Understanding Query Performance in Accumulo

Scott M. Sawyer, B. David O'Gwynn, An Tran, and Tamara Yu
MIT Lincoln Laboratory
Emails: {scott.sawyer, dogwynn, atran, tamara}@ll.mit.edu

***Abstract*—Open-source, BigTable-like distributed databases provide a scalable storage solution for data-intensive applications. The simple key–value storage schema provides fast record ingest and retrieval, nearly independent of the quantity of data stored. However, real applications must support non-trivial queries that require careful key design and value indexing. We study an Apache Accumulo–based big data system designed for a network situational awareness application. The application's storage schema and data retrieval requirements are analyzed. We then characterize the corresponding Accumulo performance bottlenecks. Queries are shown to be communication-bound and server-bound in different situations. Inefficiencies in the open-source communication stack and filesystem limit network and I/O performance, respectively. Additionally, in some situations, parallel clients can contend for server-side resources. Maximizing data retrieval rates for practical queries requires effective key design, indexing, and client parallelization.**

I. Introduction

An increasing number of applications now operate on datasets too large to store on a single database server. These Big Data applications face challenges related to the volume, velocity and variety of data. A new class of distributed databases offers scalable storage and retrieval for many of these applications. One such solution, Apache Accumulo [1] is an open-source database based on Google's BigTable design [2].

BigTable is a tabular key–value store, in which each key is a pair of strings corresponding to a row and column identifier, such that records have the format:

```
(row, column)→value
```

Records are lexicographically sorted by row key, and rows are distributed across multiple database servers. The sorted records ensure fast, efficient reads of a row, or a small range of rows, regardless of the total system size. Compared to HBase [3], another open-source BigTable implementation, Accumulo provides cell-level access control and features an architecture that leads to higher performance for parallel clients [4].

The BigTable design eschews many features of traditional database systems, making trade-offs between performance, scalability, and data consistency. A traditional Relational Database Management System (RDBMS) based on Structured Query Language (SQL) provides atomic transactions and data consistency, an essential requirement for many applications. On the other hand, BigTable uses a "NoSQL" model that relaxes these transaction requirements, guaranteeing only eventual consistency while tolerating stale or approximate data in the interim. Also unlike RDBMS tables, BigTables do not require pre-defined schema, allowing columns to be added to any row at-will for greater flexibility. However, new NoSQL databases lack the mature code base and rich feature set of established RDBMS solutions. Where needed, query planning and optimization must be implemented by the application developer.

As a result, optimizing data retrieval in a NoSQL system can be challenging. Developers must design a row key scheme, decide which optimizations to employ, and write queries that efficiently scan tables. Because distributed systems are complex, bottlenecks can be difficult to predict and identify. Many open-source NoSQL databases are implemented in Java and use heavy communication middleware, causing inefficiencies that are difficult to characterize. Thus, designing a Big Data system that meets challenging data ingest, query, and scalability requirements represents a very large tradespace.

In this paper, we study a practical Big Data application using a multi-node Accumulo instance. The Lincoln Laboratory Cyber Situational Awareness (LLCySA) system monitors traffic and events on an enterprise network and uses Big Data analytics to detect cybersecurity threats. We present the storage schema, analyze the retrieval requirements, and experimentally determine the bottlenecks for different query types. Although we focus on this particular system, results are applicable to a wide range of applications that use BigTable-like databases to manage semi-structured event data. Advanced applications must perform ad hoc, multi-step queries spanning multiple database servers and clients. Efficient queries must balance server, client and network resources.

Related work has investigated key design schemes and performance issues in BigTable implementations. Although distributed key–value stores have been widely used by web companies for several years, studying these systems is a relatively new area of research. Kepner, *et al.* propose the Dynamic Distributed Dimensional Data Model (D4M), a general purpose schema suitable for use in Accumulo or other BigTable implementations [5]. Wasi-ur-Rahman, *et al.* study performance of HBase and identify the communication stack as the principal bottleneck [6]. The primary contribution of our work is the analysis of Accumulo performance bottlenecks in the context of an advanced Big Data application.

This paper is organized as follows. Section II discusses the data retrieval and server-side processing capabilities of Accumulo. Section III contains a case study of a challenging application using Accumulo. Then, Section IV describes our performance evaluation methodology and presents the results. Finally, Section V concludes with specific recommendations for optimizing data retrieval performance in Accumulo.

This work is sponsored by the Assistant Secretary of Defense for Research and Engineering under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

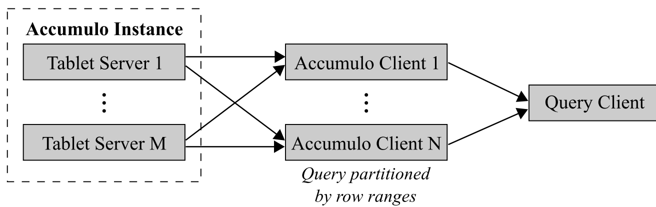


Fig. 1. In a parallel client query, M tablet servers send entries to N Accumulo clients where results are further processed before being aggregated at a single query client.

Primary Table

	Src.	Dest.	Bytes	Port
13-01-01_a8c8	Alice	Bob	128	
13-01-02_c482	Bob	Carol		80
13-01-02_7204	Alice	Carol		8080
13-01-03_5d86	Carol	Bob	55	21

Index Table

	13-01-01_a8c8	13-01-02_c482	13-01-02_7204	13-01-03_5d86
Alice|Src.	1		1	
Bob|Dest.	1			1
Bob|Src.		1		
Carol|Dest.		1	1	
Carol|Src.				1

Fig. 2. In this example, network events are stored in a primary table using a datestamp and hash as a unique row identifer. The index table enables efficient look-up by attribute value without the need to scan the entire primary table.

II. Data Retrieval in Accumulo

A single key–value pair in Accumulo is called an *entry*. Per the BigTable design, Accumulo persists entries to a distributed file system, generally running on commodity spinning-disk hard drives. A contiguous range of sorted rows is called a *tablet*, and each server in the database instance is a *tablet server*. Accumulo splits tablets into files and stores them on the Hadoop Distributed File System (HDFS) [7], which provides redundancy and high-speed I/O. Tablet servers can perform simple operations on entries, enabling easy parallelization of tasks such as regular expression filtering. Tablet servers will also utilize available memory for caching entries to accelerate ingest and query.

At query time, Accumulo tablet servers process entries using an iterator framework. Iterators can be cascaded together to achieve simple aggregation and filtering. The server-side iterator framework enables entries to be processed in parallel at the tablet servers and filtered before sending entries across the network to clients [8]. While these iterators make it easy to implement a parallel processing chain for entries, the iterator programming model has limitations. In particular, iterators must operate on a single row or entry, be stateless between iterations, and not communicate with other iterators. Operations that cannot be implemented in the iterator framework must be performed at the client.

An Accumulo client initiates data retrieval by requesting scans on one or more row ranges. Entries are processed by any configured iterators, and the resulting entries are sent from server to client using Apache Thrift object serialization middleware [9], [10]. Clients deserialize one entry at a time using client-side iterators and process the entries according to application requirements. Simple client operations may include generating a list of unique values or grouping and counting entries.

Unlike a RDBMS, Accumulo queries often require processing on both the server side and client side because of the limitations of the iterator framework. Fortunately, tablet servers can handle simultaneous requests from multiple clients, enabling clients to be parallelized to accelerate queries. A simple way to parallelize an Accumulo client is to partition the total row range across a number of client processes. Each Accumulo client processes its entries, and then results can be combined or reduced at a single query client. Figure 1 shows how data flows in a query with parallel Accumulo clients.

In a multi-step query, an Accumulo client receives entries and uses those results to initiate a new scan. For example, using an index table to locate rows containing specific values requires a two-step query. Suppose each row in the primary table represents a persisted object, and each column and value pair in that row represent an object attribute. In the index table, each row key stores a unique attribute value from the primary table, and each column contains a key pointing to a row in the primary table. Figure 2 illustrates a notional example of this indexing technique. Given these tables, an indexed value look-up requires a first scan of the index table to retrieve a set of row keys, and a second scan to retrieve those rows from the primary table. In a RDBMS, this type of query optimization is handled automatically.

III. Case Study: Network Situational Awareness

Network Situational Awareness (SA) is an emerging application area that addresses aspects of the cybersecurity problem with data-intensive analytics. The goal of network SA is to provide detailed, current information about the state of a network and how it got that way. The primary information sources are the logs of various network services. Enterprise networks are large and complex, as an organization may have thousands of users and devices, dozens of servers, multiple physical locations, and many users connecting remotely. Thus, network logs exhibit the variety, velocity and volume associated with Big Data. The data must be efficiently stored and retrieved to support real-time threat detection algorithms as well as forensic analysis.

The various data sources capture different types of network events. For example, an event may represent a user logging in to a device or a device requesting a web page through a proxy server. Each type of event has a set of attributes associated with it. LLCySA currently stores more than 50 event types in an Accumulo database instance. A data enrichment process can create additional event types by combining information from multiple captured data sources.

Recall, Accumulo is a tabular key–value store in which keys contain a row and column identifier. In LLCySA, each event type is stored in a separate table. A row in an event table corresponds to a particular event occurrence. Columns store the

various attributes of the event occurrence. For example, web proxy traffic is stored in a dedicated Accumulo table. Each row represents a specific web request and contains many columns corresponding to the event's attributes, such as time, requested URL, source IP address, and dozens of other properties.

LLCySA also applies three optimization techniques at ingest time in order to accelerate future queries. First, the row identifier contains a reversed timestamp, encoded to lexicographically sort in descending order (such that the most recent events sort to the top). This feature enables efficient queries constrained by time range, and enables the most recent events to be returned to the user first. Second, each event table has a corresponding index table to support efficient queries based on column values. In the index table, the row identifiers begin with values concatenated with column names, and each column points to a row in the primary event table. Without such an index table, finding a particular value would require scanning the entire event table. This indexing technique would be equivalent to the following SQL:

```
CREATE INDEX field_index
  ON event_table (field, event_time)
```

for each event attribute, `field`. Figure 3 shows the key scheme for the event and index tables. Note that Accumulo keys support column families (for locality grouping), cell versioning, and cell access control. For simplicity, these components of the entry keys are not shown in Figure 3.

The final optimization is *sharding*, a partitioning technique that distributes rows across database servers. LLCySA uses random sharding in the event tables to ensure database scans will always have a high degree of parallelism by avoiding hot spots corresponding to time ranges. Sharding in Accumulo is accomplished by pre-pending a shard number to the start of the row key. Shard number for an event is determined based on a hash of the event's attributes, resulting in an essentially random assignment, uniformly distributed among the allowed shard numbers. We instruct Accumulo to distribute these shards across tablet servers by pre-splitting tablets based on shard number ranges. As the database grows, the exact mapping between shard number and tablet server may change, but the uniform distribution will be largely maintained. Scanning for a single time range in the sharded event tables requires specifying a key range for each valid shard number. This adds complexity to the Accumulo clients but results in a higher degree of scan parallelism.

Accumulo and the LLCySA storage schema enable high-performance parallel data ingest. We have previously demonstrated data ingest rates in excess of four million records per second for our 8-node Accumulo instance, with speedup for up to 256 client processes parsing raw data and inserting records [11]. These ingest rates on the LLCySA system are roughly an order of magnitude faster than insert rates for traditional relational databases reported on the web [12], and the Accumulo architecture offers significantly more scalability.

Network SA also requires high-performance data retrieval for a variety of queries. Simple queries may extract all or some of the columns for events in some time range. Use cases for these simple scan-based queries include graph formation or listing active devices over some time range. Other queries find the set of events where a column contains a particular value (i.e., the equivalent of a SQL SELECT query with a WHERE clause). These queries may utilize the index table to identify applicable event rows. Another common query creates a histogram for event occurrences over a time range. The next section identifies the bottlenecks of each of these types of query.

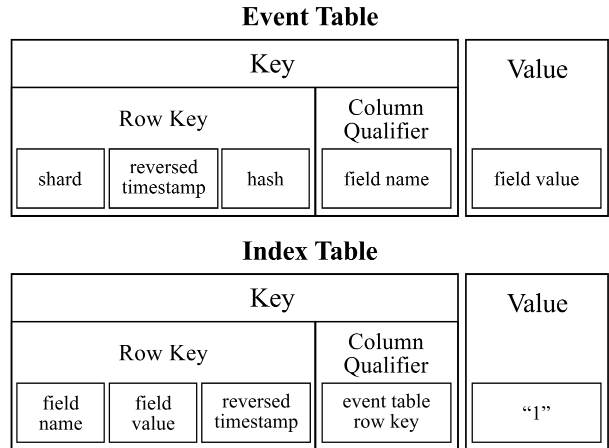


Fig. 3. The LLCySA storage schema uses two tables per event type and carefully designed keys to support efficient queries.

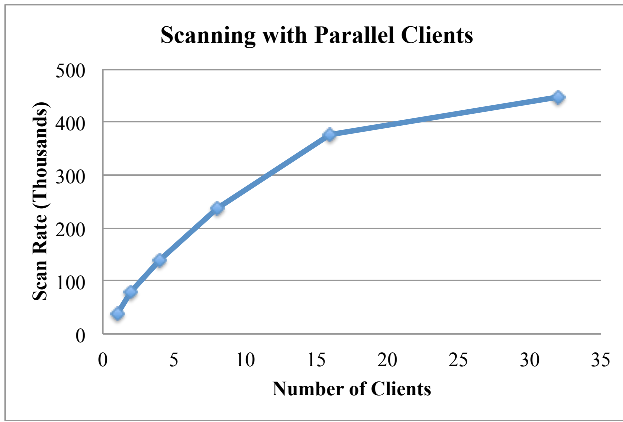
IV. Performance Results

To quantify Accumulo data retrieval performance, we run four types of queries on the LLCySA `web_request` table. Rows of the table have over 50 columns (e.g., `event_time`, `source_ip`, and `domain_name`) corresponding to the various event attributes captured in the proxy server log file. In each experiment, the number of Accumulo clients is varied, with the parallel clients partitioned by time range. For example, if a one-hour time range were partitioned across four clients, each client would request a contiguous 15-minute chunk of that hour. The sharding scheme ensures data requested by each client will be distributed uniformly among the tablet servers. The key performance metrics for the query experiements are query *speedup* (how many times faster the parallel clients complete the query compared to a single client) and *scan rate* (the total number of entries per second received by clients).

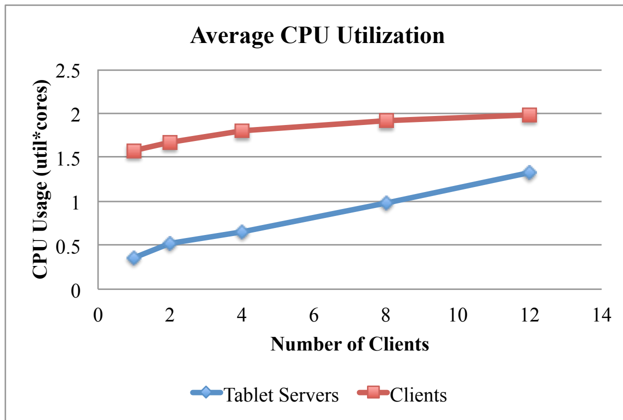
Experiments are performed on a 16-node cluster managed by the Grid Engine scheduler. Each server node contains dual AMD Opteron 6100 12-core processors, 64 GB RAM, 12 TB RAID storage, and nodes are interconnected with 10 Gbps Ethernet. The first 8 servers comprise a single distributed Accumulo 1.4 database instance. The remaining 8 servers are used as grid compute nodes. To execute parallel queries, clients are dispatched onto compute nodes by the scheduler. Experiments are managed by a custom benchmarking framework, which monitors CPU and network utilization on each server and performed parameter sweeps.

Query 1: Unfiltered Scan

In this query, a full scan is performed over a time range of web request data with no server-side filtering. Every entry within the time range is returned to a client. The total time



(a)



(b)

Fig. 4. Query 1 results. (a) Scan rate increases with number of clients. (b) Client CPU usage stalls at 2.0.

range is partitioned across a variable number of parallel clients. The clients receive entries and count them, but perform no other processing. The web request records are sharded randomly across all tablet servers, so each parallel client will receive entries from all 8 tablet servers. Figure 4 (a) shows that scan rate increases with the number of clients, with diminishing returns for higher numbers of clients as scan rate approaches about 500,000 entries per second. Figure 4 (b) shows average CPU usage across clients peaks around 2.0 (i.e., 2 fully utilized cores out of 24), while server CPU usage continues to increase linearly with the number of clients. These trends suggest that Query 1 is bound primarily by the ability of the clients to receive entries from the network and de-serialize the messages. Although these queries are communication-bound, average network utilization for the 12-client case is 29.8 Mbps, more than 300x below the network capacity. Therefore, scan rate is limited by communication overhead rather than network bandwidth.

Query 2: Server-Side Filtering

Now, records are filtered by the tablet servers. *Acceptance rate* refers to the fraction of entries read at the tablet server that are sent to the client. A custom server-side iterator is used to vary acceptance rate as an independent variable. By comparison, Query 1 had an acceptance rate of 1.0. The number of clients, n, is also varied. Figure 5 (a) shows that for high acceptance rates (right side of x-axis), scan rate varies with number of clients, which indicates scan rate is bound by the clients' ability to receive entries (as seen in Query 1). However, for low acceptance rates (left side of x-axis), the total scan rate is independent of the number of clients.

In Figure 5 (b), *entries handled per second* refers to the rate at which entries are read from disk and passed through the server-side iterator chain. As acceptance rate decreases (going right to left along the x-axis), the total number of entries handled peaks at a rate mostly independent of client count. Figure 5 (c) shows the average CPU usage at client and server. Together, these three plots demonstrate that data retrieval is client bound for high acceptance rates and server bound for low acceptance rates, with the crossover point around 1% for this particular dataset and cluster. For server-bound scans, a sustained disk read rates of 80–160 MB/s were observed, consistent with expected read performance but below the maximum performance of the underlying RAID. Thus we conclude scans with low acceptance rates are I/O bound by HDFS read rates.

Query 3: Histogram

Counting occurrences of unique values or binning values to form a histogram are common queries for database applications. While Accumulo provides some server-side capability for aggregating results in the iterator framework, many of these count-type operations must be performed at the client. For example, consider this SQL query that creates a histogram of web requests to a particular domain name over the time range $[t_1, t_2)$, assuming timestamps have been previously binned and stored in the `bin` field:

```
SELECT bin, COUNT(bin) FROM web_requests
WHERE event_time >= t1 AND event_time < t2
  AND domain_name = "example.com"
GROUP BY bin;
```

Figure 6 shows results for the simple histogram query, in which time stamps from web requests are binned into time ranges by parallel clients. This query differs from the SQL example above because the bins are computed by the client, rather than pre-computed and stored as a separate column. Our schema enables this particular query to be performed using only the index table since the WHERE clause depends only on the timestamp and one other field (recall the index table keying scheme in Figure 3). Measured runtime does not include the final reduce step, which would combine the counts generated by each client. However, the runtime of this final step is insignificant compared to total query runtime. Query 3 is run for two different time ranges of web request data. For the larger time range (8 hours), this query achieves increasing speedup for up to 5 clients before contention for the index table eventually lowers the scan rate.

Query 4: Index Look-up

Finally, Query 4 looks at parallelization of a query using both tables. This is two-step query, in which the index table is used to find a set of rows containing a particular attribute name and value pair. The index table also supports conditions on time range with no additional computation. After the first step completes, the clients have a set of keys corresponding to

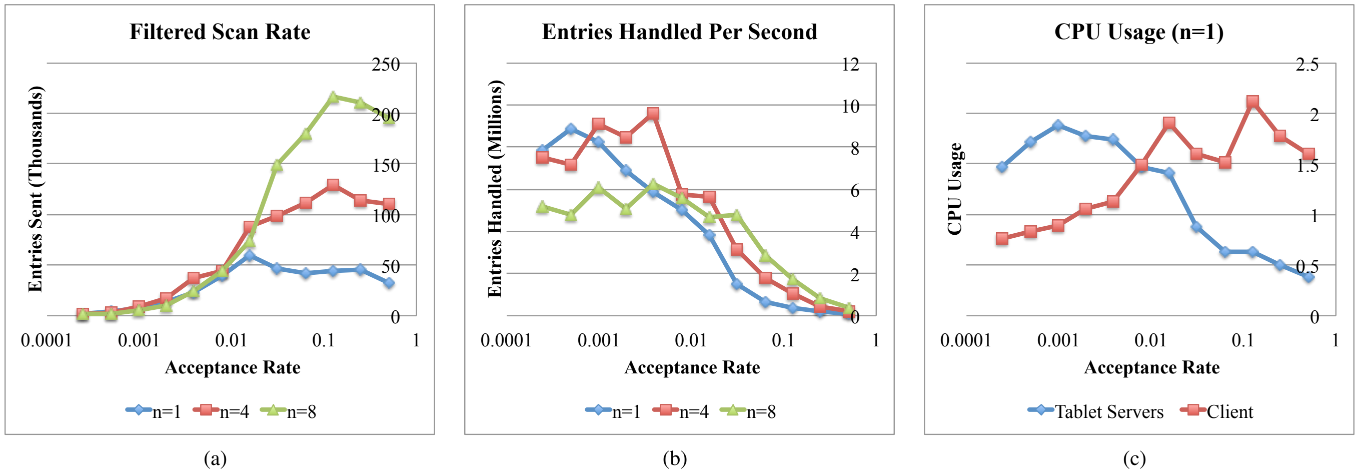


Fig. 5. Query 2 results showing (a) scan rate, (b) entry handling rate, and (c) client and server CPU usage as a function of filter acceptance rate, where n is the number of Accumulo clients.

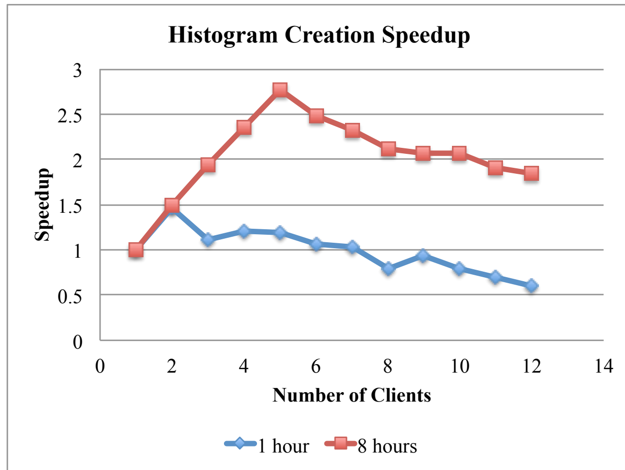


Fig. 6. Query 3 results show speedup for parallel histogram creation queries.

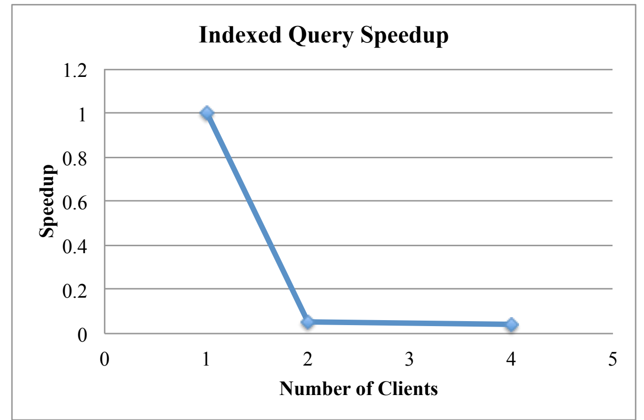


Fig. 7. Query 4 results show speedup for parallel queries using index table.

rows in the event table. The clients then issue scans to retrieve the desired columns from those event rows. In LLCySA, this type of query could be used to retrieve all source IP addresses that accessed a particular domain name. This query could be equivalently expressed in SQL as follows:

```
SELECT source_ip FROM web_requests
WHERE event_time >= t1 AND event_time < t2
  AND domain_name = "example.com";
```

Without an index table, this query would require a complete scan of the event table over the desired time range. If only a small percentage of traffic went to "example.com", the query would have a very low acceptance rate and would consume significant tablet server resources to complete the scan. This multi-step Accumulo query approach could also be used to accomplish JOINs across tables.

Figure 7 shows speedup for Query 4. Interestingly, parallelizing this query across multiple clients significantly degrades performance. In the case of two and four clients (as shown), the index table step comprised 1–2% of the total query runtime time. Thus the bottleneck is clearly on the second step, retrieving the set of relevant rows from the event table. During this step, each client requests a disjoint set of rows, located randomly across the tablet servers. Accumulo is unable to efficiently process these random-access requests from multiple clients. The contention for the various tablets and underlying files decimates performance.

V. Conclusions

The results for our four experimental queries help define the landscape of data retrieval performance in Accumulo. For someone with a background in traditional database systems, these results may seem unintuitive. Accumulo is a scalable distributed tabular key–value store, built on an open-source Java-based software stack. Compared to traditional databases, the Accumulo code base is orders of magnitude smaller and less mature. Moreover, developers must implement their own application-specific query planning and optimization. Parallel database management systems have demonstrated far better query performance at the scale of 10–100 nodes [13]. Already for several years, the inefficiency of popular open-source data-intensive computing platforms have been noted [14]. However, the cost, complexity, and limited flexibility of commercial parallel database solutions have pushed many users to the

Hadoop paradigm for Big Data applications. Additionally, Accumulo (and other BigTable implementations) can achieve far better ingest rates compared to other database architectures, which can be a key requirement for applications with ingest-heavy workloads.

In the first query, we observed scan rates limited by the client's ability to receive and process messages over the network. When tablet servers send all entries to a small number of clients, the query is bound by communication overhead at the client, as a small number of threads fully utilize a subset of the CPU cores and bottleneck the scan rate. Communication in Accumulo is handled by Apache Thrift, which does not saturate network bandwidth, as confirmed by benchmarks available on the web [15]. Re-architecting the communication stack in future versions of Accumulo could alleviate this bottleneck [6]. For other queries with low acceptance rates, the scan rate is server bound. In this case, the query is bound by I/O rates, and disk read rates are limited by the performance of the distributed filesystem and the storage hardware.

We also observed queries in which server-side contention for records bottlenecked scan rate performance. The histogram operation, in which multiple clients create hotspots in the index table, shows speedup for only a small number of clients. Beyond that, performance is likely bound by contention for the index tablets. Two-step queries utilizing the index table were not effectively parallelized at all. Multiple clients randomly accessing rows from the tablet servers resulted in contention that seriously degraded scan rate performance. Queries with random row-access patterns should not be parallelized.

Given these performance results, we can issue some additional recommendations to application developers. First, row keys should be selected to accelerate the application's queries. For example, in our application case study, all queries were constrained by a time range. Therefore, including a timestamp in the event table row keys improved query performance. Next, an index table should only be used when the scan acceptance rate is expected to be very low. For the LLCySA system, the critical filter acceptance rate was around 1%. This value will vary for different hardware and software configurations. Finally, an area of future work is improving client-bound scan rates by optimizing the Accumulo client. The open-source community has recently created a C++ implementation of an Accumulo client [16]. Using a lower-level language may help improve communication efficiency.

In summary, we have characterized performance of typical queries in a Big Data application. Some queries are bound by the communication stack, while others are bound by filesystem I/O. In both cases, hardware is not fully utilized by the open-source Java software stack, which could benefit from optimization. Although Accumulo offers schema-less storage, developers must carefully design row keys in order to efficiently support their application's query requirements. Accumulo and Hadoop offer easy server administration and programmability. On the other hand, traditional database systems provide advanced query planning and optimization features. No single product fits all applications perfectly. However, Accumulo can be an effective storage solution for data-intensive applications requiring high ingest rates, distributed processing, and a high degree of scalability.

References

[1] *Apache Accumulo*. [Online]. Available: http://accumulo.apache.org

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[3] *Apache HBase*. [Online]. Available: http://hbase.apache.org

[4] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 9.

[5] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz *et al.*, "Dynamic distributed dimensional data model (D4M) database and computation system," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. IEEE, 2012, pp. 5349–5352.

[6] M. Wasi-ur Rahman, J. Huang, J. Jose, X. Ouyang, H. Wang, N. S. Islam, H. Subramoni, C. Murthy, and D. K. Panda, "Understanding the communication characteristics in HBase: What are the fundamental bottlenecks?" in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 122–123.

[7] *Apache Hadoop*. [Online]. Available: http://hadoop.apache.org

[8] A. Fuchs, "Accumulo: extensions to Google's Bigtable design," March 2012, lecture, Morgan State University.

[9] *Apache Thrift*. [Online]. Available: http://thrift.apache.org

[10] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, vol. 5, 2007.

[11] C. Byun, W. Arcand, D. Bestor, B. Bergeron, M. Hubbell, J. Kepner, A. McCabe, P. Michaleas, J. Mullen, D. O'Gwynn *et al.*, "Driving big data with big compute," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–6.

[12] P. Zaitsev, "High rate insertion with MySQL and Innodb," January 2011. [Online]. Available: http://www.mysqlperformanceblog.com/2011/01/07/high-rate-insertion-with-mysql-and-innodb/

[13] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009, pp. 165–178.

[14] E. Anderson and J. Tucek, "Efficiency matters!" *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 40–45, 2010.

[15] "Java serialization benchmarking," 2013. [Online]. Available: https://github.com/eishay/jvm-serializers/wiki

[16] C. J. Nolet, *Accumulo++: A C++ library for Apache Accumulo*. [Online]. Available: https://github.com/cjnolet/accumulo-cpp