducted for bulk insertion using Hadoop and MapReduce, where iterative ingest was replaced by a MapReduce job generated files in the native format used by the table store followed by bulk import of these files into a table. This experiment measures how the policies of managing splits and compaction affect high-speed ingest [26].

The advent of YCSB and YCSB++ has provided a standard, systematic viewpoint to benchmarking scalable table stores. YCSB provides a base benchmarking platform with an extensible framework. YCSB++ demonstrates the use of YCSB to benchmark advanced features. YCSB++ has developed the adapters for Accumulo, which were not provided in YCSB, and also created a client-coordination framework. It provides benchmarks for testing advanced features of HBase and Accumulo. Both tools have used a relatively small cluster of six nodes in their demonstration test

4 THE APACHE ACCUMULO CONTINUOUS TEST SUITE

The Apache Accumulo Continuous Test Suite is used to stress-test the Accumulo table store and its support system. Multiple clients can run the tests in the suite continuously to produce high volumes of writes, reads, scans, and batch scan operations. The test suite includes the ability to perform offline scans, failure recovery scenarios via dynamic simulation of master and tablet server failures, and high-speed parallel table verification using Hadoop MapReduce.

Overall, the continuous test suite provides basic ingest, query, and search workloads that involve multiple index-forwarding operations on a graph. The primary test included in the suite is the Continuous Ingest test that builds a set of linked lists. These linked lists contain elements that are Accumulo table rows with random row IDs. Each row is a key–value pair, where the value is a link or reference to a row ID in the same table. The Continuous Ingest test generates random row IDs and in the process creates a random graph index structure. Other tests, such as the continuous walker and the continuous batch walker, traverse the graph sequentially and in parallel. The tools in the suite can also verify failure recovery as servers are randomly stopped and restarted and provides MapReduce code for data verification.

4.1 The Continuous Ingest Test

The Continuous Ingest test creates a table and adds rows with random row IDs to build a large graph data structure. Each row of the table represents a graph vertex. Each row's value references another row and represents an edge of the graph. The test uses a three-stage iterative method to build a long linked list containing elements connected in this way.

In the first step, a fixed number (n_1) of rows $\{x_1, x_2, \ldots x_{n_1}\}$ is created, with keys (row IDs and columns) generated randomly. The value of these nodes is initially set to null.

In the iteration step, for each row (x_i,) where, $1 \le i \le \ n_1$, a fixed number (n_2) of new rows, $\{y_{i1}, y_{i2}, \ldots y_{in_2}\}$, is created with randomly generated keys. Each row (y_{ij}), $1 \ \le j \le n_2$, is given a value such that y_{i1} links to x_1, y_{i2} links to y_{i1}, …. and, y_{in_2} links to y_{in_2-1}.

In the last step, additional n_1-1 links from each of x_{i+1} to y_{in_2} are added.

These steps can be repeated continuously and over time. A given row may accumulate multiple columns, each referencing other rows. The schematic view of the data structure developed is shown in Figure 4-1, where each box represents a row ID or a vertex in the graph and an arrow corresponds to a link in a row ID, called *previous row,* or a directed edge of the graph. At the end, $x_1$has a value of null and is called the *header node* of the list.

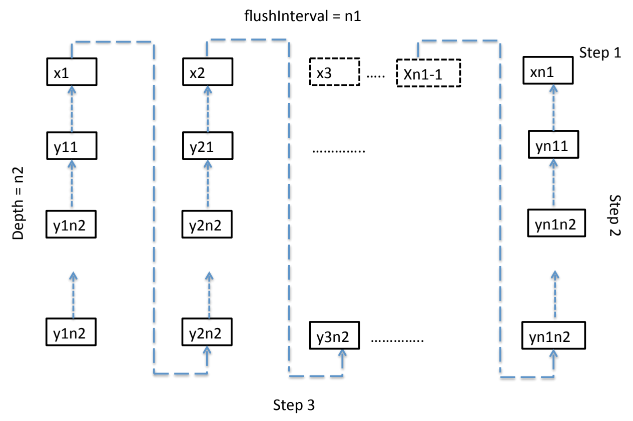

Figure 4-1: Continuous Ingest Data Generation Process

The variable x_i corresponds to the row ID of the table rows and is generated randomly from a given range. The test also creates a column family and column qualifier randomly from a range of numbers. Note that the random generation of row IDs results in random references to rows in general, resulting in the formation of a randomly structured graph.

The size of the table created and the graph density are determined by the number of ingest clients used in the test, the range of random values assigned to the row ID, and the column values. The values of n_1 and n_2 are given by the test parameters *flush interval* and *depth,* respectively, the default values of which are 1,000,000 and 25, respectively. The total number of steps is calculated as $n_1n_2 + n_1 - 1$. It is repeated until a given test parameter value is reached. After each iteration step is complete, the rows are flushed to the tablet server persistent storage device. Other system parameters can be used to modify the behavior of the Continuous Ingest test. These add the ability to control the batch size of the buffer that the test uses, the maximum latency or hold time before memory data is written to the tablet server, and the maximum number of ingest threads to use for each ingest process.

An example of two rows generated by the Continuous Ingest test is shown below. The first 16 digits correspond to the row ID. The column family and column qualifier follow, separated by a colon (:). Next, an empty visibility is contained within the pair of brackets ([]). The row timestamp is not shown. After the brackets, the value field appears, which is split into four parts separated by colons. The third part is a row ID that references another row in the table. In the example shown, the row with an ID of 0020efc13153756f is pointing to the row 11af847f1e67c681. The row with row ID 11af847f1e67c681 has an empty value for its link to the previous row, which means that it is a header node in the linked list.

```
11af847f1e67c681 19a2:6a28 []    277cbce9-26ae-467e-a553
1684bf29db02:0000000000119492:7cd8a9ec
......
0020efc13153756f 19a2:6a28 []    277cbce9-26ae-467e-a553
1684bf29db02:0000000000119492:11af847f1e98c681:7cd8a9ec
```

The value field is the concatenation of a universally unique identifier for the test, the number of rows created so far (in this case, 119,492), and the row ID of the previous row. This provides the link to the previous row, which is an edge of the graph being created. The final part of the value is the checksum for the entry. The size of the value field is 54 bytes for a non-null valued link and no checksum field.

4.2 The Continuous Walker Test

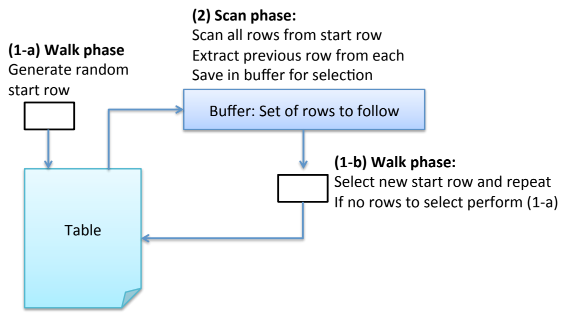
(2) Scan phase:
Scan all rows from start row
Extract previous row from each
Save in buffer for selection
(1-a) Walk phase
Generate random start row
Buffer: Set of rows to follow
(1-b) Walk phase:
Select new start row and repeat
If no rows to select perform (1-a)
Table

Figure 4-2: The Continuous Walker Test

The Continuous Walker test traverses the graph that the Continuous Ingest test created. This process has two phases: scan and walk. In the scan phase, the test selects a starting row (or graph vertex) and collects all of the rows that that row references by using an Accumulo Scanner. In the walk phase, it randomly selects one of the saved rows and uses it as the starting row for the next iteration of the scan. In this way, a *walk* is a move from the randomly selected current row of the table to the row that the value field references. In subsequent scan phases, additional row references are added to the set collected in earlier scan executions. This process is repeated until all of the saved value fields are processed. This process is shown in **Error! Reference source not found.**.

Each new iteration cycle starts after all of the elements in the set of values have been picked. The time delay between the cycles and iteration can be modified to control the speed of the walk. These time delays are used to represent simulated computational load for analytics processing associated with the graph traversal steps. This iterative process results in a walk of the graph, beginning with randomly choosing a graph vertex and walking to other random vertices connected by edges.

4.3 The Continuous Batch Walker Test

The Batch Walker test is a parallel version of the Walker test. It employs the same scan and walk technique that the Walker test uses, but in this test, the walk phase is a set of parallel walks. This test uses the Accumulo BatchScanner interface [32] to perform parallel walks in multiple ranges of the rows in the table. An Accumulo BatchScanner instance is created to access the table. At that time, it is initialized, with the number of threads to use for parallel batch scanning. First, a set of row IDs is selected randomly. An array of Accumulo Range instances is then created using the Accumulo Ranges API, starting at each row ID in the set. The setRanges method is used to initialize the BatchScanner instance with the array of Ranges. The different threads of the BatchScanner use different Range instances from this array for scanning. This process is repeated until the row reference from the value of the current row being walked points to a row with a null value. Then, the test is repeated, with the creation of a new set of batches and ranges.

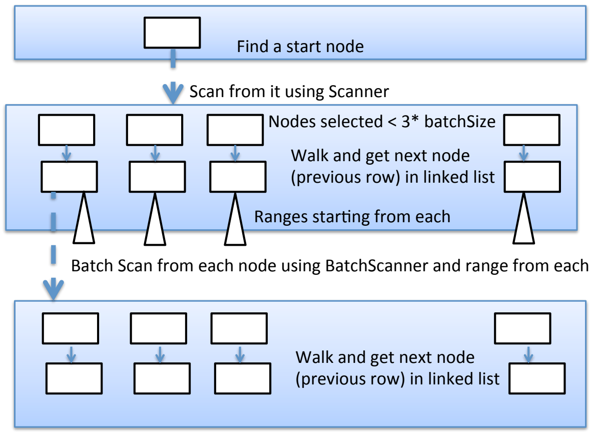
Find a start node
Scan from it using Scanner
Nodes selected < 3* batchSize
Walk and get next node (previous row) in linked list
Ranges starting from each
Batch Scan from each node using BatchScanner and range from each
Walk and get next node (previous row) in linked list

Figure 4-3: The Continuous Batch Walker Test

Similar to the Continuous Walker, the Batch Walker test uses sleep times to create temporal order among operations. The batch size and query thread parameters are configured when invoking the test. The set or row IDs used to create the array of Ranges is limited to three times the batch size value. The query-threads parameter is used when creating the BatchScanner for the table. The default batch size and query threads are 10,000 bytes and 16 bytes, respectively.

The variable ‘count’ maintains the number of elements in the set (rowsToSelect) from which the starting row is selected. It uses another set (rowsSeen) to track which rows have been seen so far.

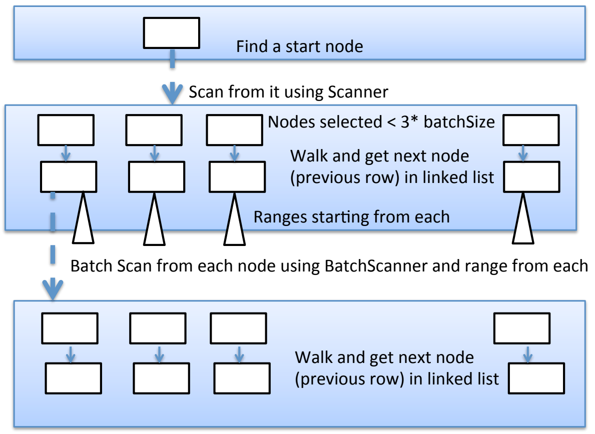
Find a start node
Scan from it using Scanner
Nodes selected < 3* batchSize
Walk and get next node (previous row) in linked list
Ranges starting from each
Batch Scan from each node using BatchScanner and range from each
Walk and get next node (previous row) in linked list

Figure 4-3 is a schematic diagram showing the process. Each walk runs in parallel with others running on multiple threads.

4.4 Graph Construction and Traversal

The Continuous Ingest test builds a random index structure, which in general is representative of a collection of graphs. A row ID in the table corresponds to a vertex in the graph data structure. The row ID is randomly generated from the range 0 to maximum Java *long* value. The column family and qualifier are also randomly generated from the range 0 to maximum Java *int* value. One can configure these ranges to generate graphs of different densities and characteristics. A key–value pair represents a graph edge from the vertex corresponding to the row ID to the vertex corresponding to the row ID that the value field references. The column qualifier is the label of the edge.

Figure 4-4 shows the Ingest, Walker, and Batch Walker tests as graph operations. Each vertex represent a table row, and an arrow represents a write in the Ingest test and a read or query in the Walker and Batch Walker tests, respectively.

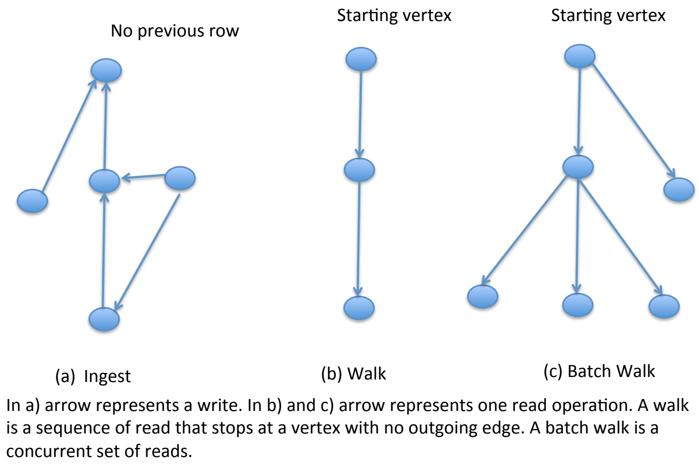
No previous row
Starting vertex
Starting vertex
(a) Ingest
(b) Walk
(c) Batch Walk
In a) arrow represents a write. In b) and c) arrow represents one read operation. A walk is a sequence of read that stops at a vertex with no outgoing edge. A batch walk is a concurrent set of reads.

Figure 4-4: Graph View of the Ingest, Walk, and Batch Walk Tests

In the case of the batch walk, multiple arrows are traversed at the same time on multiple query threads. Note that the walk selects a starting vertex randomly and stops a walk when it reaches a vertex that has no outgoing arrow.

The Continuous Ingest and Walker tests can serve as benchmarks for graph construction and traversal-based applications. In the current version of the test, only random row ID, column family, and qualifiers are generated. These can be extended to weighted value generation, which may be more representative of a specific application domain. In our study, we have used the current version of the test.

4.5 Cell-Level Security

The test for estimating operation overhead for checking cell-level authorization (ACL) is implemented as an option available in Ingest and Walker tests. The overhead simulated in the test is based on the random selection of different authorization properties. The visibilities, authorizations, and authorization symbols are defined in separate files and flags provided as test configuration parameters. The visibilities are randomly chosen during record storage in an ingest operation and verified in a walk operation. The visibilities are given as arbitrary symbols in individual lines of the visibility file that represents authorized operator.

4.6 The MapReduce Table Verification Test

The Verification test is a MapReduce job that checks the integrity of the table that the Continuous Ingest test creates. It verifies that all referenced nodes are defined to ensure that no data was lost during the course of the Ingest test. For each row of the table, it outputs key–value pairs for the reduce method to process. In this the key is the row ID of the previous row, obtained from the value field of the current row and the value is the row itself. The previous row is the end vertex of the edge from the vertex corresponding to the row. The reduce method gets the list of all rows that has the same previous row. In graph terms, this is the list of all vertices that has the same end vertex, as shown schematically in Figure 4-5.

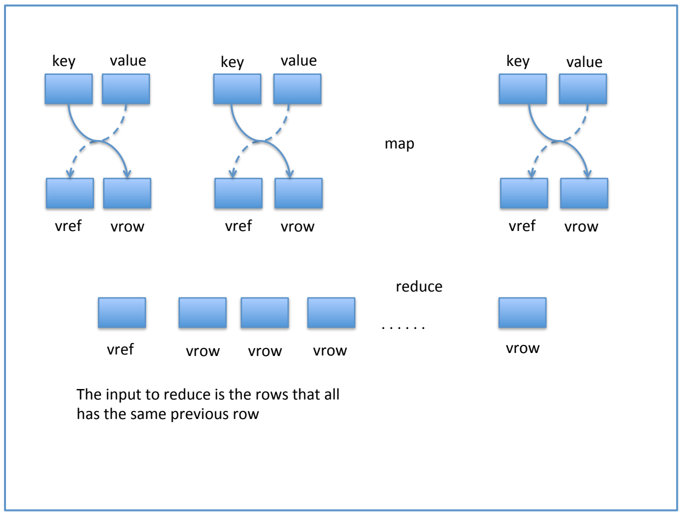

Figure 4-5: MapReduce Table Verification

The MapReduce job input class is the AccumuloInputFormat class from the Accumulo client library. A set of row ranges, based on the number of mappers to be used, is obtained and assigned to the AccumuloInputFormat using the setRanges method of this class [32]. The mapper input key and value are the Accumulo Key and Value types. The mapper output classes for the key and value are the LongWritable and the VLongWritable classes of the Hadoop library [31].

The map method gets the row ID from the key. It then calls the validate method with the key, and the corresponding value reads and reports checksum errors. The row value field is then used to obtain the previous row, outputting it as a key, while the row itself is outputted as the value of the mapper output stream, as illustrated by the arrows in Figure 8.

The reducer reads the key values from the mapper output as a key–value pair: vref, (collection of vrows). The vref is a row ID reference, and the value associated with it is the collection containing all vrows that corresponds with the row IDs and has the row ID referenced by vref as the previous row or end vertex. The reduce method reads the values in the collection, checks whether the value is legitimate (a row is undefined if it has a inappropriate content in the value field), and counts them. If the collection is not empty, then all these elements are added to a string and outputted as a value for the reduce key. The output of the reduce method identifies the rows that are undefined, unreferenced, or are proper and referenced.

4.7 The Stats Collector

The stats collector gathers data on Accumulo metrics that include resource usage, file system, and MapReduce and table statistics. These metrics are gathered every 30 seconds and written to a log throughout the lifetime of the tests.

The getACUStats method returns the number of tablet servers, the total table entries (all), the ingest rate (all), the query rate (all), records in table (test), records in table in memory (test), the ingest rate (test), the query rate (test), tablets used (test), and tablets online (test). The “all” and “test” in parentheses are all tables and the test table, respectively. The getFSStats method collects information about the Hadoop file system usage associated with the Accumulo instance and the table being tested. It returns the size of the accumulo directory in HDFS that corresponds with an Accumulo instance, its directory count, the number of files, the test table directory length, and file counts. The getMRStats method returns the number of map tasks, max map tasks, reduce tasks, max reduce tasks, number of task trackers, and the count of blacklisted task trackers. The getTabletStats method returns the minimum, maximum, average, and standard deviation from the metadata table. All these values are printed in the statistics log file

5 APACHE ACCUMULO BENCHMARK

We aim to highlight the following aspects of performance in the benchmark tests:

- Throughput and latency
- Scalability
- Sustained performance.

Sustained performance is an important factor often needed in applications that continuously process online data in semi–real-time mode. We wanted workloads that represent operations on multi-indexed data structure or graphs.

5.1 Test Execution Plan

The Accumulo continuous test suite uses a multi-indexed data structure as the test table data. The Continuous Ingest, Walker, and Batch Walker tests perform long-term ingest or write operations as well as traversal or walk of the indexed data structure records in random, sequential, and multithreaded batch operations. These tests measure the throughput and latency aspects. The scalability issue can also be examined by running these tests in clusters of different numbers of tablet servers. These tests are designed to run continuously over a long period of time and are suited to testing sustained performance. In view of these factors, we used the Accumulo continuous test suit in our benchmark tests. We also used two different target cluster environments: a high-performance analytics workbench environment and a commodity cloud.

5.2 Performance and Scalability

We observed ingest and query rates for both configurations to understand the relationship between write load and read latency. We hosted Ingest as well as Walker test clients on the same host. In addition, we run ingest and walker clients independently, as in the Ingest Only and Walk/Batch Walk Only tests to observe maximum and average ingest rates and query rates.

The performance measures in YCSB are given by read latency for different levels of update and write loads on the table store under test. In our benchmark we simulate this scenario via the Partial Ingest Full Walk/Batch Walk configuration (see Section 5.3). The Full denotes involving N test clients, where N is the number of tablet servers, while the Partial limits this to around N/3.

To measure scalability, we track the change average ingest and query rates as we increase the number of computers and the ingest and query load increases proportionately. In a good, scalable system, both ingest and query rates remain constant.

5.3 Sustained Performance

We wanted to examine the long-running behavior of Accumulo. In this context, we examined the role of the table split threshold of the Accumulo table. This parameter (by default, 1 GB) is the combined size of the files in a tablet that triggers a tablet to be split. A low split value is good, as parallelism in table operation increases as more tablets are created and distributed in a balanced manner across the tablet servers. However, in the case of a long-term test scenario, with the increasing size of the table, the file compaction overhead increases with a low split threshold value. This becomes a major bottleneck in maintaining a sustained level of table operations.

We have examined the role of table split threshold in maintaining parallelism in the Ingest Only test. We start with the default threshold. After a moderate number of tablets have been generated and evenly distributed across the cluster, we stopped the test and increased the configured value of the split threshold of the test table to a high value, and then restarted the Ingest Only test.

5.4 Benchmark Workload

Each test generates a basic workload. The Continuous Ingest test, described earlier, generates random row IDs and values that link to previously generated rows. It starts with a set of randomly generated flushInterval row IDs and builds linked lists of length maxDepth that iteratively link back to the previously generated set. The flushInterval and maxDepth are defined in Section 4.0. We have used 1,000,000 and 25 for these two parameters, respectively. The column family and column qualifiers are randomly generated. The row ID is 16 bytes; the column family and column qualifier are 8 bytes in size. The value field is 54 bytes, with no checksum and a non-null previous link.

The Walker test randomly selects a first row to read using a scanner and continuously scans a randomly chosen previous row to which it links. It repeats this operation after a certain wait time. The Batch Walker test uses a BatchScanner with a given number of threads to perform a parallel scan, repeating with a certain wait time. The wait time, number of threads to use, and a batch size giving a bound on the number of queries to perform are provided as external parameters. The value used for wait time for the Walker is 10 milliseconds. The values for wait time, number of threads, and batch size used for the Batch Walker are 180 seconds, 16, and 10,000, respectively. These are the default values defined in continuous-env.sh, which resides in the continuous test directory in the Accumulo distribution.

We conducted the tests in clusters of sizes ranging from 10 to 1000 nodes in two test environments: a cluster of physical machines connected via a high-speed network a commodity cluster composed based on VMs. We ran the tests individually as well as in combinations. The individual tests include Ingest Only, Walk Only, and Batch Walk Only. The combination tests are Full-Ingest-Full-Walk, Full-Ingest-Full-Batch Walk, Partial-Ingest-Full-Walk, and Partial-Ingest-Full-Batch Walk. In the Partial-Ingest-Full-Walk/Batch Walk and the Full-Ingest-Full-Walk/Batch Walk tests, we simultaneously used Ingest and Walker tests. In the Partial-Ingest-Full-Walk test, we limited ingest to a fraction of *N* Ingest test clients and *N* Walker test clients. In the Full-Ingest-Full-Walk, we used all *N* ingest and walker clients to ensure that proportionate load is offered as the cluster size changes.

To create sufficient ingest and query load, multiple concurrent test clients were employed. Each client was configured with a large send buffer and high number of threads. The clients ran on the same computers as the Accumulo tablet servers. In general, for an *N* server Accumulo configuration, we had *N* clients performing the test operations, but in some test configurations, multiple test clients were run per tablet server. By default, each client used a 100,000,000-byte buffer and 4 threads. This is modified in certain test cases to higher buffer size and number of threads to accommodate higher operation throughputs. If the test client generates 10^9 operations, which is the default value, it will build a table up to 10^9N entries. We change this default value to a high number when we run a sustained Ingest test that builds a much larger table. In a Walk-Only or Batch Walk test, *N* test clients continuously walk the indexed structure that the Ingest-Only test built earlier.

We used the Accumulo monitor to observe high-level behavior and collected performance statistics using the continuous-stats program. We conducted high-speed parallel data verification using the Accumulo verify MapReduce job.

We used the Accumulo continuous stats collector to collect the results of the tests. The stats collector provides the time of test, size of the table at start and end, write and read throughput, and number of tablets used. A high read throughput amounts to a low latency. The test logs capture the counts of paths traversed. When test results are compared, we maintain their running times to be close and use the average value of throughputs. We have also used the MapReduce-based Verify test as a benchmark for the table verification process. We have not used other tests in the Accumulo continuous test suite, such as the Agitate and Scanner tests, as our main objectives are realized by the Ingest and Walker tests.

5.5 Test Environments

We ran the benchmarks in two cluster environments. The first environment was the AWB. The AWB is a cluster consisting of high-performance computers and network infrastructure. The second environment we tested was Amazon EC2. Amazon EC2 is a public cloud environment in which we used low-cost virtual servers and shared networking. The operating system, Java Virtual Machine, Hadoop, and Accumulo settings used when performing these benchmarks are described in Appendices A and B.

5.6 Configuration Notes

Accumulo and other configuration settings used are detailed below. These settings were based on discussions with the developers. The operating system parameter vm.swappiness was assigned 0, the maximum file descriptors—fs.file-max—was assigned to 65535, and the file system was mounted with noatime,nodiratime options.

The ZooKeeper configuration had max client connections as 1,000. We used five and three ZooKeeper instances for the AWB and Amazon EC2 clusters, respectively. In most cases, three ZooKeeper instances were considered sufficient for fault tolerance. Adding additional ZooKeeper servers to a cluster slows write rates but increases read rates.

As given in the directory of the distribution, we used the Accumulo configuration $ACCUMULO_HOME/conf/examples/3GB, with the following modification for some of the optional items to suit the cluster architecture in the AWB cluster. In the Amazon EC2 cluster, no modifications were made except for the maximum client connections value mentioned above for ZooKeeper servers.

The Accumulo Tablet Server (commonly referred to as the *tserver*) was deployed on each worker node alongside a Hadoop datanode, tasktracker, and Accumulo logger. We assigned Java heap for the tserver to the maximum 6 GB, with a stack size of 256 KB and new generation size of 1 GB. The tserver.memory.maps.max was assigned 6 GB in the Accumulo configuration file, accumulo-site.xml, as native memory outside of the Java heap. Of the Java heap, 4.5 GB is reserved for index and data caches, configured by assigning the tablet server data cache to 512 MB by configuring the tserver.cache.data.size parameter to 512 MB. The tablet server index cache was assigned 4 GB by configuring the tserver.index.data.size parameter to 4 GB.

We configured the Write Ahead log (using logger.dir.walog) to use one local drive and the remaining 11 drives (in the AWB cluster nodes) to be used by HDFS. The minor and major compaction threads were configured to be six each (using tserver.compaction.major.concurrent.max and tserver.compaction.minor.concurrent.max parameters). The tablet server block size was assigned 256 MB (using table.file.blocksize).

The continuous tests are run via shell scripts in the test/system/continuous directory. The configuration is controlled via continuous-env.sh. We set the number of ingest tests to run per client to 1,000,000,000. The generation of checksums is turned off. We used the default values for maximum random row, column family, and column qualifiers. We also used default values for maximum buffer size and number of ingest threads. For the Walker tests, the default sleep times were used. For the Batch Walker test, we used the default values for sleep time, batch size, and threads. To run the Cell-Level Security test, we defined the HDFS vis.txt and auths.txt files. The parameters VISIBILITIES, AUTHS, and VERIFY_AUTHS were assigned accordingly. To configure the Verify Table test using MapReduce, we used 200 threads each for the map and reduce tasks. For the Cell-Level Security test, we used A, B, C, D, and E as authorization items and the combinations {A, B, C, D, E, A&B, A&C, A&D, A&E, A|B, A|C, A|D, A|E, A&B&C, A&B&D, A&B&E, A|B|C, A|B|D, and A|B|E} in the visibility file, and {A,B,C,D,E, {A,B,C,D,E}} in the authorization file. These are assigned to the AUTHS, VISIBILITIES, and VERIFY_AUTHS parameters, respectively, in continuous-env.sh.

6 BENCHMARK RESULTS FROM THE EMC^2 ANALYTICS WORKBENCH

6.1 Continuous Test Results

We ran the Accumulo continuous tests in the EMC^2 AWB cluster using a three cluster sizes: 300 nodes, 500 nodes, and 1000 nodes. Each cluster used eight servers to provide core services such as the Accumulo master and related processes, the Hadoop Namenode and Job Tracker, and five ZooKeeper servers. As a result, the actual number of Accumulo tablets was 287, 488, and 984, respectively. As expected in large-scale computing environments, there were cases where servers were simply inoperative because of hardware failures. As a result, fewer tablet servers were used for some tests. The test clients were also run on the same computers that ran the tablet servers. Each test client was configured to perform 1 billion operations (ingest/query) using the configurations described in Section 5.4. For some tests, we ran multiple test clients per server. As we present these results, we use M, B, and T to represent 10^6 $(millions)$, $10^9(billions)$, and $10^{12}(trillions)$. We also use GB to represent *gigabytes,* or 10^9 bytes.

6.1.1 Performance Tests

We begin by presenting results of the performance evaluation tests that examine ingest and query throughput in isolation. These results are based on observations from running the Ingest Only, Walk Only, and Batch Walk Only tests described in Section 5.2. Following this, we present results from tests that examine the performance of multiple concurrent ingest and query operations—the Partial-Ingest-Full-Walk/Batch Walk and the Full-Ingest-Full-Walk/Batch Walk tests. The latter tests show results with simultaneous ingest and scan operations.

Ingest Only Tests

An important factor in Accumulo ingest performance is the overhead of tablet splitting activity. A tablet split occurs when the size of the tablet exceeds the table split threshold. The split results in new tablets that the load balancer of the master server evenly distributes across the cluster. These additional tablets allow greater parallelism, which facilitates high throughput ingest. Table splitting quickly becomes a key factor in Continuous Ingest test runs that begin with an empty table.

The uniform distribution of the keys in the Continuous Ingest test tends to amplify the impact of table splitting in the context of these tests. The Continuous Ingest test generates uniformly random row IDs for the rows to be ingested, making ingest to each tablet in the cluster equally probable, and all tablets grow at a similar rate. Unfortunately, this means that all tablets are likely to be split at the same time, swamping the administrative nodes with split operations.

To understand performance outside of the context of tablet splitting, we used a presplit table for ingest to determine the best-case performance possible. We also used tables that were not presplit to estimate the worst-case performance. In real-world scenarios, with non-uniform key distributions, performance is expected to fall between these boundary cases.

Continuous Ingest with Presplit Tables

We performed several Ingest test runs on presplit tables that were initially empty. To defeat splitting at ingest time, we changed the tablet-split threshold to an arbitrarily large number (128 GB in each case). Each tablet server hosted three ingest clients, each configured to use 1 GB of buffer and 16 threads. A test client generated 10^9 entries.

Table 3-1 provides the results of these runs. There were 287, 488, and 980 tablet servers overall for each and 30, 40, and 32 tablet servers per machine in the 300, 500, and 1000 node clusters, respectively. We stopped after the table size was $10^9 \times N$, where N is the total tablet servers.

Cluster Size	Tablets per Tablet Server	Initial Table Size (Entries)	Final Table Size (Entries)	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
300	30	0	287B	2.3	42M	37M
500	40	0	488B	3.1	62M	46M
1,000	32	0	1T	2.6	165M	108M

Table 6-1: Continuous Ingest into Presplit Tables (AWB)

We ran Ingest tests with various split sizes. The values used in Table 2 represent our best ingest performance.

Sustained Continuous Ingest with Presplit Tables

Table 6-2 presents Ingest test results for a presplit table for 26 hours in a 1,000-node cluster that had 931 tablet servers. There were three test clients per server, each with 1 GB of buffer and 16 threads. To avoid splitting the table, we used a split threshold of 128 GB. We started the test with 942 tablet servers. During the course of the test, 11 tablet servers went down, but the Continuous Ingest test went on without any interruption.

Cluster Size	Tablets per Tablet Server	Initial Table Size (Entries)	Final Table Size (Entries)	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
1,000	33–34	0	7.56T	26	173M	94M

Table 6-2: Sustained Continuous Ingest into Presplit Tables (AWB)

Continuous Ingest Without Presplit Tables

As described previously, we observe considerable overhead as tablets split. Splits occur naturally as tables grow: Initially, one tablet is assigned to the empty table, which is hosted on one tablet server. The table split threshold parameter controls the table split operation. The default table split value is 1 GB. As ingest to the table takes place and the table split threshold is reached, the tablet is split into two tablets. This process continues as more ingests occurs. A higher-value table split threshold reduces the frequency of the tablet splits.

In exploring the effect of the table split threshold on ingest throughput, we ran the ingest test in a 300-node cluster that had 287 test clients (each on a tablet server) and an initially empty table having the default table split threshold of 1 GB. We also ran the test on table with 211-GB entries and a table split threshold value of 128 GB. Although the maximum ingest throughput was the same, the average ingest rate was considerably better, as shown in Table 6-3

The throughput profiles for these ingest tests are given in Figure 6-1 for 1 GB and in Figure 6-2 for 128 GB. The *x*-axis is the time (counts of half-minutes), and the *y*-axis is the write throughput. We noted a more sustained ingest throughput for the second case, resulting in a higher average value for ingest throughput (Figure 6-2).

Test	Table Split Threshold	Initial Table Size	Table Size (Millions of Entries)	Test Period (Hours)	Maximum Ingest Throughput	Average Ingest Throughput
Ingest Only	1 GB	0	89.98B	8.145	26,591K	3,270K
Ingest Only	128 GB	211B	488B	6.65	27,266K	11,003K

Table 6-3: Continuous Ingest into Non-Presplit Tables (AWB)

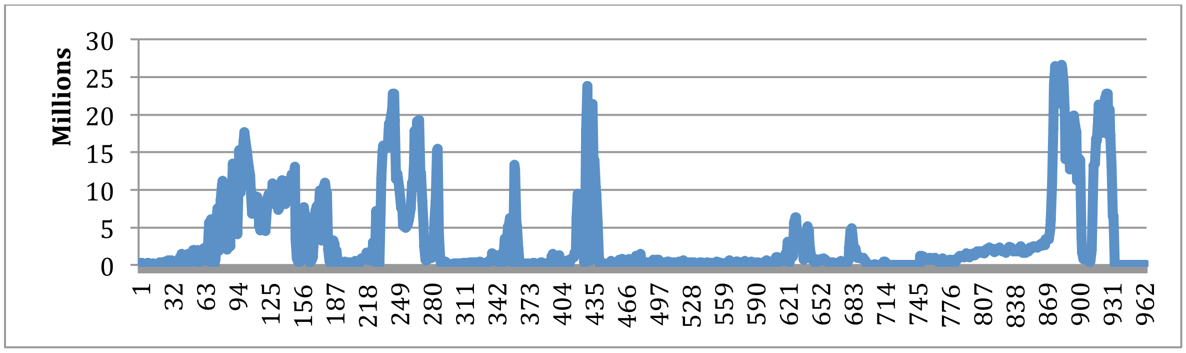

Figure 6-1: Ingest Throughput Profile (1-GB Split Size)

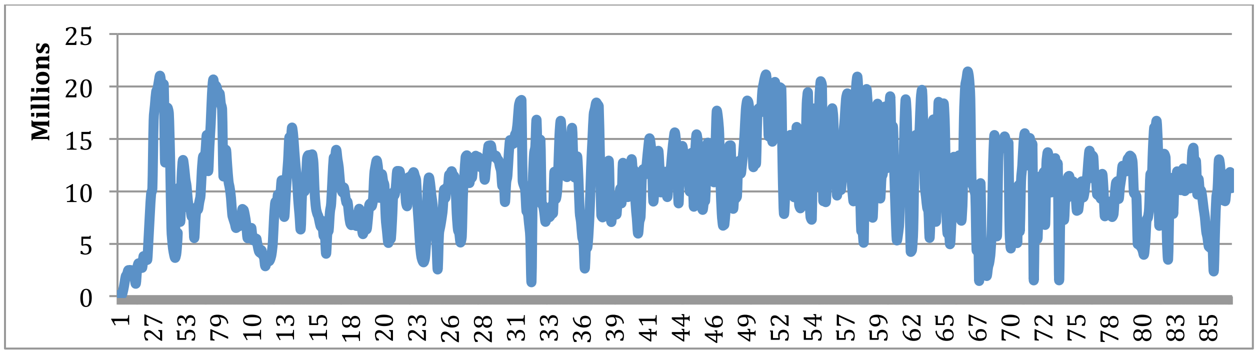

Figure 6-2: Ingest Throughput Profile (128-GB Split Size)

We also ran several tests to evaluate ingest throughput without presplitting on clusters of different sizes. There were 287, 488, and 980 Continuous Ingest test clients for the 300-, 500-, and 1,000-node environments, respectively. Each ingest client used a 100-MB buffer and 4 threads. Different table split threshold values were used. The results are shown in Table 6-4. Notice the higher average throughput rates with higher split threshold values, shown in bold for the 500- and 1,000-node clusters. The rates are within 5 percent of one another.

We investigated the performance of ingest throughput with different initial tablet distributions (table sizes). We conducted a multiphase Ingest test with a fixed table-split threshold of 4 GB. Each test client ran with a 100-MB buffer and 4 threads. We began with 100 concurrent ingest clients. In the second phase of the test, we ran the test using 250 clients on the table that was previously ingested by the 100 clients to a size of 96.74 billion entries in the first phase. In the third phase of the test, we ran the test using 500 concurrent clients (two clients per server) to ingest into the table with 193.64 billion entries that had earlier been ingested in phases 1 and 2. The result is shown in Table 6-5.

Cluster Size	Tablet Split Threshold	Initial Table Size	Final Table Size	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
300	1 GB	4.69M	90B	5.3	9.75M	5.13M
500	1 GB	3.9M	102B	15.9	12.47M	1.86M
	10 GB	102B	449B	7.1	35.65M	16.2M
	128 GB	452B	559B	1.9	42.88M	**18.35M**
1,000	1 GB	91B	314B	24.4	22.51M	3.6M
	128 GB	430B	1025B	10.3	47.59M	**17.56M**

Table 6-4: Continuous Ingest into Non-Presplit Tables, Varying Split Thresholds (AWB)

Table Size Start–End (Entries)	Period (Hours)	Max Ingest (Operations per Second)	Average Ingest (Operations per Second)
4.6M–96.74B	6.275	9.75M	5.13M
96.74B–193.63B	2.7	26.92M	10.41M
193.63B–356.5B	13.55	38.65M	3.45M

Table 6-5: Continuous Ingest on Non-Presplit Tables, Varying Initial Table Size (AWB)

The results show that with increasing table size and higher ingest load, the maximum ingest throughputs achieved were increasingly higher. However, we observed longer periods of low ingest rates, as there was increased table splits, file compaction, and garbage collection activities. If we allow sufficient tablet distribution, and then increase the split threshold value of the table, we can generally expect to observe higher average throughput.

We have observed that tablet splitting is the single most significant factor in Accumulo ingest performance, with presplit tables able to sustain ingest rates that are 5 times higher than tables that are not presplit. Presplit tables support greater initial parallelism and reduce data management overhead. In the context of many real-world applications, presplitting tables is entirely possible, because the data involved contains natural partitions. In other cases, non-uniform key distributions lessen the effect of split "storms" as a result of concurrent split activity. In the end, splitting in Accumulo can be controlled manually by reconfiguring the threshold parameter to achieve the best performance of any application.

Walker, Batch Walker, and Mixed Tests

The Walker and Batch Walker tests were run to examine scan latencies in different configurations. After this, we ran mixed tests involving the Ingest and Walker tests. In the mixed tests, we had partial and full Ingest tests, where $\frac{\boldsymbol{N}}{\mathbf{3}}$ and $\boldsymbol{N}$ test clients were used (where N tablet servers were available). The tables with the results give the query throughput. A high query rate implies low latency.

Walker and Batch Walker on Presplit Tables

We ran the Walker and Batch Walker tests on a presplit table (33–34 tablets per server). One Walker and one Batch Walker test were used per server, with a 1-GB buffer and 16 threads (Table 6-6). The table split threshold was 128 GB. We expect that the size of the table will influence performance. We did not examine this but had used nearly the same table sizes.

Test	Tablet Servers	Table Size (Entries)	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
300-Node Test					
Walk Only	286	705B	1.0	2,880	2,559
Batch Only	286	705B	0.81	231,913	36,709
500-Node Test					
Walk Only	480	1T	1.46	3,744	3,577
Batch Only	480	1T	1.07	29,988	15,489
1,000-Node Test					
Walk Only	946	1T	1.5	6,197	5,821
Walk Only	938	7.56T	8.3	7,838	3,794
Batch Only	946	1T	0.8	318,624	21,977

Table 6-6: Walker and Batch Walker Tests on Presplit Tables (AWB)

The average latencies for the Walker test for 300-, 500-, and 1,000-node clusters were 0.39, 0.28, and 0.17 milliseconds, respectively. For the Batch Walker test, the average latencies were 27, 64, and 45 microseconds, respectively.

Walker and Batch Walker Tests on Non-Presplit Tables

We ran the Walker Only and Batch Walker Only tests with a 1-GB buffer and 16 threads. We used 1-GB table split thresholds and 128-GB thresholds on relatively large tables. We do not expect these settings to affect query throughput, as no tablet split can occur during this test. The table size is expected to influence performance. Table 6-7 presents results on 300-, 500-, and 1,000-node clusters with 287, 486, and 984 test clients.

The average latencies for the Walker test for 300-, 500-, and 1,000-node clusters were 0.50, 0.29, and 1.40 milliseconds, respectively. For the Batch Walker test, these averages were 32, 95, and 51 microseconds, respectively. The multiple threads employed by the Batch Walker test in combination with cluster size have a direct effect on the order of magnitude of query latency. For smaller tables, we noticed that the throughputs were comparable to the results for presplit tables.

Full and Partial Ingest, Walker, and Batch Walker Tests

The results of Ingest throughput and Walk test in a Full-Ingest-Full-Walk/Batch Walk (287 ingest clients) and Partial-Ingest-Full-Walk/Batch Walk (90, 150, and 300 ingest test clients, respectively) in 300-, 500-, and 1,000-node clusters are given in Table 6-8 and Table 6-9. The results in are based on the observations from running the tests on presplit and general tables, respectively. For the presplit table tests, the table split threshold was 128 GB, and the test client used a 10-GB buffer and 16 threads. For the general table tests, we used the default values of 1 GB, 1 GB, and 4 threads for these parameters.

Test	Table Size (Entries)	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
300-Node Test				
Walk Only	280B	2.777	3,620	1,979.4
Batch Only	280.3B	0.9913	84,041	31,103
500-Node Test				
Walk Only	635.2B	2.16	3,619	3553
Batch Only	635.3B	2.78	157,101	10,454
1,000-Node Test				
Walk Only	314.5B	2.11	2,522	713
Walk Only (128 GB)	1288.6B	0.391	2,403	90
Batch Only	314.5B	0.593	567,392	19,467
Batch Only (128 GB)	1288.6B	5.068	105,302	3,700

Table 6-7: Walk and Batch Walker Tests, Varying Cluster Sizes (AWB)

Ingest Type	Walk Type	Initial Table Size	Final Table Size	Tablet Servers	Test Period (Hours)	Max Ingest (Operations per Second)	Max Query (Operations per Second)	Average Ingest (Operations per Second)	Average Query (Operations per Second)
300 Nodes									
Full	Walk	705B	978B	286	3.5	37.6M	1,404	23M	249
500 Nodes									
Full	Walk	1T	1.18T	478	1.44	49.4M	15,315	19.35M	988
Part	Walk	1.25T	1.38T	478	2.6	21.8	3,054	14.57M	1,478
1,000 Nodes									
Full	Walk	1T	1.25T	946	0.98	102M	5,242	55M	3,734
Full	Walk	7.56T	7.8T	938	3.2	109M	3,400	35M	1,999
Part	Walk	1.25T	1.36T	946	1.1	43M	8,887	29M	6,840

Table 6-8: Mixed Workloads on Presplit Tables (AWB)

Ingest Type	**Walk Type**	**Initial Table Size**	**Final Table Size**	**Test Period** (Hours)	**Max Ingest** (Operations per Second)	**Max Query** (Operations per Second)	**Average Ingest** (Operations per Second)	**Average Query** (Operations per Second)
300 Nodes								
Full	Walk	488B	525B	1.18	20.4M	408	8.8M	9.2
Partial	Walk	605B	626B	1.04	6.8M	231	5.8M	9.34
Full	Batch	525B	603B	2.494	10.8M	82K	9.0M	1,088
Partial	Batch	626B	680B	2.43	6.9M	89K	6.5M	1,244
500 Nodes								
Full	Walk	1580B	1725B	1.97	35.8M	141	25.2M	8.1
Partial	Walk	1731B	1767B	0.86	13.8M	124	12.7M	16.1
1,000 Nodes								
Full	Walk	416B	420B	0.84	7.5M	1,658	1.5M	65
Partial	Walk	424B	428B	0.72	7.7M	1,230	1.5M	70
Partial	Walk	1,289B	1,335B	1.01	16.0M	2,318	13.0M	42

Table 6-9: Mixed Workloads on Non-Presplit Tables (AWB)

As expected, we noted an increase in query throughput in partial ingest cases compared with the full ingest cases. We note a significant drop in query throughput in the case of the non-presplit table, mainly because of the impact of tablet split overhead resulting from the ingest operations. However, we observe marginal improvement in query throughput in case of partial tests.

6.1.2 Scalability Tests

Scalability is the ability to perform operations at the same level of throughput as the cluster size and loads are increased proportionately. We examine the results of performance tests as the cluster size increases. We determine the ingest throughput for different numbers of tablet servers using the results described in Table 6-1, and run tests with loads proportional to cluster sizes. For a cluster with *N* tablet servers, we used the load of ingesting a table of size *N* billion entries, starting from an initially empty table. Figure 6-3 gives the average ingest, query, and batch query rates for presplit tables.

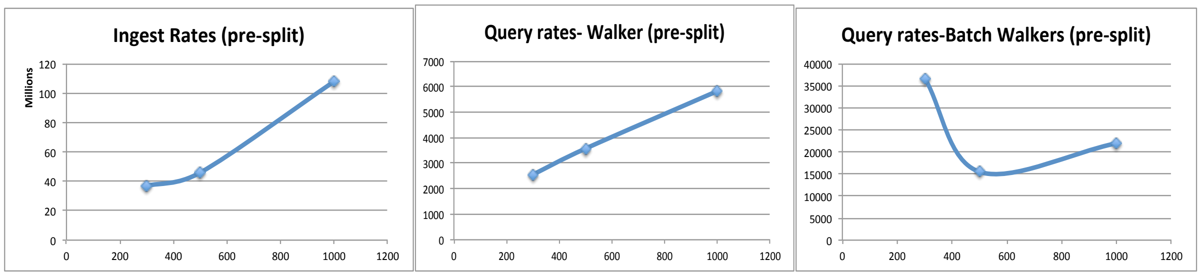

Figure 6-3: Average Throughputs in Presplit Table Clusters of Increasing Size (AWB)

For a non-presplit table, we ran the ingest test for 5.5 to 6 hours on clusters that had 300, 500, and 1,000 nodes (287, 488, and 980 tablet servers and test clients). The table ingested was empty initially. As each test client was ingesting 1 billion rows, we expect 287 billion, 488 billion, and 980 billion table entries to be ingested as the proportionate loads. We did not reach these volumes within the test time window. Instead, we found 54 billion, 49 billion, and 84 billion entries in the table at the time once we stopped each test. Each test client was configured to perform up to 10^9 operations, used a 100-MB buffer, and had 4 threads. Table 6-10 presents the throughputs observed. We also ran the Walker and Batch Walker tests on tables that had 89 billion, 635 billion, and 314 billion entries.

Test	Initial Table Size	Final Table Size	Period (Hours)	Max (Operations per Second)	Average (Operations per Second)
300-Node Test					
Ingest	0	54B	5.5	23,821,770	3,187,710
Walk	89B	89B	2.7	6,239	1,252.34
Batch Walk	89B	89B	0.5	121,796	35,272
500-Node Test					
Ingest	0	49B	5.5	12,346,126	2,069,606
Walk	635B	635B	2.7	3,619	3,549
Batch Walk	635B	635B	2.7	157,101	10,453
1,000-Node Test					
Ingest	0	86B	6.0	17,468,545	4,089,657
Walk	314B	314B	2.1	2,522	713
Batch Walk	314B	314B	0.5	567,392	19,467

Table 6-10: Scalability on Non-Presplit Tables (AWB)

As observed earlier, ingest throughput in Accumulo can be high, reaching a maximum throughput of more than 150 million inserts per second for a presplit table. Average throughput can also be near 100 million inserts per second in this case. High ingest throughputs can be maintained for a presplit table, as well. In the case of non-presplit tables with the highest expected run time system overhead of multiple table splitting at the same time, ingest throughputs can be in the range of 3 million to 20 million inserts per second (on average). We saw an improvement in ingest rates in the non-presplit table after some time when a sufficiently large number of tablets was created and distributed across the cluster.

In the Walker and Batch Walker tests, we found that latencies were generally in the tens of milliseconds and microseconds ranges, respectively, for both presplit and non-presplit tables. In mixed tests, we found more stable performance for the presplit tables, where both the Ingest and Walker tests were running at the same time. For non-presplit tables, the latencies increased as more ingest occurred.

The scalability is almost linear in the case of presplit tables, as more test loads are introduced in proportion to the size of the test cluster. This is not quite so for a non-presplit table.

The performance and scalability figures are generally better than most distributed table stores for which benchmark data is available. The presplit table test provided the best-case scenario, and the non-presplit table case provided the worst-case scenario. In practice, we expect performance to lie within these two extreme cases. Our tests were based on random generation of row references. In a weighted scheme of row reference, we expect better performance for non-presplit tables.

6.1.3 Cell-Level Security Test

To examine the performance cost of cell-level security (ACL), we ran the Ingest test with and without the feature (Table 6-11). This feature is activated via flags along with visibility and authorization text files in the HDFS directory. These files are configured in continuous-env.sh. The results show that the variation in average throughput is within 15 percent of each other.

Continuous Ingest Test	Table Size (Millions of Entries)	Period (Hours)	Maximum Ingest	Average Ingest
500				
No ACL	0–34.5B	4.56	8.2M	2.2M
With ACL	0–33.5B	3.75	9.0M	2.5M
1,000				
No ACL	0–86B	6.03	17.5M	4.1M
With ACL	0–75.5B	6.00	22.6M	3.5M

Table 6-11: Continuous Ingest with Varying Visibilities (AWB)

6.1.4 MapReduce Table Verification Tests

The Table Verification Test uses variables defined in mapred-setup.sh and continuous-env.sh. It launches a MapReduce job to run the verify table program available in the continuous test suite. The elapsed times of this test are given in Table 6-12.

Cluster Size	Table Size (Entries)	Mappers	Reducers	Time
300	1.46B	200	200	41 min, 16 sec
1,000	12.3B	200	200	34 min, 46 sec
1,000	1T	10,000	5,000	17 hours

Table 6-12: MapReduce Table Verification (AWB)

6.2 YCSB Results

We used the five workloads employed in the original YCSB workload packages and performed 10,000 operations. Clusters of 300, 500, and 1,000 nodes were used for the YCSB benchmark test. The default buffer size was 100 KB. We also used a new workload, Workload F, which used an ingest ratio of 1 and was designed to perform ingest (insert/update) with different buffer sizes. We ran a test client with a single thread and 50 threads, respectively, for each workload. The number of records was 10,000 for each workload, as in YCSB. We used buffer sizes of 10 KB, 100 KB, 1 MB, 10 MB, and 100 MB with Workload F. Table 6-13 and Table 6-14 show the throughput and latency values for the tests. The behavior is similar to that reported earlier [26].

We observed that a 100-KB buffer provides a higher ratio of throughput to latency. The throughput and latency are given for each workload and cluster size. The top number is the throughput in operations per second, and the lower number is the latency in milliseconds.

Note that the ingest rates in Table 6-13 are tiny compared with those for the Accumulo Continuous Ingest test. This is because only one ingest client was used in the YCSB test. When multiple threads were used, the single client resulted in higher ingest throughput in general, although these were still not comparable to the continuous ingests tests run with a large number of test clients. The latency figures from these tests are very close to what was obtained by the Accumulo Continuous Walker test.

Nodes	Workload (1 Thread)									
	A	B	C	D	E	F				
						10 KB	100 KB	1 MB	10 MB	100 MB
300	11.20 0.205	5.93 0.033	5.78 172.72	8.28 0.10	5.71 0.11	181.88 5.39	202.24 4.82	539.57 1.65	1559 0.16	1225 0.028
500	-	-	-	-	1.5 0.127	601.1 1.56	1295.3 0.672	2963.8 0.229	3974 0.12	4361 0.03
1,000	16.85 0.022	8.66 0.07	9.03 110.5	12.45 0.09	8.63 0.11	1430.6 0.5571	3497.7 0.149	4616.8 0.069	4814 0.05	4344 0.03

Table 6-13: YCSB++ Single Thread Workload (AWB)

Nodes	Workload (50 Threads)									
	A	B	C	D	E	F				
						10 KB	100 KB	1 MB	10 MB	100 MB
300	369.45 0.024	238.6 0.031	219.27 180.51	272.8 0.125	199.6 0.138	5512.6 3.50	3348.9 2.855	5007.5 0.635	7501 0.68	6983 0.768
500	575.27 0.031	284.0 0.034	261.23 157.81	267.5 0.124	300 0.126	-	-	-	-	-
1,000	94.63 0.027	50.76 0.036	49.02 1003.6	60.52 0.09	48.59 0.133	4370.6 2.56	4500.4 1.774	5000 0.242	3643 0.50	3726 0.223

Table 6-14: YCSB++ 50 Thread Workload (AWB)

7 BENCHMARK RESULTS FROM AMAZON EC2

7.1 Continuous Test Results

We ran the Accumulo continuous tests on Amazon EC2 clusters of 100, 330, 500, and 1,000 tablet server configurations. For each test run, we used six additional servers that ran the core services. These included: one Accumulo master, one Hadoop name node, one Hadoop JobTracker, and the three ZooKeeper servers. All instances were m1.large with initial tests performed using Ubuntu server version 11.10 and later tests using Ubuntu 12.10. Instances used EBS- based root devices without provisioned IOPS. HDFS and Accumulo data were stored on local ephemeral disk. Further details are given in Appendix C. These instances were located in the US-West-1 (California) and US-West-2 (Oregon) regions.

An Amazon EC2 m1.large instance has about one-sixth resident memory and one-fourth core of the slave machines used in the test on AWB. This setup obviously put tighter restrictions in memory allocation to the service and test processes. As with the tests on the AWB, test clients were run on the same machines that ran the tablet servers. Each test client was configured to perform one billion operations (ingest/query) using the configurations described in Section 5.4. In running high-throughput tests on presplit tables, a 60 MB buffer size was used and the number of threads was set to four to avoid resource exhaustion. In the results we present here, we use M, B and T to represent 10^6 $(millions)$, $10^9(billions)$, and $10^{12}(trillions)$. We also use GB to represent gigabytes, 10^9 bytes.

7.1.1 Performance Tests

We present results of ingest and query throughputs. As in the tests for the AWB detailed in the previous section, we ran the Ingest Only, Walker Only, and Batch Walker Only tests. To understand the best-case performance, we ran the tests on presplit tables. We also used tables that were not presplit to estimate worst-case performance. Furthermore, we ran a set of Partial-Ingest-Full-Walker/Batch Walker and Full-Ingest-Full-Walker/Batch Walker tests.

Ingest Only Tests

We ran Ingest tests on presplit tables as well as on tables without presplits. Although the former scenario excludes the overhead of tablet-split operations, both had file compaction and load-balancing overhead.

Continuous Ingest with Presplit Tables

We used presplit tables that were initially empty. We used splits that resulted in around 15–16 tablets per tablet server, and modified the table-split threshold to 128 GB. We had one set of ingest clients, each client configured to use a 60 MB buffer and 4 threads.

Table 7-1gives the results of these runs. These tests were run in the US-West-2 region. There were 100, 500, and 995 tablet servers overall for each, and 15 and 16 tablet servers per machine. In the cluster, the Accumulo master, Hadoop Name Node, and Hadoop JobTracker services were run on separate server nodes. Additionally, three separate server nodes were used to form the ZooKeeper quorum, with each hosting a ZooKeeper service instance. In each case, we ran the test for 4–5 hours continuously. In each case, the initial table size was zero.

Sustained Continuous Ingest with Presplit Tables

Table 7-2 presents Ingest test results for a presplit table for 11 hours on a 1,000-node cluster with 931 tablet servers. Due to a lack of resource availability in the US-West-2 region, this test was run in the US-West-1 region. Throughout the course of this test, we observed a relatively lower throughput than for a shorter run of the same test in the US-West-2 region (as Table 16 shows). In investigating the cause of this throughput degradation, we compared TCP ping times between a few pairs of nodes within the US-

West-1 and US-West-2 regions. In all cases, we observed on average a two-times higher ping time in the US-West-1 region compared with the US-West-2 region. This issue was not present for other tests executed in the US-West-1 region; in those cases, we verified and found that the ping values were in line with those experienced in US-West-2. Unfortunately, we were unable to re-run this test once the throughput issue had abated.

For this run, we used three test clients per server, each with a 1 GB buffer and 16 threads. To avoid splitting, the table split threshold used was 128 GB. We started the test with 942 tablet servers. During the course of the test, five tablet servers went down, but the Continuous Ingest test went on without any interruption.

Cluster Size	Tablets per Tablet Server	Initial Table Size (Entries)	Final Table Size (Entries)	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
16	40	0	4.58 B	2.9	0.781 M	0.441 M
100	15	0	56.60 B	4.925	5.673 M	3.280 M
330	15	0	105 B	3.65	14.410 M	8.256 M
500	15	0	170 B	4.51	20.178 M	10.870 M
1,000	15	0	205.4 B	4.23	25.085 M	14.076 M

Table 7-1: Continuous Ingest into Presplit Tables (Amazon EC2)

Cluster Size	Tablets per Tablet Server	Initial Table Size (Entries)	Final Table Size (Entries)	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
1,000	15–16	0	0.332 T	11	20 M	9 M

Table 7-2: Sustained Continuous Ingest into Presplit Tables (Amazon EC2)

Continuous Ingest Without Presplit Tables

As in the case of AWB, we used tests run on tables without presplit as a worst-case scenario. Each table begins the test with a single tablet. As the amount of data in a tablet approaches a threshold, the tablet splits in two. The process of splitting introduces latency into ingest but ultimately allows the load to be distributed across multiple tablet servers. The default table split threshold is 1 GB. These tests were run on US-West-2.

Cluster Size	Tablet Split Threshold	Initial Table Size	Final Table Size	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
16	1 GB	1.68M	10.718M	18.38	272,362	167,081
100	1 GB	1.1M	1.3B	1.11	1,703,227	315,091
500	1 GB	4.2M	27.332B	12.8	4,264,292	617,420

Table 7-3: Continuous Ingest into Non-Presplit Tables (Amazon EC2)

Walker, Batch Walker, and Mixed Tests

The Walker and Batch Walker tests were run to examine scan latencies in different configurations. We also ran mixed tests involving the simultaneous running of the Ingest and Walker tests. In the Partial Ingest test, we used one-third of the Ingest test clients than that of Walker test clients.

Walker and Batch Walker on Presplit Tables

We ran the Walker and Batch Walker tests on a presplit table with 15–16 tablets per server. One test client was used on each tablet server node with a 60-MB buffer and 4 threads. The table split threshold was set to 128 GB to prevent splitting. The tests on a 100-node cluster were run on the US-West-1 region. The tests on the 330, 500 and 1000 node clusters were run in the US-West-2 region.

Test	Table Size (Entries)	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
100-Node Test				
Walk Only	12.82 B	0.94	2.52 K	2.19 K
Batch Only	12.82 B	0.94	89.67 K	18.41 K
330-Node Test				
Walk Only	105,074	0.92	2.46 K	2.27 K
Batch Only	105,074	0.57	242.98 K	21.52 K
500-Node Test				
Walk Only	171,605	0.92	3.6 K	1.77 K
Batch Only	171,605	0.94	278.65 K	27.24 K
1,000-Node Test				
Walk Only	207,358	0.43	4.75 K	4.07 K
Batch Only	207,358	0.51	439.45 K	38.73 K

Table 7-4: Walker and Batch Walker Tests on Presplit Tables (Amazon EC2)

With respect to Table 7-4, the average query latencies for the Walker test for the 100-, 330-, 500-, and 1,000-node clusters were 0.45, 0.44, 0.57, and 0.25 milliseconds, respectively. For the Batch Walker test, these were 54.3, 46.5, 36.7, and 25.8 microseconds, respectively. Parallel query execution and cluster size contribute to an order of magnitude difference in latency.

In terms of throughput in bytes per second, noting that the table entry size is 50 to 54 bytes, the query throughput in the Batch Walker tests for 100 tablet servers reached almost a million bytes per second. For 330 tablet servers, it was 1.2 million bytes per second. It reached a 2.1 million bytes per second query rate for the 995 tablet servers. The tests were running a typical workload and were not specifically investigating maximum query throughput.

Walker and Batch Walker Tests on Non-Presplit Tables

We ran the Walker Only and Batch Walker Only tests with a 60-MB buffer and 4 threads. We used 1-GB table split thresholds. We did not expect the table split threshold to impact query throughput, as no tablet split may occur during these tests. The table size was expected to influence performance, with larger tables supporting more parallel transactions. Table 7-5 presents results on 100- and 500-node clusters with the same order of test clients.

Test	Table Size (Entries)	Test Period (Hours)	Maximum (Operations per Second)	Average (Operations per Second)
100-Node Test				
Walk Only	143.112 B	3.58	279	217
Batch Only	143.112 B	1.9	98,453	1421
500-Node Test				
Walk Only	27.295 B	0.90	701	211
Batch Only	27.295 B	1.33	121,081	4697

Table 7-5: Walker and Batch Walker Tests on Non-Presplit Tables (Amazon EC2)

The average latencies for the Walker test for 100- and 500-node clusters were 4.6 and 4.7 milliseconds, respectively. For the Batch Walker test, these were 0.7 and 0.2 milliseconds, respectively.

Full and Partial Ingest, Walker, and Batch Walker Tests

The results of ingest throughput and walk latency in a Full-Ingest-Full-Walker/Batch Walker (~100 and 500 ingest clients) and Partial-Ingest-Full-Walk/Batch Walk (30 and 150 test clients) in 100- and 500-node clusters are given here. The table split threshold was 128 GB for these tests for both the100- and 500-node clusters. Before the mixed tests were run, we grew the table to an initial size of the order of 4–10 B entries. Then we ran the mixed-mode tests. Table 7-7 and Table 7-8 contain the results for presplit and non-presplit tables.

Ingest Type	Walk Type	Initial Table Size	Final Table Size	Test Period (Hours)	Maximum Ingest (Operations per Second)	Maximum Query (Operations per Second)	Average Ingest (Operations per Second)	Average Query (Operations per Second)
100 Nodes								
Full	Walk	8.6 B	10.5 B	0.59	2.95 M	1051	974 K	482
Partial	Walk	10.5 B	14.0 B	0.71	2.42 M	1166	1.39 M	611
500 Nodes								
Full	Walk	3.9 B	6.2 B	0.25	6.3 M	822	2.52 M	163
Partial	Walk	7.0 B	9.1 B	0.28	5.6 M	4794	2.16 M	1581

Table 7-6: Mixed Workloads on Presplit Tables (Amazon EC2)

Ingest Type	Walk Type	Initial Table Size	Final Table Size	Test Period (Hours)	Maximum Ingest (Operations per Second)	Maximum Query (Operations per Second)	Average Ingest (Operations per Second)	Average Query (Operations per Second)
100 Nodes								
Full	Walk	1.4 B	4.0 B	0.55	2.5 M	18,542	1.32 M	6198
Partial	Walk	4.1 B	7.4 B	0.73	2.3 M	1145	1.23 M	661
500 Nodes								
Full	Walk	1.5 B	2.4 B	0.27	1.9 M	5928	670 K	2691
Partial	Walk	2.4 B	3.1 B	0.29	1.8 M	7329	610 K	5063

Table 7-7: Mixed Workloads on Non-Presplit Tables (Amazon EC2)

7.1.2 Scalability Tests

We evaluated scalability by observing the test performance on clusters of increasing size and proportionately increasing workloads. This was achieved by increasing the number of test clients in conjunction with the cluster size. As in the case of the test on the AWB cluster, we used the average ingest from Table 7-1 and query rates from Table 7-4. Figure 7-1 gives the plots for the average ingest, query, and batch query rates for presplit table.

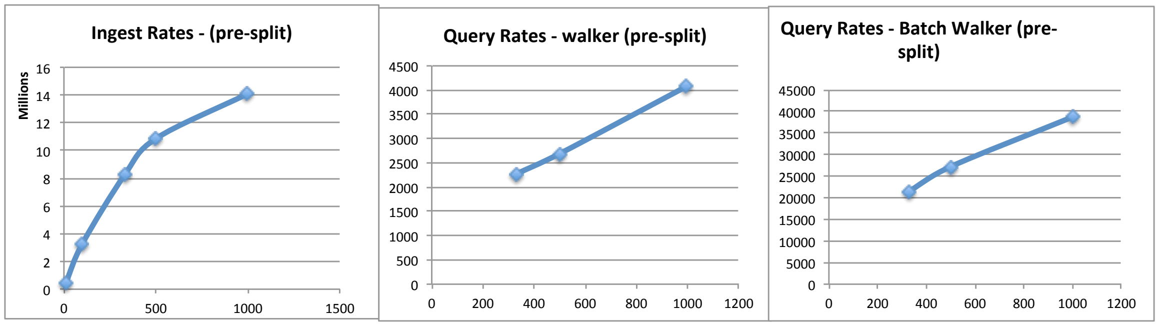

Figure 7-1: Average Throughputs in Presplit Table Clusters of Increasing Size

We used 10, 100, 330, 500, and 995 parallel test clients in 16-, 100-, 330-, 500-, and 1,000-node Amazon EC2 m1.large instance-based clusters. We observed the ingest rates increased almost linearly, dropping slightly after 500 nodes for the Ingest test. The query rates were linear with the size of the cluster.

7.1.3 Cell-Level Security Test

We evaluated performance with and without the cell-level security options. We used the same visibility, authorization files, and flags as used in the AWB tests. Table 23 gives the results for running the Ingest test with and without ACL on clusters of 100 and 500.

Continuous Ingest Test	**Table Size (Millions of Entries)**	**Period (Hours)**	**Maximum Ingest**	**Average Ingest**
100				
No ACL	0–1.7 B	0.9	2.4 M	0.5 M
With ACL	0–1.6 B	1.0	1.7 M	0.5 M
500				
No ACL	0–1.4 B	1.5	0.9 M	0.3 M
With ACL	0–1.7 B	1.5	1.8 M	0.3 M

Table 7-8: Continuous Ingest with Varying Visibilities (Amazon EC2)

We saw that results are quite close, indicating any overhead, corresponding to the ACL attributes used, is not significant.

7.1.4 MapReduce Table Verification Tests

The Hadoop MapReduce job to verify a table of 12 million entries took 2 minutes, 57 seconds with 10 map threads on a 10-node cluster. It took 9 minutes, 12 seconds with 200 map threads in a 100-node cluster.

7.2 YCSB Results

We ran the YCSB++ benchmarks on 10- and 100-node configurations on the Amazon EC2 US-West-2 region using m1.large instances as described previously. We used the five YCSB++ workloads and Workload F as in the AWB test. Table 24 and Table 25 give the throughput and latency observed for each workload and cluster size. In each table entry, the top number is the throughput in operations per second and the bottom number is the latency in milliseconds.

Nodes	Workload (1 Thread)									
	A	B	C	D	E	F				
						10 KB	100 KB	1 M	10 M	100 M
10	158.1 0.04	84.54 0.06	81.37 12.23	90.32 0.24	49.6 0.201	379.88 2.56	996.9 0.92	2884.3 0.244	2897 0.14	2824 0.078
100	167.97 0.046	98.24 0.065	94.67 10.47	107.3 0.196	50.53 0.198	601.1 1.56	1295.3 0.672	2963.8 0.229	3974 0.12	4361 0.03

Table 7-9: YCSB++ Single Thread Workload (Amazon EC2)

Nodes	Workload (50 Threads)									
	A	B	C	D	E	F				
						10 KB	100 KB	1 M	10 M	100 M
10	570.2 0.372	295.4 0.63	284.74 162.95	313.4 0.247	156.7 1.126	987.26 33.82	2836.8 9.58	3567.6 3.144	3209 3.62	3121 3.65
100	1310.6 1.05	1131 2.42	1162.8 35.79	1062 0.893	449.7 3.808	1120.7 36.028	1324.1 22.657	1113.2 8.746	978 10.2	-

Table 7-10: YCSB++ 50 Thread Workload (Amazon EC2)

8 CONCLUSION

We have presented results for several benchmark tests using the continuous test suite available with the open source distribution of Accumulo distributed key-value store. We have used the Ingest, Walker, and Batch Walker tests independently as well as in mixed mode for defining different workloads of write and read operations. We have used the Verify test to run table verification tests that use Hadoop MapReduce. We have also tested the overhead of using a cell-level ACL for ingest operations.

A fixed number of test clients were run on a slave node of the cluster that hosts a tablet server in the Accumulo configuration. This makes the number of test clients proportional to the number of tablet servers. In general, by increasing the number of test clients and associated buffers, when the local servers' resources allow it, we could increase the operational load on the table store.

We used two target test environments: a high-performance cluster provided by AWB and an Amazon EC2 cloud-based standard commodity cluster. The platform differences are apparent in the benchmark results we captured. There are multiple factors that impact performance. Nevertheless, we were able to observe a number of salient features of throughput/latency, scalability, and sustained ingest.

We observed that on a 1,000-node configuration, the average ingest performance for presplit tables reached an order of 100 million table entry (equivalent to 5.4 billion bytes per second) ingest operations per second on the AWB high-performance cluster. This value is around 14 million table entry ingest operations per second (equivalent to 0.8 billion bytes per second) on a 1,000-node Amazon EC2 (m1.large instance) cluster. Overall, we observed that the Ingest test performance on the AWB high-performance cluster was around 7 to 10 times faster than on the Amazon EC2 cluster. We observed that the scalability of the Accumulo table store was almost linear with the size of the cluster in both cluster environments.

We observed that the Walker and Batch Walker test performance on the AWB high-performance cluster and the Amazon EC2 cluster were generally similar. We observed a query rate of the order of 2 million bytes per second with batch walker in both of the cluster environments used in our tests. In the mixed-mode tests, generally, a lower ingest load with the same walk load resulted in better query rates. Our results for running the Ingest test with and without the cell-level ACL feature showed very low overhead in both environments.

The continuous test suite workload is defined by the ingest of the large continuous linked-list structure and the traversal using the Walker and Batch Walker tests (as well as other tests in the suite, such as the Scanner test). These test workloads correspond to an application environment that gives insight into operations on indexed and graph-like data structures. We have also run the YCSB workloads in our benchmark study on both cluster environments, including an examination of the impact of different buffer sizes in performance.

9 ACKNOWLEDGEMENTS

We would like to wholeheartedly thank the Accumulo developers Eric Newton and Keith Turner for valuable suggestions of testing procedures, system configurations, and troubleshooting approaches; the EMC^2/Greenplum AWB team members Tasneem Maistry and Clinton Ooi for their enthusiasm and extensive support for conducting the test on the AWB; the EMC^2/Greenplum business development and consulting team, Nick Cayou, Michael Parks, and Don Miner, for their support in planning this activity; and Paul Brown and Aaron Cordova of Koverse for helping to get the ball rolling on this work. We would like to thank Todd Hildebrant and Dave Hirko of AWS for all the help provided.

10 REFERENCES

1. R. Cattell. Scalable SQL and NoSQL Data Stores. http://cattell.net/datastores/Datastores.pdf.
2. M. Seltzer. 2008. Beyond Relational Databases. Communications of the ACM, 51(7).
3. R. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. November 2006. “BigTable: A Distributed Storage System for Structured Data.” In Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation. Seattle, WA: OSDI 2006.
4. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. October 2007. “Dynamo: Amazon’s Highly Available Key-Value Store.” In Proc. of the 21st ACM Symposium on Operating Systems Principles. Stevenson, WA: SOSP 2007.
5. HBase. Apache HBase. http://hbase.apache.org/.
6. Apache Cassandra. http://cassandra.apache.org/.
7. Project Voldemort. http://project-voldemort.com.
8. Apache Accumulo. http://accumulo.apache.org.
9. M. Cafarella, E. Chang, A. Fikes, A. Halevy, W. Hsieh, A. Lerner, J. Madhavan, and S. Muthukrishnan. 2008. Data Management Projects at Google. SIGMOD Record, 37(1).
10. SciDB. Use Cases for SciDB. http://www.scidb.org/use/.
11. SimpleDB. http://aws.amazon.com/simpledb.
12. SQL Data Services/Azure Services Platform. http://www.microsoft.com/azure/data.mspx.
13. Google App Engine. http://appengine.google.com.
14. Yahoo! Query Language. http://developer.yahoo.com/yql/.
15. TPC Transaction Processing Performance Council. http://www.tpc.org/information/benchmarks.asp.
16. Standard Performance Evaluation Corporation. http://www.spec.org/benchmarks.html.
17. J.L. Seng, Y. Lin, J.C. Wang. 2008. “A Comparative Study of Database Benchmark in Internet Commerce.” hicss, vol. 8, pp.263, 35th Annual Hawaii International Conference on System Sciences. HICSS 02-Volume 8.
18. L. Gillam, B. Li, J. O’Loughlin, A. P. S. Tomar. 2012. “Fair Benchmarking for Cloud Computing Systems.” University of Surrey Technical Report.
19. J. Gray, editor. 1993. “The Benchmark Handbook for Database and Transaction Processing Systems. Morgan Kaufmann.
20. F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber. “BigTable: A Distributed Storage System for Structured Data.” OSDI 2006.
21. “Benchmarking Cassandra Scalability on AWS—Over a Million Writes per Second.” http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html.
22. “Hypertable vs. HBase Performance Evaluation II.” http://hypertable.com/why_hypertable/hypertable_vs_hbase_2/.
23. B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. June 10-11, 2010. “Benchmarking Cloud Serving Systems with YCSB.” In Proc. of the 1st ACM Symposium on Cloud Computing. SoCC 2010.
24. B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, R. Yerneni. August 2008. “PNUTS: Yahoo!’s hosted data serving platform.” Proc. VLDB Endowment, Volume 1, Issue 2.
25. MySQL Cluster. http://www.sql.com/tech-resources/articles/mysql-cluster-7.2-ga.html.

26. S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lopez, G. Gibson, A. Fuchs, B. Rinaldi. October 27-28, 2011. "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores." Cascais, Portugal: SOCC 2011.
27. Graph Data Structure. http://www.algorithmist.com/index.php/Graph_data_structures.
28. EMC/Greenplum AWB Whitepaper. http://www.greenplum.com/sites/default/files/Greenplum-Analytics-Workbench-Whitepaper.pdf.
29. Amazon EC2. http://aws.amazon.com/ec2/.
30. P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil. 1996. "The Log-Structured Merge-Tree (LSM-Tree)." Acta Informatica, Volume 33, Issue 4, pp. 351-385.
31. Apache Hadoop. http://hadoop.apache.org.
32. Apache Accumulo User Manual Version 1.4. http://accumulo.apache.org/1.4/user_manual/.
33. Greenplum UAP Stack. http://www.greenplum.com/sites/default/files/2012_0209_UAP_WP.pdf.
34. Apache ZooKeeper. http://zookeeper.apache.org.
35. Apache Accumulo in the Cloud. Amazon EC2. http://www.slideshare.net/acordova00/accumulo-on-ec2.
36. Apache Accumulo Tutorial. http://www.accumulodata.com/ec2.html.

11 APPENDIX A: EMC^2/GREENPLUM CLUSTER CONFIGURATION

The EMC^2/Greenplum AWB is built to provide a mixed-mode environment to support innovation and validation in data warehousing and management at scale. Mixed-mode environments provide an easy way to combine structured and unstructured data using both traditional SQL and Apache Hadoop/no-SQL techniques without having to rework existing processes.

The AWB is a large server cluster organized as 50 data racks, three core racks, and one infrastructure rack. There are 1,000 data nodes, six master nodes, and a number of infrastructure nodes to support deployments, monitoring, and data loading.

Each of the data racks contains 20 data nodes, and each node encompasses 12 disk drives. The first two drives are mirrored OS currently running CentOS 6.1. The additional 10 drives are Just a Bunch of Disks used for data partitions. There are 50 data racks (20 data nodes per rack). A data rack has switches and data nodes. A data node has 12 disk drives; disk 0 and disk 1 are RAID 1 (mirrored for OS), 12x2 TB drives (Seagate 7200 rmp), 48 GB memory, dual Intel Westmere (Hex-core) CPUs—24 cores total. The master nodes are 98 GB memory, 6x 136 GB 2.5" drives (RAID 5), dual Intel Westmere (Hex-core) CPUs—24 cores total, 1/2U nodes in four-node chassis. Inside a core rack, there are four layer-2 Mellanox switches, one top-of-the-rack Mellanox switch, one layer-2 Netgear switch, two four-blade 2U SuperMicro chassis. The infrastructure rack contains six layer-3 Mellanox switches, one top-of-the-rack Mellanox switch, two layer-3 Netgear switches, one SMC switch, two four-blade 2U SuperMicro chassis, 10 2U SuperMicro. The cluster is running Greenplum Hadoop, GPHD 1.2, based on the open source Apache 1.0.3 Hadoop stack. The additional patches on top of Apache Hadoop 1.0.3 consist of the Greenplum stack. The vision is to have the Greenplum Unified Analytics Platform (UAP) stack running on the AWB. This offering includes Greenplum DB, Greenplum HD, and Greenplum Chorus. This suite will allow collaboration among data scientist teams, providing a platform that can process structured and unstructured data seamlessly. The team is currently building out this vision by having Greenplum HD, and very soon Greenplum DB, collocated on the AWB. More info about the UAP stack can be found in [33].

The Apache Accumulo configurations in the AWB cluster environment described previously had a master, Apache Hadoop name server, and Apache Hadoop JobTracker running on separate servers. We had five Apache ZooKeeper servers in quorum, as Figure A-1 shows. Typically, we had the remaining nodes as data and tablet servers. Hadoop data nodes and Accumulo tablet servers were co-located on the same node. These nodes were also used as test clients. The servers were configured to run Sun/Oracle Java 1.6. We used a separate client node to launch tests and collect results.

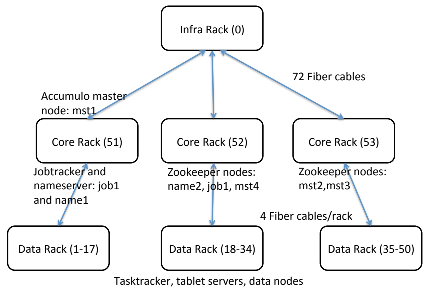
Infra Rack (0)
72 Fiber cables
Accumulo master node: mst1
Core Rack (51)
Core Rack (52)
Core Rack (53)
Jobtracker and nameserver: job1 and name1
Zookeeper nodes: name2, job1, mst4
Zookeeper nodes: mst2,mst3
4 Fiber cables/rack
Data Rack (1-17)
Data Rack (18-34)
Data Rack (35-50)
Tasktracker, tablet servers, data nodes

Figure A-1: AWB Cluster Topology and Server Locations

The AWB will continue to offer a 1,000-node instance of Apache Accumulo available to any project interested in using this product. This will provide another tool for projects to do big data analytics. The cell-level security provided by Apache Accumulo will allow projects interested in processing sensitive data, such as healthcare data, a mechanism to do so on the AWB. Faster analytics, with the ability to store petabytes of data on the AWB, could provide groundbreaking insights.

12 APPENDIX B: EMC^2/GREENPLUM SERVICES CONFIGURATION

12.1 Apache Accumulo Parameters

(Accumulo-env.sh)

```
JAVA_HOME=$JAVA_HOME
HADOOP_HOME=$HADOOP_LOCATION
ACCUMULO_LOG_DIR=$ACCUMULO_HOME/logs
ZOOKEEPER_HOME="$ZOOKEEPER_HOME”
ACCUMULO_TSERVER_OPTS="-Xmx6g -Xms6g -Xss256k -XX:NewSize=1G”
ACCUMULO_MASTER_OPTS="-Xmx1g -Xms128m”
ACCUMULO_GC_OPTS="-Xmx128m -Xms64m”
ACCUMULO_MONITOR_OPTS="-Xmx128m -Xms64m”
ACCUMULO_LOGGER_OPTS="-Xmx1g -Xms128m”
ACCUMULO_GENERAL_OPTS="-XX:+UseConcMarkSweepGC -
XX:CMSInitiatingOccupancyFraction=75”
ACCUMULO_OTHER_OPTS="-Xmx4g -Xms64m"
```

(Accumulo-site.xml)

```
logger.dir.walog  Value -   /data1/accumulo-1_4/walogs,/data2/accumulo-1_4/walogs
tserver.logger.count        2
tserver.memory.maps.max 6G
tserver.cache.data.size     512M
tserver.cache.index.size    4G
tserver.compaction.major.concurrent.max     10
tserver.compaction.minor.concurrent.max     10
table.file.blocksize        256M
```

12.2 Hadoop parameters

(hadoop-env.sh)

```
HADOOP_HEAPSIZE=1000
HADOOP_NAMENODE_OPTS="-Xmx65536m -server -XX:ParallelGCThreads=8 -XX:+UseConcMarkSweepGC
-XX:ErrorFile=/var/log/gphd/hadoop/hs_err_pid%p.log -XX:NewSize=3G -XX:MaxNewSize=3G -
Xloggc:/var/log/gphd/hadoop/gc.log-`date +'%Y%m%d%H%M'` -verbose:gc -XX:+PrintGCDetails -
XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -Dsecurity.audit.logger=INFO,DRFAS -
Dhdfs.audit.logger=INFO,DRFAAUDIT -Dcom.sun.management.jmxremote
.management.jmxremote
HADOOP_SECONDARYNAMENODE_OPTS="-Xmx65536m -server -XX:ParallelGCThreads=8 -
XX:+UseConcMarkSweepGC -XX:ErrorFile=/var/log/gphd/hadoop/hs_err_pid%p.log -XX:NewSize=3G -
XX:MaxNewSize=3G -Xloggc:/var/log/gphd/hadoop/gc.log-`date +'%Y%m%d%H%M'` -verbose:gc -
XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -Dsecurity.audit.logger=INFO,DRFAS
-Dhdfs.audit.logger=INFO,DRFAAUDIT -Dcom.sun.management.jmxremote
HADOOP_DATANODE_OPTS="-Xmx1024m -Dcom.sun.management.jmxremote
HADOOP_BALANCER_OPTS="-Dcom.sun.management.jmxremote
HADOOP_JOBTRACKER_OPTS="-Xmx65536m -server -XX:ParallelGCThreads=8 -
XX:+UseConcMarkSweepGC -XX:ErrorFile=/var/log/gphd/hadoop/hs_err_pid%p.log -XX:NewSize=3G -
XX:MaxNewSize=3G -Xloggc:/var/log/gphd/hadoop/gc.log-`date +'%Y%m%d%H%M'` -verbose:gc -
XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -Dsecurity.audit.logger=INFO,DRFAS
-Dmapred.audit.logger=INFO,MRAUDIT -Dmapred.jobsummary.logger=INFO,JSA -
Dcom.sun.management.jmxremote
```

(from mapred-site.xml)

```
mapred.map.tasks = 12
mapred.reduce.tasks = 8
mapred.tasktracker.map.tasks.maximum = 12
mapred.tasktracker.reduce.tasks.maximum = 8
mapred.map.child.java.opts = -Xmx1536M
mapred.reduce.child.java.opts = -Xmx4096M
mapred.map.tasks.speculative.execution = true
mapred.reduce.tasks.speculative.execution = true
mapreduce.tasktracker.outofband.heartbeat = true
io.sort.mb = 10
tasktracker.http.threads = 60
mapred.job.tracker.handler.count = 120
mapred.tasktracker.shuffle.provider.plugin = com.mellanox.hadoop.mapred.UdaShuffleProviderPlugin
```

(from hdfs-site.xml)

```
dfs.replication = 3
dfs.datanode.handler.count = 32
dfs.namenode.handler.count = 100
dfs.datanode.max.xcievers = 3000
dfs.name.dir = /data/hadoop/name
dfs.data.dir=
/data1/hadoop/data,/data2/hadoop/data,/data3/hadoop/data,/data4/hadoop/data,/data5/hadoop/data,/data6/h
adoop/data,/data7/hadoop/data,/data8/hadoop/data,/data9/hadoop/data,/data10/hadoop/data
dfs.block.size = 134217728
```

12.3 ZooKeeper parameters

(from zoo.cfg)

```
tickTime=2000
initLimit=10
syncLimit=5
maxClientCnxns=1000
globalOutstandingLimit=10000
```

13 APPENDIX C: AMAZON EC2 CLUSTER CONFIGURATION

The Amazon EC2 cluster was created with Amazon EC2 m1.large instances. The Apache Accumulo, Apache Hadoop, and Apache ZooKeeper services were configured and run on the cluster. The configuration was adapted from [36] with some modifications. Apache Accumulo release version 1.4.2, Apache Hadoop version 1.0.4, and Apache ZooKeeper version 3.4.3 were used. Test clients were co-located on the slave nodes. The nodes ran Ubuntu versions 11 and 12. Instances were run in the same availability zone.

13.1 Node Types in Cluster

- **One Accumulo Master node.** Hosting Apache Accumulo master, monitor, garbage collector services
- **Three ZooKeeper nodes.** Quorum peers running Apache ZooKeeper services.
- **One HDFS NameNode**. Hosting Apache Hadoop NameNode service
- **One JobTracker node**. Hosting Apache Hadoop JobTracker service
- **Remaining nodes in cluster are slave nodes**. Hosting Apache Accumulo tablet servers, loggers, Apache Hadoop TaskTrackers, and DataNodes

13.2 Preparing Amazon Machine Instance for Amazon EC2 Instances

Using the Amazon EC2 services dashboard, a standard Ubuntu Amazon Machine Instance is launched. Ubuntu Server (64-bit) version 12.04.01 Precise Pangolin was used. An ephemeral storage device /dev/sdb is available by default. Add an addition ephemeral device /dev/sdc. A security group with ports open to allow Secure Shell (SSH), TCP, and User Datagram Protocol ports 0-65000 (doing so allows access to the Accumulo, Hadoop, and ZooKeeper services on default ports), Internet Control Message Protocol, and the HTTP port 80 is created to be associated with the launched instance.

Log into the instance via SSH using proper security credentials. The main steps to install software and prepare devices are given here. Example shell commands are given in the box. We assume the Amazon security keys are available in ~/.ec2 and Amazon EC2 client library is available in ~/.ec2/bin, which is also the working directory.

a) SSH into the Amazon EC2 instance. The access code for the private key file should be 600.

```
ssh -i ~/.ec2/ranjankey.pem  ubuntu@ec2-54-245-169-199.us-west-2.compute.amazonaws.com
```

b) Configure a password-less SSH for the instance.

```
#configure passwordless SSH
#create a ssh key pair (with default key files and empty passphrase)
ubuntu@ip-10-253-76-162:~$ ssh-keygen
#(append the public key to the authorized_keys file)
#edit the /etc/ssh/ssh_config file to uncomment StrictHostKeyChecking and set the value to ‘no’
sudo vi /etc/ssh/ssh_config
```

c) Upload helper scripts and the split files (Section 13.7). Check the script testhosts.sh to confirm the internal IP is created properly. For example, in the case of AZ us-west-2, the script should have “us-west-2.compute.internal” and for AZ us-west-1, it should be us-west-1.compute.internal.

```
$ scp -i X.pem * ubuntu@ec2-50-18-147-116.us-west-1.compute.amazonaws.com:/home/ubuntu

conf-slaves.sh                                    100% 1394     1.4KB/s   00:00
conf-zk.sh                                       100%  184     0.2KB/s   00:00
copy-parallel-duper.sh                              100% 1015     1.0KB/s   00:00
copy-parallel.sh                                  100% 1299     1.3KB/s   00:00
etchosts.sh                                      100%   89     0.1KB/s   00:00
install.sh                                      100% 2144     2.1KB/s   00:00
installYCSB.sh                                     100%  309     0.3KB/s   00:00
setZK.sh                                         100%  253     0.3KB/s   00:00
setupData.sh                                      100%  782     0.8KB/s   00:00
setupDataAll.sh                                    100%   79     0.1KB/s   00:00
swappiness-spread.sh                                 100%  299     0.3KB/s   00:00
sync-continuous-env.sh                                100%  218     0.2KB/s   00:00
testhosts.sh                                     100%  245     0.2KB/s   00:00
```

d) Run install.sh script as root from /opt as working directory to install the Apache Hadoop, Java software development kit, required library, Apache Accumulo, Apache ZooKeeper, and other required software; add file system and mount drives; and copy the Apache Library into Apache Hadoop /lib.

```
ubuntu@ip-10-253-19-149:/opt$ sudo bash  ~/install.sh

ubuntu@ip-10-253-19-149:/opt$ sudo chown -R ubuntu .
```

e) Set up home paths, classpaths, library, and paths. Add the following lines into ~/.bashrc and reinitialize environment variables. Make the appropriate change to versions of jars and other values as necessary.

```
export JAVA_HOME="/usr/lib/jvm/java-6-openjdk-amd64"
export  HADOOP_HOME="/opt/hadoop"
export  ACCUMULO_HOME="/opt/accumulo"
export  ZOOKEEPER_HOME="/opt/zookeeper"
export LIBJARS=$ZOOKEEPER_HOME/zookeeper-3.4.3.jar:$ACCUMULO_HOME/lib/libthrift-
0.6.1.jar:$ACCUMULO_HOME/lib/accumulo-core-1.4.3.jar:$ACCUMULO_HOME/lib/cloudtrace-
1.4.3.jar:ACCUMULO_HOME/lib/commons-collections-3.2.jar:$ACCUMULO_HOME/lib/commons-configuration-
1.5.jar:$ACCUMULO_HOME/lib/commons-io-1.4.jar:$ACCUMULO_HOME/lib/commons-jci-core-
1.0.jar:$ACCUMULO_HOME/lib/commons-jci-fam-1.0.jar:$ACCUMULO_HOME/lib/commons-lang-
2.4.jar:$ACCUMULO_HOME/lib/commons-logging-1.0.4.jar:$ACCUMULO_HOME/lib/commons-logging-api-1.0.4.jar
export CLASSPATH=$HADOOP_HOME/hadoop-core-1.0.4.jar:$HADOOP_HOME/hadoop-tools-1.0.4.jar:$LIBJARS
export CLASSPATH=$CLASSPATH:"/home/ubuntu/YCSB/db/accumulo/conf"
export PATH=$PATH:$ACCUMULO_HOME/bin:$JAVA_HOME/bin
```

f) Configure Hadoop.

```
In hadoop-env.sh add JAVA_HOME,  add the following tags
HADOOP_NAMENODE_OPTS="-Xmx7000m -server -XX:ParallelGCThreads=4 -XX:+UseConcMarkSweepGC -
XX:NewSize=256M  -XX:MaxNewSize=256M"
HADOOP_SECONDARYNAMENODE_OPTS="-Xmx7000m -server -XX:ParallelGCThreads=4 -
XX:+UseConcMarkSweepGC -XX:NewSize=256M -XX:MaxNewSize=256M"
HADOOP_JOBTRACKER_OPTS="-Xmx7000m -server -XX:ParallelGCThreads=4 -XX:+UseConcMarkSweepGC -
XX:NewSize=256M -XX:MaxNewSize=256M"
HADOOP_DATANODE_OPTS="-Xmx512m -Xms512m"
HADOOP_TASKTRACKER_OPTS="-Xmx256m  -Xms256m"
export HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote $HADOOP_NAMENODE_OPTS"
export HADOOP_SECONDARYNAMENODE_OPTS="-Dcom.sun.management.jmxremote
$HADOOP_SECONDARYNAMENODE_OPTS"
export HADOOP_DATANODE_OPTS="-Dcom.sun.management.jmxremote $HADOOP_DATANODE_OPTS"
export HADOOP_BALANCER_OPTS="-Dcom.sun.management.jmxremote $HADOOP_BALANCER_OPTS"
export HADOOP_JOBTRACKER_OPTS="-Dcom.sun.management.jmxremote $HADOOP_JOBTRACKER_OPTS"
```

Place the following in the **$HADOOP_HOME/conf/core-site.xml**

```
<name>fs.default.name</name>
<value>hdfs://<IP address of namenode>:9000</value>
</property>
```

Add the following tags in **$HADOOP_HOME/conf/hdfs-site.xml**

```
<property>
<name>dfs.name.dir</name>
<value>/mnt/namenode</value>
</property>
<property>
<name>dfs.data.dir</name>
<value>/mnt/hdfs,/mnt2/hdfs</value>
</property>
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
<property>
<name>dfs.datanode.handler.count</name>
<value>12</value>
</property>
<property>
<name>dfs.namenode.handler.count</name>
<value>40</value>
</property>
<property>
<name>dfs.datanode.max.xcievers</name>
<value>3000</value>
</property>
<property>
<name>dfs.block.size</name>
<value>134217728</value>
</property>
```

Add the following property and values in **$HADOOP_HOME/conf/mapred-site.xml**

```
<property>
<name>mapred.job.tracker</name>
<value>10.253.79.156:9001</value>
</property>
<property>
<name>mapred.local.dir</name>
<value>/mnt/mapred,/mnt2/mapred</value>
</property>
<property>
<name>mapred.child.java.opts</name>
<value>-Xmx512m</value>
</property>
<property>
<name>mapred.map.tasks</name>
<value>1</value>
</property>
<property>
<name>mapred.reduce.tasks</name>
<value>1</value>
</property>
<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value>1</value>
</property>
<property>
<name>mapred.tasktracker.reduce.tasks.maximum</name>
<value>1</value>
</property>
<property>
<name>mapred.map.child.java.opts</name>
<value>-Xmx512m</value>
</property>
<property>
<name>mapred.reduce.child.java.opts</name>
<value>-Xmx512m</value>
</property>
<property>
<name>mapred.map.tasks.speculative.execution</name>
<value>true</value>
</property>
<property>
<property>
<name>mapred.reduce.tasks.speculative.execution</name>
<value>true</value>
</property>
<property>
<name>mapred.tasktracker.outofband.heartbeat</name>
<value>true</value>
</property>
<property>
<name>io.sort.mb</name>
<value>10</value>
</property>
<property>
<name>tasktracker.http.threads</name>
<value>40</value>
</property>
<property>
<name>mapred.job.tracker.handler.count</name>
<value>80</value>
</property>
```

g) Configure Accumulo.

```
copy the conf/examples/3GB/standalone/*  to the conf directory, add JAVA_HOME, HADOOP_HOME and
ZOOKEEPER_HOME in accumulo-env.sh, add JAVA_HOME into accumulo-env.sh
        export JAVA_HOME="/usr/lib/jvm/java-6-openjdk-amd64" (replace actual path)
        test -z "$ACCUMULO_TSERVER_OPTS" && export ACCUMULO_TSERVER_OPTS="${POLICY} -Xmx2000m -
        Xms2000m  -XX:NewSize=256m -XX:MaxNewSize=512m"
        test -z "$ACCUMULO_MASTER_OPTS"  && export ACCUMULO_MASTER_OPTS="${POLICY} -Xmx1g -
        Xms1g"
        test -z "$ACCUMULO_MONITOR_OPTS" && export ACCUMULO_MONITOR_OPTS="${POLICY} -Xmx1g -
        Xms256m"
        test -z "$ACCUMULO_GC_OPTS"      && export ACCUMULO_GC_OPTS="-Xmx256m -Xms256m"
        test -z "$ACCUMULO_LOGGER_OPTS"  && export ACCUMULO_LOGGER_OPTS="-Xmx1g -Xms256m"
        test -z "$ACCUMULO_GENERAL_OPTS" && export ACCUMULO_GENERAL_OPTS="-
        XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75"
        test -z "$ACCUMULO_OTHER_OPTS"   && export ACCUMULO_OTHER_OPTS="-Xmx1g -Xms256m"
        export ACCUMULO_LOG_HOST=`(grep -v '^#' $ACCUMULO_HOME/conf/masters ; echo localhost ) 2>/dev/null |
        head -1`
create a logs directory in $ACCUMULO_HOME
        ubuntu@ip-10-253-76-162:/opt/accumulo$ mkdir logs
in accumulo-site.xml change password to 'root'
<property>
   <name>trace.password</name>
  <value>root</value>
</property>
<property>
     <name>logger.dir.walog</name>
     <value>/mnt/walogs</value>
   </property>
   <property>
     <name>tserver.memory.maps.max</name>
     <value>1500M</value>
   </property>
   <property>
     <name>tserver.cache.data.size</name>
     <value>50M</value>
   </property>
   <property>
     <name>tserver.cache.index.size</name>
     <value>256M</value>
   </property>

copy the master/slaves files
        ubuntu@ip-10-254-11-47:/opt/accumulo$ vi conf/accumulo-site.xml
        ubuntu@ip-10-254-11-47:/opt/accumulo$ vi conf/accumulo-site.xml
        ubuntu@ip-10-254-11-47:/opt/accumulo$ cp ~/acc-masters conf/masters
        ubuntu@ip-10-254-11-47:/opt/accumulo$ cp ~/acc-masters conf/monitor
        ubuntu@ip-10-254-11-47:/opt/accumulo$ cp ~/acc-masters conf/gc
        ubuntu@ip-10-254-11-47:/opt/accumulo$ cp ~/acc-masters conf/tracers
        ubuntu@ip-10-254-11-47:/opt/accumulo$ cp ~/slaves conf/slaves
```

h) Configure ZooKeeper.

```
create conf/zoo.cfg by copying conf/zoo_sample.cfg and adding the following into zoo.cfg
        dataDir=/tmp/zookeeper
        # the port at which the clients will connect
        clientPort=2181
        server.1=<zookeeper server1>:2888:3888
        server.2=<zookeeper server2>:2888:3888
        server.3=<zookeeper server3>:2888:3888
        #
        maxClientCnxns=1000
```

i) Configure continuous test.

```
#initialize the test/system/continuous files
create continuous-env.sh from continuous-env.sh.example and ensure the following
put the proper values to HADOOP_HOME, ACCUMULO_HOME, JAVA_HOME, ZOOKEEPER_HOME
Ensure
INSTANCE=ci
USER=root
PASS=root
TABLE=ci
CHECKSUM=false
NUM_THREADS=4
MAX_MEM=1000000000

# create ingesters.txt from ingesters.txt.example
# create walkers.txt from walkers.txt.example
# create batch_walkers.txt from batch_walkers.txt.example
#create scanner.txt from scanner.txt.example
```

j) Install and configure pssh.

```
        # install parallel-ssh
        ubuntu@ip-10-254-17-100:~$ sudo apt-get install pssh
        ubuntu@ip-10-254-17-100:~$ cd /usr/bin
        ubuntu@ip-10-254-17-100:/usr/bin$ sudo ln -s parallel-ssh pssh
```

k) Ensure $ACCUMULO_HOME/test/system/continuous/logs directory is present.

We create an image of this instance and use it in launching cluster nodes.

13.3 Configure an Amazon EC2 Instance Cluster

First, we created an image of the instance created in Section 14.2. To create an Amazon EC2 instance cluster of N nodes, we launch N instance from this image. Because the image has password-less SSH configured, all the cluster nodes can SSH into each other without passwords and do not prompt for known-hosts authentication.

13.3.1 Identifying Accumulo, Hadoop, and ZooKeeper Nodes

Once the N instances in the cluster are running, we access and control them from a client machine. On this client machine, we use the script get-running-instance-default.sh to obtain the instance descriptions. Section 14.7 gives the Amazon EC2 helper scripts. We use the scripts proc-instance-domains.sh and proc-instance-IP.sh to get the public domain names and the internal IP addresses of the instances. We use public domain names to access the nodes from the client machine. We upload the internal IP addresses to the Apache Accumulo master node. This node acts as the main point of setting up and distributing the configuration files across the cluster.

Once the IP addresses of all the cluster nodes are uploaded to the Accumulo master node, we SSH into it and create the following files:

a) **acc-master.** Contains the IP address of the Accumulo master node
b) **master.** Contains the IP addresses of the Hadoop NameNode and JobTracker
c) **zknodes.** Contains the IP addresses of the ZooKeeper nodes
d) **slaves**. Contains the IP addresses of the N-6 nodes, which are Accumulo slaves hosting the tablet and logger services as well as the Hadoop data nodes and TaskTrackers

Illustration is given in the box.

```
$ . get-running-instances-default.sh >instances
$ . proc-instances-domain.sh
$ . proc-instances-IP.sh >allnodes
$ scp -i <**.pem>  allnodes ubuntu@<public domain name of controller>:/home/ubuntu/allnodes
```

Log in to the controller node, and create the acc-masters, masters, zknodes, and slaves files.

Edit the testhosts.sh file to remove the shutdown command, save it, and run the script etchosts.sh to edit the /etc/hosts file across the cluster to use IPV4. Doing so will reboot all the machines.

```
$ .  etchosts.sh
```

Configure Hadoop by adding the IP addresses of the NameNode (the first IP address in the masters file) and JobTracker (second IP address in the masters file) in the core-site.xml and mapred-site.xml files, and initialize the masters and slaves files.

```
Add namenode in core-site.xml
Add jobtracker in mapred-site.xml
Copy the ~/masters to $HADOOP_HOME/conf/masters
Copy the ~/slaves to $HADOOP_HOME/conf/slaves
```

Configure Accumulo by adding the IP addresses of the ZooKeeper nodes in the accumulo-site.xml file, and initialize the masters, monitor, tracers, gc, and slaves files from the acc-master, and slaves file in the home directory of the Accumulo master.

```
Add zookeepers into accumulo-site.xml
Copy the ~/acc-masters to $ACCUMULO_HOME/conf/masters
Copy the ~/acc-masters to $ACCUMULO_HOME/conf/monitor
Copy the ~/acc-masters to $ACCUMULO_HOME/conf/tracers
Copy the ~/acc-masters to $ACCUMULO_HOME/conf/gc
Copy the ~/slaves to $ACCUMULO_HOME/conf/slaves
```

Configure ZooKeeper, adding the IP values in zknodes for server1, server2, and server3 in zoo.cfg.

Sync these files across clusters using the conf-slaves.sh script. In larger clusters, use the copy-parallel.sh script for faster synchronization. The latter script uses a number of nodes to copy the files in parallel. It requires a file first containing the IP addresses of the nodes (say p of them) that will perform the parallel copy and p files—fileX, where X = 1, 2, … p containing the target nodes for each. These files need to be present in the home directory, which is where the scripts are run.

```
$ .  conf-slaves.sh
$ .   swappiness-spread.sh
```

```
$ .  copy-parallel.sh
$ .  conf-zk.sh
$  .  swappiness-spread.sh
```

From the home directory, run the setupDataAll.sh and setZK.sh scripts to initialize the local files for Hadoop file system use and initializing the IDs and files for the ZooKeepers.

```
$ .   setupDataAll.sh
$.    setZK.sh
```

Configure the continuous test files on the Accumulo master, which is where the tests are launched later. Edit the IP of the ZooKeepers in the continuous-env.sh file; copy the ~/slaves into ingester.txt, walker.txt, batch_walker.txt, and scanner.txt

13.3.2 Starting Services

```
$HADOOP_HOME/bin/hadoop namenode –format
$HADOOP_HOME/bin/start-dfs.sh
```

SSH into the NameNode, format the Hadoop file system (needed one time), and start the hdfs service. Check that the services are running by jps—NameNode and SecondaryNameNode will run on the NameNode, and DataNode will run on the slave nodes.

On the JobTracker node, start the mapreduce service. Check that JobTracker is running on the JobTracker node and TaskTrackers are running on the slaves.

```
$HADOOP_HOME/bin/start-mapred.sh
```

Start the ZooKeepers on each ZooKeeper node.

```
$ZOOKEEPER_HOME/ java -cp zookeeper-3.4.3.jar:lib/slf4j-api-1.6.1.jar:lib/slf4j-log4j12-1.6.1.jar:lib/log4j-1.2.15.jar:conf:.  org.apache.zookeeper.server.quorum.QuorumPeerMain conf/zoo.cfg
```

13.4 Create Instance and Run Accumulo

Init instance and Start Accumulo:

```
$ACCUMULO_HOME/bin/accumulo init
```

(The instance is 'ci' and the root password is 'root'.)

On the Hadoop NameNode, run the following to verify instance creation:

```
ubuntu@ip-10-253-42-16:/opt/hadoop$ bin/hadoop fs -ls /accumulo/instance_id
```

Start Accumulo:

```
$ACCUMULO_HOME/bin/start-all.sh
```

Check that the tablet servers are registered in ZooKeeper:

```
$ZOOKEEPER_HOME/bin/zkCli.sh –server <zk server>
```

Use the <monitor domain name>:50095 to launch the monitor page on a Internet browser.

13.5 Run Continuous Tests

Change the working directory:

```
cd $ACCUMULO_HOME/test/system/continuous/
```

Run the statistics collection program:

```
. start-stats.sh
```

Run the ingest test:

```
. start-ingest.sh
```

13.5.1 Creating Table—Partitioning Table

For testing with a presplit table, a split file is used. The split file is generated using a utility class called GenSplits that is included in the Continuous ingest code in Accumulo 1.5 and higher.

Use the split file to split the table ci.

```
ubuntu@ip-10-254-17-100:/opt/accumulo$ bin/accumulo shell -u root -p root
root@ci> createtable ci
root@ci> addsplits –sf <split file path>  -t   ci
root@ci> offline –t  ci
root@ci> online –t  ci
also, run the following to flush the !METADATA table
root@ci> flush  –t  !METADATA
```

13.5.2 Checking Results

In the Accumulo master node, $ACCUMULO/test/system/continuous/logs directory the files <time-stamp>*stats.out will have summary results of the tests.

In the test clients (slaves), the logs in the same path will have individual ingest client outputs in <time-stamp>*ingest.out.

The results of walker and batch walker tests will be in similar files with ‘walker’ and ‘batch_walker’ substituted for ‘ingest’.

13.6 Helper Scripts and Split Files

The following scripts are used for instance creation, initialization, configuration, copying, and synchronizing across clusters. The accumulo-gensplits.tar.gz contains the split file generation utility. The command is:

```
$java –cp target/accumulo-gensplits-1.4.2.jar org.apache.accumulo.util.GenSplits <number of splits> <split-file-name>
```

13.6.1 install.sh

```
#!/bin/bash
# run this from /opt directory as ubuntu

sudo sysctl -w vm.swappiness=0
echo -e "ubuntu\t\tsoft\tnofile\t65536" | sudo tee --append /etc/security/limits.conf
echo -e "ubuntu\t\thard\tnofile\t65536" | sudo tee --append /etc/security/limits.conf

# install software
RELEASE=`lsb_release -c | awk {'print $2'}`
sudo apt-get install python-software-properties -y
sudo add-apt-repository "deb http://archive.canonical.com/ $RELEASE partner"

wget http://www.fightrice.com/mirrors/apache/hadoop/common/stable/hadoop-1.0.4.tar.gz
tar -xzf hadoop-1.0.4.tar.gz
ln -s hadoop-1.0.4 hadoop

wget http://download.nextag.com/apache/zookeeper/zookeeper-3.4.3/zookeeper-3.4.3.tar.gz
tar -xzf zookeeper-3.4.3.tar.gz
ln -s zookeeper-3.4.3    zookeeper

sudo apt-get update
sudo apt-get install openjdk-6-jdk -y
java -version

sudo apt-get install gcc g++ python-software-properties xfsprogs -y

# get Accumulo

wget http://www.bizdirusa.com/mirrors/apache/accumulo/1.4.3/accumulo-1.4.3-dist.tar.gz
```

```
tar -xzf accumulo-1.4.3-dist.tar.gz
ln -s accumulo-1.4.3    accumulo

sudo cp accumulo/lib/accumulo-core-1.4.3.jar hadoop/lib/
sudo cp accumulo/lib/log4j-1.2.16.jar hadoop/lib/
sudo cp accumulo/lib/libthrift-0.6.1.jar hadoop/lib/
sudo cp accumulo/lib/cloudtrace-1.4.3.jar hadoop/lib/
sudo cp zookeeper/zookeeper*.jar hadoop/lib/
# setup data directory

sudo umount /mnt;
sudo /sbin/mkfs.xfs -f /dev/xvdb;
sudo mount -o noatime /dev/xvdb /mnt;

# change this if /dev/sdc is available
sudo mkdir  /mnt2;
sudo /sbin/mkfs.xfs -f /dev/xvdc;
sudo mount -o noatime /dev/xvdc /mnt2;


sudo chown -R ubuntu /mnt

sudo mkdir /mnt/hdfs
sudo mkdir /mnt/namenode
sudo mkdir /mnt/mapred
sudo mkdir /mnt/walogs

sudo mkdir /mnt2/hdfs
sudo mkdir /mnt2/mapred

sudo chown -R ubuntu  /mnt/hdfs
sudo chown -R ubuntu  /mnt/namenode
sudo chown -R ubuntu  /mnt/mapred
sudo chown -R ubuntu  /mnt/walogs

sudo chown -R ubuntu  /mnt2/hdfs
sudo chown -R ubuntu  /mnt2/mapred

#  chown of hadoop, accumulo and zookeeper directories recursively

sudo chown -R ubuntu hadoop
sudo chown -R ubuntu accumulo
sudo chown -R ubuntu zookeeper
```

13.6.2 conf-slaves.sh

```
#!/bin/bash
for i in `cat /home/ubuntu/allnodes`; do
echo "copying to " $i
#moving hadoop
	#scp -q /opt/hadoop/conf/hadoop-env.sh   ubuntu@$i:/opt/hadoop/conf/hadoop-env.sh
	scp -q /opt/hadoop/conf/core-site.xml ubuntu@$i:/opt/hadoop/conf/core-site.xml
	scp -q /opt/hadoop/conf/mapred-site.xml ubuntu@$i:/opt/hadoop/conf/mapred-site.xml
#	scp -q /opt/hadoop/conf/hdfs-site.xml ubuntu@$i:/opt/hadoop/conf/hdfs-site.xml
	scp -q /opt/hadoop/conf/masters ubuntu@$i:/opt/hadoop/conf/masters
	scp -q /opt/hadoop/conf/slaves ubuntu@$i:/opt/hadoop/conf/slaves

#moving accumulo
	#scp -q /opt/accumulo/conf/accumulo-env.sh ubuntu@$i:/opt/accumulo/conf/accumulo-env.sh
	scp -q /opt/accumulo/conf/accumulo-site.xml ubuntu@$i:/opt/accumulo/conf/accumulo-site.xml
	scp -q /opt/accumulo/conf/masters ubuntu@$i:/opt/accumulo/conf/masters
	scp -q /opt/accumulo/conf/slaves ubuntu@$i:/opt/accumulo/conf/slaves
	scp -q /opt/accumulo/conf/gc ubuntu@$i:/opt/accumulo/conf/gc
	scp -q /opt/accumulo/conf/monitor   ubuntu@$i:/opt/accumulo/conf/monitor
	scp -q /opt/accumulo/conf/tracers       ubuntu@$i:/opt/accumulo/conf/tracers

done

for ii in `cat /home/ubuntu/zknodes`; do
#moving zookeeper
```

```
    echo "zookeeper conf  to " $ii
    scp -q /opt/zookeeper/conf/zoo.cfg ubuntu@$ii:/opt/zookeeper/conf/zoo.cfg
done
```

13.6.3 setupData.sh

```
#!/bin/bash
sudo umount /mnt;
sudo /sbin/mkfs.xfs -f /dev/xvdb;
sudo mount -o noatime /dev/xvdb /mnt;

# change this if /dev/sdc is available
sudo mkdir /mnt2;
sudo /sbin/mkfs.xfs -f /dev/xvdc;
sudo mount -o noatime /dev/xvdc /mnt2;

sudo chown -R ubuntu /mnt
#sudo chown -R ubuntu /mnt2

sudo rm -rf /mnt/hdfs
sudo rm -rf /mnt/namenode
sudo rm -rf /mnt/mapred
sudo rm -rf /mnt/walogs

sudo mkdir /mnt/hdfs
sudo mkdir /mnt/namenode
sudo mkdir /mnt/mapred
sudo mkdir /mnt/walogs

sudo rm -rf /mnt2/hdfs
sudo rm -rf /mnt2/mapred

sudo mkdir /mnt2/hdfs
sudo mkdir /mnt2/mapred

sudo chown -R ubuntu  /mnt/hdfs
sudo chown -R ubuntu  /mnt/namenode
sudo chown -R ubuntu  /mnt/mapred
sudo chown -R ubuntu  /mnt/walogs

sudo chown -R ubuntu    /mnt2/hdfs
sudo chown -R ubuntu    /mnt2/mapred
```

13.6.4 etchosts.sh

```
#!/bin/bash
pssh  -h  /home/ubuntu/allnodes -t 100 sudo bash  /home/ubuntu/testhosts.sh
```

13.6.5 testhosts.sh

```
!/bin/bash
hostname | awk '{split($1, parts, "-"); print parts[2]"."parts[3]"."parts[4]"."parts[5]"\tip-"parts[2]"-"parts[3]"-"parts[4]"-"parts[5]".us-west-
1.compute.internal\tip-"parts[2]"-"parts[3]"-"parts[4]"-"parts[5]  > "/etc/hosts" }'

#shutdown reboot – delete this line on the controller node (acc-master)
sudo shutdown –r now
```

13.6.6 swappiness-spread.sh

```
#!/bin/bash
count=0
for i in `cat /home/ubuntu/allnodes`; do
   ssh -o "StrictHostKeyChecking no" $i "echo; rm -rf /home/ubuntu/.ssh/known_hosts; touch /home/ubuntu/.ssh/known_hosts";
    ssh  $i sudo sysctl vm.swappiness=0; ((count++)); echo $count  $i
    ssh  $i sudo sysctl fs.file-max=65536;
done
```

13.6.7 conf-zk.sh

```
#!/bin/bash
for ii in `cat /home/ubuntu/zknodes`; do
#moving zookeeper
    echo "zookeeper conf  to " $ii
```

```
        scp -q /opt/zookeeper/conf/zoo.cfg ubuntu@$ii:/opt/zookeeper/conf/zoo.cfg
done
```

13.6.8 copy-parallel.sh

```
#!/bin/bash
count=0
for i in `cat /home/ubuntu/first`; do
echo "copying to " $i
#moving hadoop
        scp -q /opt/hadoop/conf/core-site.xml ubuntu@$i:/opt/hadoop/conf/core-site.xml
        scp -q /opt/hadoop/conf/mapred-site.xml ubuntu@$i:/opt/hadoop/conf/mapred-site.xml
        #scp -q /opt/hadoop/conf/hdfs-site.xml ubuntu@$i:/opt/hadoop/conf/hdfs-site.xml
        scp -q /opt/hadoop/conf/masters ubuntu@$i:/opt/hadoop/conf/masters
        scp -q /opt/hadoop/conf/slaves ubuntu@$i:/opt/hadoop/conf/slaves

#moving accumulo
        scp -q /opt/accumulo/conf/accumulo-site.xml ubuntu@$i:/opt/accumulo/conf/accumulo-site.xml
        scp -q /opt/accumulo/conf/masters ubuntu@$i:/opt/accumulo/conf/masters
        scp -q /opt/accumulo/conf/slaves ubuntu@$i:/opt/accumulo/conf/slaves
        scp -q /opt/accumulo/conf/gc ubuntu@$i:/opt/accumulo/conf/gc
        scp -q /opt/accumulo/conf/monitor   ubuntu@$i:/opt/accumulo/conf/monitor
        scp -q /opt/accumulo/conf/tracers       ubuntu@$i:/opt/accumulo/conf/tracers

    scp /home/ubuntu/copy-parallel-duper.sh  ubuntu@$i:/home/ubuntu/copy-parallel-duper.sh
    ((count++))
    echo $count "file moved"
    scp /home/ubuntu/file$count  ubuntu@$i:/home/ubuntu/file
done
pssh -h /home/ubuntu/first   -t 5000 .   /home/ubuntu/copy-parallel-duper.sh
```

13.6.9 copy-parallel-duper.sh

```
#!/bin/bash
for i in `cat /home/ubuntu/file`; do
echo "copying to " $i
#moving hadoop
        scp -q /opt/hadoop/conf/core-site.xml ubuntu@$i:/opt/hadoop/conf/core-site.xml
        scp -q /opt/hadoop/conf/mapred-site.xml ubuntu@$i:/opt/hadoop/conf/mapred-site.xml
        scp -q /opt/hadoop/conf/hdfs-site.xml ubuntu@$i:/opt/hadoop/conf/hdfs-site.xml
        scp -q /opt/hadoop/conf/masters ubuntu@$i:/opt/hadoop/conf/masters
        scp -q /opt/hadoop/conf/slaves ubuntu@$i:/opt/hadoop/conf/slaves
#moving accumulo
        scp -q /opt/accumulo/conf/accumulo-site.xml ubuntu@$i:/opt/accumulo/conf/accumulo-site.xml
        scp -q /opt/accumulo/conf/masters ubuntu@$i:/opt/accumulo/conf/masters
        scp -q /opt/accumulo/conf/slaves ubuntu@$i:/opt/accumulo/conf/slaves
        scp -q /opt/accumulo/conf/gc ubuntu@$i:/opt/accumulo/conf/gc
        scp -q /opt/accumulo/conf/monitor   ubuntu@$i:/opt/accumulo/conf/monitor
        scp -q /opt/accumulo/conf/tracers       ubuntu@$i:/opt/accumulo/conf/tracers
done
```

13.7 Amazon EC2 Scripts

13.7.1 get-running-instances-default.sh

```
./ec2-describe-instances --region us-west-1| grep running
```

13.7.2 proc-instances-domain.sh

```
awk '/INSTANCE/ { split($5,parts,"-"); split(parts[5],subb,"."); split($5,parts2,".");
        print $4
        # print parts[2]"."parts[3]"."parts[4]"."subb[1]"\t"$5"\t"parts2[1]
}'  instances
```

13.7.3 proc-instances-IP.sh

```
awk '/INSTANCE/ { split($5,parts,"-"); split(parts[5],subb,"."); split($5,parts2,".");
        print $15
        # print parts[2]"."parts[3]"."parts[4]"."subb[1]"\t"$5"\t"parts2[1]
}'  instances
```

13.7.4 start-inst-proc.sh

```
!/bin/bash
```

```
_list=`awk '/INSTANCE/ { print $2  }'  instances`
ec2start  --region us-west-2  $_list
#ec2start  $_list
```

13.7.5 stop-inst-proc.sh

```
#!/bin/bash
_list=`awk '/INSTANCE/ { print $2  }'  instances`
```