2013 IEEE International Congress on Big Data

Benchmarking Apache Accumulo BigData Distributed Table Store Using Its Continuous Test Suite

Ranjan Sen
Booz Allen Hamilton
720 Olive Way, Suite 1200
Seattle, WA 98074
Sen_Ranjan@bah.com

Andrew Farris
Booz Allen Hamilton
304 Sentinel Drive
Annapolis Junction, MD 20701
Farris_Andrew@bah.com

Peter Guerra
Booz Allen Hamilton
304 Sentinel Drive
Annapolis Junction, MD 20701
Guerra_Peter@bah.com

Abstract—**In this paper, we present results of benchmarking Apache Accumulo distributed table store using the continuous tests suite included in its open source distribution. The continuous test suite contains tests that build and traverse a very large linked list, implemented via a simple table-row indexing mechanism. This underlying design provides insight for developing applications dealing with complex relationship among data sets as typically found in graph analytics applications. The benchmark study investigated sustained continuous mode stress testing and identified optimum configurations for very high-throughput data ingest, sequential and random query operations. Apache Accumulo also has the unique feature of cell level data access security, and the benchmark evaluates the processing overhead for this feature. We also tested high-speed table data verification and validation. These benchmark tests were run on a large cluster optimized for large-scale analytics and we present the performance figures for Apache Accumulo found in the study.**

Keywords- BigData; Benchmark, Scalable Table Store; BigTable Implementation; NoSQL; Apache Accumulo

I. INTRODUCTION

Large volumes of streaming and history data, in semi-structured and unstructured forms, have proliferated and impacted every aspect of the information industry. Government departments such as the defense, healthcare, energy and science and technology are facing the so-called, Big Data challenge [1]. Data mining and analytic data management that refers to querying a data store for use in business planning, problem solving and decision support [2] is often the common solution thread in various data processing challenges faced by these agencies and departments.

Distributed key/value table stores, also known as scalable table stores, provide a lightweight, cost-effective, scalable and available alternative to traditional relational databases [3, 4]. Today, scalable table stores, such as BigTable [4], Amazon Dynamo [5], Apache HBase [6], Apache Cassandra [7], Voldemort [8], and Apache Accumulo [9], are becoming an essential part of the Internet services. These are used for high volume data-intensive applications, such as business analytics and scientific data analysis [10, 11]. In some cases they are available as a cloud service, such as Amazon's SimpleDB [12] and Microsoft's Azure SQL Services [13], as well as application platforms, as in Google's AppEngine [14] and Yahoo's YQL [15].

The ingest and query support in distributed table stores defines a data serving system that provides online insert, update, read access to data, as opposed to a batch system such as Hadoop [16] or relational OLAP systems that are generally backend support to serving workloads [17].

A benchmark needs to be relevant to an application domain [18]. A benchmark for data mining and analytics for the Hadoop batch-processing environment was given in [19]. There is growing interest in benchmarking data serving systems in general including in the context of processing complex relationships in data implemented as indexed structures, such as graphs [20]. In this paper we present benchmark results for the Apache Accumulo data serving system that address this goal. Our study used the continuous tests available with the open source distribution of Apache Accumulo. We ran the benchmark on a cluster of up to 1000 machines and studied the results of performance and scalability of Apache Accumulo. We have not compared other table store products.

In section II, we discuss the problem of benchmarking Apache Accumulo in the context of its architecture and features. In section III we describe the benchmark tests and the rationale for using it. In section IV we present the results of running the benchmarks on the EMC/Greenplum Analytic Workbench (AWB) cluster [21].

II. BENCHMARKING APACHE ACCUMULO

In benchmarking a distributed table store as a data serving system we are primarily interested in evaluating the performance and scalability of ingest and query throughputs. The benchmark tests need to generate workloads composed of suitable distribution of the ingest and query data serving operations. The standard benchmark, known as the Yahoo! Cloud Serving Benchmark (YCSB) [17] and its extension, YCSB++ [22], apply a uniform set of tests to multiple scalable table stores. The results of using YCSB in a small cluster for Cassandra, HBase, Yahoo!'s PNUTS [23] and a simple shared MySQL implementation [24] were given. YCSB++ examined the advanced features of Apache HBase and Apache Accumulo, such as server side programming, available in both, and cell level security, available only in Apache Accumulo. However, their tests were run on relatively small clusters. Six server-class machines and multiple multi-core client machines were used 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 and update operation modified one of the fields. The five advanced features examined were weak consistency, bulk insertions, table pre-splitting, server-side filtering and fine-grained access control. Experiments were conducted for bulk insertion using Hadoop MapReduce. The experiments measured how high speed ingestions are affected by the policies of managing splits and compaction. Six million rows of an Apache Accumulo table were split into various numbers of partitions and completion times were found. Worst-case performance estimation was obtained for the security feature of Apache Accumulo by using unique Access Control List (ACL) for each key and a larger size for the data cell.

Benchmarks specific to BigTable, Cassandra and Hypertable [23, 24] are known. In the performance and scalability evaluation of BigTable, as reported in [4], the operations of interest were sequential and random reads and writes on tables of known sizes. The Hypertable benchmark was based on this work. Netflix developed the write oriented benchmark stress tool for Cassandra in Amazon's EC2 instances [27]. With 60 client instances this test generated 1.1 million writes per second and was complete in two hours creating a 7.2G table of records.

A. *Apache Accumulo*

Apache Accumulo is based on Google's BigTable design with addition of two unique features. One feature is the iterator framework that embeds user-programmed functionality (server-side programming) into different Log-Structured Merge Tree (LSM-tree) stages [28]. The second is the cell-level security that enables fine-grain data access control.

A distributed table, containing sorted rows of key-value pairs, shown in Figure 1(a), is partitioned into tablets and distributed to the tablet servers across the cluster. The key consists of a row id, column key and timestamp. Column keys consist of separate column family, qualifier and visibility elements. The visibility field is used for the cell-level security feature. Each tablet contains a range of rows and is assigned to a single tablet server. A single row of the table as defined by a row ID is never split across multiple tablets. Table features such as locality groups, constraints, bloom filters, and iterators are available [9]. 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.

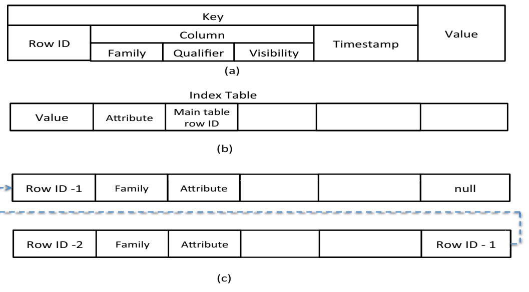

Figure 1. Apache Accumulo Key-Value pairs.

A tablet server typically manages many tablets. It receives writes to these tablets from clients, persists writes to a write-ahead log and sorts new key-value pairs in memory (*memtable*), periodically flushing sorted key-value pairs to new files, called RFiles, 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 all the files it has created and of the sorted in-memory store. When a read operation arrives the tablet server search the *memtable* as well as in the in-memory indexes associated with the table in HDFS to find the relevant values. If clients are performing a scan, many key-value pairs are returned to the client in order from the *memtable* and RFiles via a merge-sort process. 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.

When the *memtable* reaches a certain size the tablet server writes out the sorted key-value pairs to a file in HDFS. In order to manage the number of RFiles per tablet, the tablet server periodically performs *minor compactions* and *major compactions* of the files in a tablet server.

B. *Index Structure*

The table rows as key-value pairs provide a fast way to look up by a key item as attribute given by the value of a column qualifier of a row. In order to support lookups by more than one attribute of an entity, additional indexes can be built.

Consider a main table whose rows are key value pairs that represent these entities. To build an index to the rows based on attribute values construct an index table with the value as the row ID and the columns as the attribute name and row IDs of the main table, as shown in Figure 1(b). Storing the Row-ids in the column qualifier rather than in the value allows having more than one row id to be associated to a value.

It is possible to create linked data structure by considering a row as a list element and its value field as the reference to the row that corresponds to the previous or next list element in the list. A column qualifier is an attribute that relates a pair of list elements. This has been shown for a two element linked list in Figure (c). The value field of the row

(Row ID-2) is the row ID (Row ID-1) for the other element. It is straightforward to build complex indexed structures such as graphs using this basic scheme.

C. *High Speed Ingest and Query*

In a distributed key-value table store, when a table is first created it consists of just one tablet. This is the extreme situation with no scope for parallelism of table operations. As more data is ingested into the table the table size increases. When the table size crosses a limit, a configurable parameter called the table split threshold, new tablets are created from the original tablet and distributed to separate tablet servers. This takes place recursively for all tablets as more and more data is ingested and the table increases in size. The process helps in increasing operation parallelism and is the key feature of all scalable table store architectures. However, there is an overhead cost involved in table split process.

A pre-split table is a table that has already been split into tablets each of which stores a distinct subset of the rows and distributed in the cluster. Pre-splitting a table in this way helps in avoiding the split overhead that is incurred for a non pre-spit table at runtime during ingest. If we store X rows in the table, we can create X/p tablets, via a p-way partition, where the i-th tablet, $0 \leq i \leq \frac{X}{p} - 1$, will have row range of $\{ip+1, ip+2 \ldots (i+1)p\}$, where for simplicity, assume the rows of the table are $\{1, 2 \ldots X\}$. If there are N nodes in the cluster then, in an even distribution of these tablets, each node will host p/N tablets.

III. THE APACHE ACCUMULO CONTINUOUS TEST SUITE

The key feature of the Apache Accumulo Continuous test suite available with the open source distribution is the support of an uninterrupted or continuous mode operation on a very large indexed linked list. This list is created by the ingest test as the test table. The two other tests, walker and the batch walker tests, can scan and perform sequential and random reads of the linked list created by the ingest test. The tests can also be configured to estimate the overhead for cell-level security. There is also a Hadoop Map-Reduce based high-speed test to verify the table consistency. Other tests in the suite can perform tests on failure recovery, which, however, has not used in this study.

A. *The Ingest Test*

The ingest test uses a three step iterative method to build the linked list.

1. *Generate a set of list elements.*
2. *Generate a second set of list of elements that link to each element of step 1 as the head*
3. *Create one linked list by combining the lists generated in step 2*

In step 1, a linked list with $n_1 = flushInterval$ of elements are created, as rows with random row ID in the test table. Each element created has a null value for its previous element (the table row corresponding to an element has null value field). Let these rows be denoted by $\{x_1, x_2, \ldots x_{n_1}\}$.

In step 2, for each of these rows $(x_i,)$ where, $1 \leq i \leq n_1$, a fixed number $n_2 = maxDepth$ of new rows with randomly generated row IDs, $\{y_{i1}, y_{i2}, \ldots y_{in_2}\}$ are created, where for each created row (y_{ij}), $1 \leq j \leq n_2$, (y_{i1}) links to (x_1), (y_{i2}) links to (y_{i1}) …. and (y_{in_2}) links to (y_{in_2-1}). In step 3, additional n_1-1 links from each of (x_i) to $(y_{(i+1)n_2})$, for are added. The table is flushed as n_1 rows are inserted. This happens at the end of step1, and n_2 times in step 2. The table is also flushed at the end of step 3 after n_1 updates of existing rows as the linked list is connected.

The schematic view of the process is shown in Figure 2 where each box represents an element in the list and an arrow corresponds to a link to the previous element. At the end x_{n1} is the head of the entire list and distinguished by having a null value for the forward link.

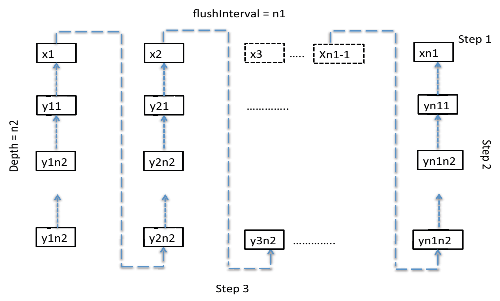

Figure 2: Continuous Ingest Test Schematic Diagram

An element stored as a row of the test table has a random row ID (between 0 and the maximum of Java long type), a random column family and qualifier (between 0 and maximum for a Java int type) and a value that contains the row ID of a previously generated row as an element.

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

Figure 3: Two rows corresponding to adjacent list elements.

An example of two rows generated by the continuous ingest test is shown in Figure 3. The first sixteen digits correspond to the row-id. The column family and column qualifier follow, separated by a colon. Next, an empty visibility field is contained within the pair of brackets. The timestamp field is not shown. After the brackets is the value field, which is split into four parts separated by colons. The third part is the index given by the row-id that references another row in the table. In the example, the row with row-id of 0020efc13153756f is pointing to the row 11af847f1e67c681. The row with row id 11af847f1e67c681 has an empty value for its link to previous row.

For each test client the ingest test logs provide information on the elapsed time between consecutive flush operations on the table as well as number of rows entered in the table.

B. *The Walker Test*

In the walker test, sequential and random scan and read operations are executed. The linked list, or more generally the graph, represented in the table created by the continuous ingest test is randomly traversed. This happens in two steps: a) scan and b) walk. In the scan step the test selects a random starting row, and scans (sequentially read) a range or set of consecutive table rows starting from it. Rows are scanned using Apache Accumulo Scanner API [9]. The consecutive set of rows is defined using an Apache Accumulo Range instance. The random selection of the starting row corresponds to a random read operation from the table. The value fields of the rows scanned are extracted and added into an intermediate buffer. In the walk step a value item is selected from the buffer, and if it is not null, the row the value item contains (previous row) is used to start a new cycle of scan and walk. If there is no candidate non-null value item in the buffer for selection in the walk step, a new row is selected randomly and the iteration starts off with a new scan followed by walk. A schematic diagram of the walker test is shown in Figure 4.

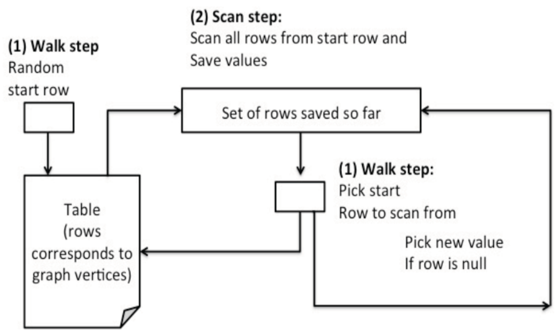

Figure 4: Schematic diagram of the Walker test.

The walker test can be customized by several parameters. These include time delay between scan/walk cycles (default is 10 milliseconds), which may be thought of as representing computation load in an application. The other parameters are the range for the generation of the random row IDs, the size of memory buffer to use and the number of threads. The time spent in scan of table rows to generate candidate rows from the randomly selected row is logged. The time for generating a random starting row is also logged.

C. *The Batch Walker Test*

The batch walker test is a parallel version of the walker test. It has the same two steps as in the walker test: scan and walk, but the walk step is actually performed in parallel.
It uses the Apache Accumulo BatchScanner API [9] to perform the parallel walks in different sets of consecutive rows, each defined as Apache Accumulo Range instances used by the parallel walk threads. These Range instances are provided as an array to the BatchScanner. This array of Range instances is created in the scan step as follows.

First a set, called batch, of rows is acquired using a Scanner that uses a Range instance that starts from a randomly selected row of the table. This is identical to the scan of the walker test. Then, the array of Range instances is created where each element is given by a Range instance that starts from a distinct row from the batch.

Multiple threads perform walks in parallel. In the walk step a thread uses a distinct Range from the array of Range instances. It randomly selects a row from the range as in the walker test and extracts the previous row from the value field of the row. It then repeats the cycle from the previous row found.

Similar to the continuous walker test the time delays between cycles of scan and walk operations can be configured. The *batchsize* parameter controls the size of the batch of rows to select a random row from in the scan step. The number of query threads can also be configured. The default batch size and query threads are 10,000 bytes and 16 respectively.

For each walker and batch walker test client the walk and batch walker logs provide the count of vertices traversed and the times for selecting a start vertex, scanning from it and extracting the adjacent vertices (previous elements) and completing a traversal as it reaches a vertex with no outgoing edge leading to adjacent vertex.

D. *Test for Cell-level Security*

Cell level security in Apache Accumulo incorporates access control using ACL to the value field in a row. A test for estimating expected overhead for checking cell-level authorization can be configured in the ingest and walker tests[1]. The test is based on the random selection of different authorization properties provided in two HDFS files. The visibilities, authorizations and authorization symbols are defined in these files. The visibility values are randomly chosen from the file and added to a row during an ingest operation and it is verified in a walk operation.

E. *Table Verification*

The table verification test runs a Hadoop MapReduce job to check the integrity of the table created by the ingest test. In the mapper method the verification test outputs key-value pairs from each test table row for the reduce method. In this key-value pair, the key is the row ID of the previous row obtained from the value field of the current row, and the value is the current row itself. So, for each row this gives the information about which row it is a previous row of. 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.

The Apache Accumulo has the AccumuloInputFormat class that provides the input interface to a table [29]. A set of Range instances, based on the number of mappers used, is obtained and assigned to the AccumuloInputFormat. The mapper input key and value are the Apache Accumulo Key and Value types. The mapper output classes for the key and value are the LongWritable and the VLongWritable classes of Hadoop [16]. The reducer reads the key values from the mapper output as a key value pair - (*vref*, (collection of

[1] Apache Accumulo 1.4.2

vrows)). The *vref* is a reference to the previous row, and the *vrows* corresponds to all the rows that have the previous row given by *vref*. The reduce method reads the values in the collection, checks if the value is legitimate and counts them. The output of the reduce method identifies the rows that are undefined, unreferenced or are inconsistent.

IV. Tests and Results

We ran the ingest test to create a test table and obtained maximum and average ingest throughputs. The walker tests were run on the table thus created to obtain maximum and average query throughputs. We ran both ingest and walker tests independently as well as together in a mixed environment, where both ingest and query were being performed simultaneously. To measure scalability, the average ingests and query throughputs were obtained for clusters of different sizes and proportionate load.

A. *Collecting Performance and Usage Data*

The Apache Accumulo continuous test suite provides a tool to collect summary results, which includes throughputs, table size, number of tablets etc. from the tests. We used the Apache Accumulo monitor service that runs on the master server to observe high-level behavior and collected performance statistics using the continuous-stats program.

B. *Workload*

An empty test table was manually created before the onset of the ingest test. An individual ingest test continuously generated rows with random row IDs and inserted them into the test table. The column family and column qualifiers were also randomly generated. The row ID is 16 bytes; and the column family and column qualifier is 8 bytes in sizes. The value field is 54 bytes with no checksum and non-null previous link.

The default values used for *flushInterval* and *maxDepth* in the ingest tests were 1,000,000 and 25 respectively. The walker test randomly selected a row to scan and walk. It repeats this operation continuously after a time delay (default 10 milliseconds). The batch walker performs the walks continuously in parallel on multiple threads and repeats after a time delay (default 180 seconds). The default batch size and number of threads are 10,000 bytes and 16 respectively.

In general, for an Apache Accumulo cluster with N-tablet servers, we had N- test clients running simultaneously, each test client co-hosted on the tablet server. The tablet servers were hosted on the same machine running the Hadoop datanode server. A test client generated 10^9 operations. In a test where one test client is run on each of the N tablet server, we have a table of size $10^9 N$ entries. As each entry is about 54 bytes, for N=1000, we get a 54 Terabyte table. In longer tests that create larger tables, we configured a test client to continue ingesting longer. In some tests we used more than one test clients per server in order to increase ingest load on the system.

C. *Test Environment and System Configuration*

We ran the benchmarks on the EMC/Greenplum Analytics workbench (AWB) [19]. The Analytics workbench is a large server cluster organized as 50 data racks, 3 core racks and 1 infrastructure rack. Each of the data racks contains 20 data nodes, and each node encompasses 12 disk drives. There are 50 data racks (20 data nodes per rack). A data node in the data rack has 12 disk drives, 48GB memory, dual Intel Westmere (Hex-core) CPU's. The master nodes in the core racks are 98GB memory, 6x 136GB, dual Intel Westmere (Hex-core) CPUs. The network switches were based on layer-2 and layer-3 Mellanox, SMC and Netgear switches. The cluster is running Greenplum Hadoop, GPHD 1.2 based on the open source Apache 1.0.3 Hadoop stack. These servers ran CentOS 6.16.

We ran these tests on Apache Accumulo configurations on the AWB clusters with a total 300, 500 and 1000 nodes. The actual number of tablet servers in these cases were 287, 488 and 984, or fewer because of failed servers, as 8 servers were used as Accumulo master, Hadoop namenode and job tracker servers, and the five zookeeper servers. As expected in large scale computing environments, other servers were simply inoperative due to hardware failures and we had used fewer tablet servers. The test clients were also hosted on the same machine that hosted the tablet servers.

D. *Results*

As stated earlier, we used the Apache Accumulo stats collector tool to compile results. We use M (million), B (billion), and T(trillion) to represent 10^6, $10^9$and 10^{12} table entries. Throughputs are given as operations per second. A table entry is 50 to 54 bytes in size depending on whether it corresponds to a head of a list or not.

We obtained results from running the tests on pre-split tables as the best-case scenario. We also obtained results for non pre-split tables as the worst-case. In practice, we expect performance to lie within these two extreme environments.

1) Ingest Tests on Pre-split Table: To examine the highest possible ingest performance we ran the ingest test on a pre-split test table. To elevate ingest rate we used three sets of ingest clients from the same host. Each test client used 1GB of buffer and 16 threads. A test client ingested 10^9 entries. So, overall we had $3N$ test clients performing the test. After the table split we reconfigured the split threshold of the table to 128G. The table split resulted in 30, 40, and 32 tablets per tablet server in the 300, 500 and 1000 node clusters respectively. The final table size for the 300, 500 and 1000 node clusters were 287B, 488B and 1T respectively and corresponds to a loads proportionate to the size of the cluster in terms of the number of tablet servers used in each case. Table I presents the result. The first three rows shows that it takes comparable times in hours for filling the table with 287B, 488B and 1T entries respectively. The fourth row presents the result for a sustained ingest on 1000 node cluster for a 26 hours. We started the sustained ingest test with 942 tablet servers. During the course of the test 11 tablet servers went down, but the ingest was continued without interruption. The final size of the test table was 7.56 T entries or 408 Terabytes.

TABLE I. PRE-SPLIT TABLE INGEST THROUGHPUTS

Cluster size	Hours	Max Ingest/Sec (M)	Average Ingest/Sec (M)
300	2.3	42	37
500	3.1	62	46
1000	2.6	165	108
1000	26.0	173	94

2) *Table Ingest on non Pre-split Table*: As stated before, this is the worst case scenario. By default a Apache Accumlo table has a table split threshold value of 1G. For a table that is not pre-split, as more and more ingests occur during the ingest test and the table size increase, more and more tablets are created by recursive tablet splits. The ingest test generates random row IDs. This makes ingest to each tablet in the cluster equally likely. In turn, this results in the tablets in a tablet server to be split almost at the same time making the entire cluster to engage in table split related operations almost at the same time. Additionally, as the test clients share common computing resource with the tablet servers, as they are co-hosted there, the originations of ingest operations also suffer.

The results for a table with split threshold of 1G, is shown in Table II below.

TABLE II. NON PRE-SPLIT TABLE INGEST THROUGHPUTS

Cluster size	Hours	Max Ingest/Sec (M)	Average Ingest/Sec (M)
300	5.5	23.82	5.14
500	5.5	12.35	2.14
1000	5.5	22.90	5.50

After the table was somewhat filled we modified the split threshold of the table in each case to 128G to avoid further splits of tablets and ran the ingest test. We observed higher average ingest throughputs, as more parallel operations were possible.

3) *Walker Tests on Pre-split Table:* The walker and batch walker tests were run on the pre-split table to examine query throughputs. Table III and IV gives the walker and batch walker query throughput. For the walker query throughputs, the average query latencies observed for the 300, 500 and 1000 node clusters were 0.39, 0.28 and 0.17 milliseconds respectively. This is similar to what we saw from a separate YCSB++ benchmark run on the same Apache Accumulo configurations. To examine sustainability we ran a walker test for 8.3 hour on a 1000 node cluster with 946 tablet servers on a pre-split table with 7.56T entries (378TBytes). The walk query latency was 0.26 milliseconds.

TABLE III. PRE-SPLIT TABLE WALKER QUERY THROUGHPUTS

Cluster size	Hours	Max Query/Sec	Average Query/Sec
300	1.00	2,880	2,559
500	1.40	3,744	3,577
1000	1.50	6,197	5,821

We ran the walker and batch walker tests on pre-split table (33-34 tablets per server). One walker and batch walkers were used per tablet server with 1GB buffer and 16 threads. The table split threshold was 128G. The table size was 750B in each case. Table 4 and 5 presents the results. Since there is no ingest no splitting occurs. The average latencies for batch walker observed for the 300, 500 and 1000 node clusters were 0.027, 0.045 and 0.045 milliseconds respectively.

TABLE IV. PRE-SPLIT TABLE BATCH WALKER QUERY THROUGHPUTS

Cluster size	Hours	Max Query/Sec	Average Query/Sec
300	1.00	231,913	36,709
500	1.00	541,902	22,025
1000	1.00	318,624	21,977

4) *Walker Tests on non Pre-split Table:* We ran the walker and batch walker tests with 1GB table split threshold. Table V and VI below present results on 300, 500 and 1000 node clusters with tables of size 280B, 635B and 314B respectively. We saw query throughputs quite similar to what we had observed for pre-split table. For the 1000 node case, however, we see a low average query throughput. This is possibly due to insufficient operation parallelism as the table size was only 314B and there was fewer tablets per tablet server.

TABLE V. NON PRE-SPLIT TABLE WALKER QUERY THROUGHPUTS

Cluster size	Hours	Max Query/Sec	Average Query/Sec
300	2.77	3,620	1,979
500	2.16	3,619	3,553
1000	2.11	2,522	713

In the walker tests, the average latencies observed for the 300, 500 and 1000 node clusters were 0.50, 0.28 and 1.4 milliseconds respectively. In the batch walker tests these values as observed were 0.032, 0.095 and 0.05 milliseconds respectively. The walker test performance and scalability was similar for both pre-split and non pre-split tables. The query throughput for the batch walker test was roughly ten times better in both cases.

TABLE VI. NON PRE-SPLIT TABLE BATCH WALKER QUERY THROUGHPUTS

Cluster size	Hours	Max Query/Sec	Average Query/Sec
300	1.00	84,041	31,103
500	1.00	157,101	10,454
1000	1.00	567,392	19,467

5) Mixed Test of Ingest and Walker: To better understand the relative impact of ingest load on query throughput we have performed a “full” and a “partial” test. Table VII gives the result for the full test on a pre-split table, when 287, 488, and 980 test clients were run. Each test client used 1G buffer and 16 threads. The table split threshold was reconfigured to 128G after splitting the table completed. The maximum and average ingest and query throughputs are given in the same column separated by ‘/’. The initial table size were 705B for the 300-node cluster and 1T for both 500 and 1000 node clusters.

TABLE VII. PRE-SPLIT TABLE FULL INGEST/WALKER THROUGHPUTS

Cluster size	Hours	Max/Avg Ingest/Sec	Max/Avg Query/Sec
300	3.50	37.6M/23M	1,404/249
500	1.44	49.4M/19.35M	15,315/988
1000	1.00	102M/55M	5,242/3,734

Table VIII gives the result for the partial test with 150 and 300 ingest clients and 488 and 980 walker test clients were simultaneously running on 500 and 1000 node clusters. The initial table size were 1.25T for both 500 and 1000 node clusters.

TABLE VIII. PRE-SPLIT TABLE PARTIAL INGEST/WALKER THROUGHPUTS

Cluster size	Hours	Max/Avg Ingest/Sec	Max/Avg Query/Sec
500	2.60	21.8M/14.5M	3,054/1,478
1000	1.1	43M/29M	8,887/6,840

Table IX and X presents the full and partial tests for non pre-split table. The same number of test clients as used for the pre-split table were used here as well. However, the buffer size was 100MB and 4 threads were used per test client. The table split thresholds were 1G for all tests. The initial table sizes were 488B, 1.5T and 416B for the 300, 500 and 1000 node cluster tests respectively. Table X gives the result for the partial test with 90, 150 and 300 ingest clients and 287, 488 and 980 walker test clients were simultaneously running. The initial table sizes were 605B, 1.7T and 424B for the 300, 500 and 1000 node cluster tests.

TABLE IX. PRE-SPLIT TABLE INGEST THROUGHPUTS

Cluster size	Hours	Max/Avg Ingest/Sec	Max/Avg Query/Sec
300	1.18	20.4M/8.8M	408/9.2
500	1.90	35.8M/25.2M	141/8.1
1000	1.00	7.5M/1.5M	1,658/65

TABLE X. PRE-SPLIT TABLE INGEST THROUGHPUTS

Cluster size	Hours	Max/Avg Ingest/Sec	Max/Avg Query/Sec
300	1.04	6.8M/5.8M	231/9.4
500	1.0	13.8M/12.7M	124/16.1
1000	0.80	7.7M/1.5M	1,230/70

We saw a rapid fall in query throughput for the mixed tests for non pre-split table compared to those for the pre-split table. This is due to the tablet splitting overhead as ingest takes place for the non pre-split table. The overheads of garbage collection, major and minor compactions are present in test with both types of tables.

6) Scalability: Scalability is the ability to perform operations as the cluster size and load are increased proportionately. The load was given by the final table size as the number of tablet servers, which were 287B, 488B and 980B respectively.

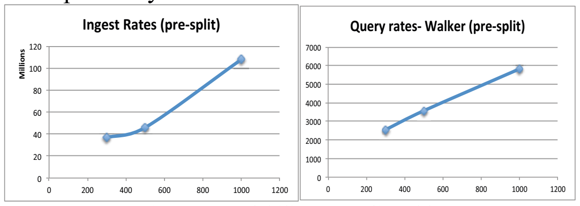

Figure 5: Scalability Pre-split Table

The table was initially empty for the ingest test. The tablets per server were determined by trial and error. We increased the ingest requests by using three test clients from each server. We also increased the buffer size and the number of threads used by the test clients. The results are from that given in Table I and III earlier. Figure 5 shows average ingest and query throughputs for pre-split table as cluster sizes and loads are increased. We see almost linear scalability for ingest and query throughputs. For a non pre-split table, we ran the ingest test for 5.5 to 6 hours on clusters with 300, 500 and 1000 nodes. The table was empty initially in each case.

We used the final table size to determine the proportional loads as in the case of pre-split table. We observed only 54B, 49B and 84B entries in the table at the end of the ingest test for the 300, 500 and1000 node clusters respectively. This is 0.18, 0.10 and 0.08 of the respective loads. In these tests each test client performed 100M operations, used 100MB buffer and 4 threads.

7) Cell Level Security (ACL): The overhead for supporting cell level security through Access Control List (ACL) is tested by running the ingest test with and without the feature on. The first figures separated by '/' give the values with no ACL and the second give the value with ACL. The variation is within 15%.

TABLE XI. PRE-SPLIT TABLE INGEST THROUGHPUTS

Cluster size	Hours	Max Ingest/Sec (M)	Average Ingest/Sec (M)
500	4.56/3.75	8.2/9.0	2.2/2.5
1000	6.03/6.00	17.5/22.6	4.1/3.5

8) Table Verification: The table verificaiton results are shown in Table XII. The verification rate was 0.35 B/min and 0.98 B/min for 200, 10000 mappers and 200, 5000 reducers respectively for the 1000 node cluster.

TABLE XII. PRE-SPLIT TABLE INGEST THROUGHPUTS

Cluster size	Table size (entries)	Mappers/Reducers	Time
300	1.5B	200/200	41 min 16 sec
1000	12.3B	200/200	34 min 46 sec
1000	1T	10000/5000	17 hours

V. CONCLUSION

The benchmark results for Apache Accumulo presented here highlights its best and the worst-case performance and scalability. The benchmark shows that Apache Accumulo can support very high levels of sustained ingest throughputs of 100 million transactions per second. This is relevant in near real time very high volume data capture scenarios. The query throughputs seen are also high and similar to what was seen in YCSB++ tests. The tests also allow observation of long-term sustainable performance of the table store.

The tests used complex indexed data structure as test table. It was run on a moderate to large cluster of up to 1000 machines.

ACKNOWLEDGMENT

We thank the Apache Accumulo development and the EMC/Greenplum AWB team for valuable support in conducting the benchmark.

REFERENCES

[1] Big Data is a Big Deal. 2012. *Office of Science and Technology Policy, Whitehouse Blog.* http://www.whitehouse.gov/blog/2012/03/29/big-data-big-deal

[2] D.J. Abadi. 2009. Data Management in the Cloud: Limitations and Opportunities. *Special Issue on Data Management on Cloud Computing Platforms, Data Engineering*. Mar 2009, Vol.32 No.1.

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

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

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

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

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

[8] Project Voldemort. http://project-voldemort.com

[9] Apache Accumulo. http://accumulo.apache.org

[10] 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), 2008.

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

[12] Amazon SimpleDB. http://aws.amazon.com/simpledb

[13] SQL Data Services/Azure Services Platform. http://www.microsoft.com/azure/data.mspx

[14] Google App Engine. http://appengine.google.com

[15] Yahoo! Query Language. http://developer.yahoo.com/yql/

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

[17] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. 2010. Benchmarking Cloud Serving Systems with YCSB, *In Proc. Of the 1st ACM Symp on Cloud Computing* (SoCC'10), June 10-11, 2010

[18] J. Gray editor. 1993. The Benchmark Handbook for Database and Transaction Processing Systems. Morgan Kaufmann, 1993.

[19] C. Bennett, R.L.Grossman, D.Locke, J.Seidman, S.Vejcik. 2010. MalStone: Towards A Benchmark for Analytics on Large Data Clouds. KDD'10 July 25-28, 2010. Washington D.C., USA.

[20] Diane J. Cook, Lawrence B. Holder, editors. 2007. Mining Graph Data. Wiley Interscience, 2007.

[21] EMC/Greenplum AWB Whitepaper, http://www.greenplum.com/sites/default/files/Greenplum-Analytics-Workbench-Whitepaper.pdf

[22] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lopez, G. Gibson, A. Fuchs, B. Rinaldi. 2011. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. *SOCC'11*, October 27-28, 2011, Cascais, Portugal

[23] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, R. Yerneni: 2008. PNUTS: Yahoo!'s hosted data serving platform, Proc. VLDB Endowment, Volume 1, Issue 2, August 2008

[24] MySQL Cluster: sql.com/tech-resources/articles/mysql-cluster-7.2-ga.html

[25] Benchmarking Cassandra scalability in AWS Over a million writes per second http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html

[26] Hypertable vs. HBase Performance Evaluation II http://hypertable.com/why_hypertable/hypertable_vs_hbase_2/

[27] Amazon EC2, http://aws.amazon.com/ec2/

[28] P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil. 1996.The Log-Structured Merge-Tree (LSM-Tree)", *Acta Informatica.* Volume 33, Issue 4, 1996, pp. 351-385

[29] Accumulo User Manual Version 1.4 http://accumulo.apache.org/1.4/user_manual/

[30] Apache Zookeeper: http://zookeeper.apache.org