Graphulo Matrix Multiply

Dylan Hutchison*, Jeremy Kepner*, Vijay Gadepally*, Adam Fuchs†

*MIT Lincoln Laboratory
†Sqrrl

***Abstract*—todo**
filler [1]

I. Introduction

Accumulo is well-known as a high performance NoSQL database with respect to ingest and scans. The next step past ingesting and scanning is computing—running algorithms and analytics. The advantage of doing computation in a database like Accumulo is *selective access*, *data locality* and *infrastructure reuse*. Accumulo's features as a database enable fast access to data subsets and queries along indexed attributes. Further, Accumulo sits atop the physical location data is stored and cached, such that computation inside Accumulo tablet servers occurs local to data storage and avoids unnecessary network transfers, effectively moving "compute to data" in contrast to client-server models that move "data to compute." Computation within Accumulo also reuses its distributed infrastructure (such as write-ahead logging, fault-tolerant execution atop Zookeeper and horizontal scalability from masters and tablet servers), which in environments already using Accumulo, are already deployed in production clusters, saving the need to install and maintain a new distributed system for computation.

One form of computation commonly run on database data is linear algebra. Researchers in the GraphBLAS Forum [**?**] have identified a set of primitive operations that form the basis for linear algebraic algorithms useful for graphs, including Sparse General Matrix Multiplication (SpGEMM), Sparse Element-wise Multiplication (SpEWiseX), Sparse Reference of a subset (SpRef), Reduction along a dimension (Reduce), Function application (Apply) and others. Graphulo is an effort to realize the GraphBLAS primitives and enable algorithms in the language of linear algebra server-side on Accumulo [**?**].

In this paper we focus on implementing SpGEMM, a core operation at the heart of GraphBLAS. In fact, most of the other GraphBLAS primitives can be expressed in terms of SpGEMM. We can realize SpEWiseX by multiplying the transpose of the first matrix with the second and using a custom multiplication function that only acts on elements whose row from the first matches the column of the second and returns multiplied elements with the using the row and column of the second. We can realize SpRef when selecting a subset of rows by multiplying on the left with an identity matrix containing those rows, and we can realize it when selecting a subset of column by multiplying similarly on the right. We can realize Reduce by multiplying on the left or right depending on the dimension of interest by a dense vector of all ones. One can even view graph search [2] and table joins as applications of SpGEMM [**?**].

Too much detail for intro

We call our implementation of SpGEMM on Accumulo TableMult, short for multiplication of Accumulo tables. Accumulo tables have many similarities to sparse matrices, though a more exact analogy is to Associative Arrays [3].

alternate wording below:

Enrichment [**?**] is the process of taking an input data set and transforming it into a new dataset, such as by applying a function or by fusing it with another dataset. If Accumulo is restricted to traditional database use as a distributed, indexed key-value data storage service, then a "client" must query Accumulo for necessary input data, apply the enrichment, and write the new data to Accumulo. Complex enrichment processes require complex client infrastructure to process large volumes of data efficiently, resulting in the need for distributed computation engines such as Spark [**?**]. Accumulo has many of these distributed features already implemented and, when used as a database, already deployed in the production clusters. Instead of duplicating Accumulo's infrastructure and well-optimized scale-out parallelism in creating a new distributed computational system, we aim to shift computation to within Accumulo where appropriate, as well as the possibility to perform computation more local to data storage. Some would call this concept server-side computation, though in the case of Accumulo we might call it database-side computation.

A. *Paper Outline*

II. Background

A. *Accumulo Iterators*

Accumulo includes a mechanism to perform limited serer-side computation called the *iterator stack*. Iterators inside the iterator stack are objects of classes that implement the `SortedKeyValueIterator` (SKVI) interface, an interface reminiscent of but more complex than built-in Java iterators from the `java.lang` package in that they have methods to return a current entry (`getTopKey` and `getTopValue`)

Dylan Hutchison is the corresponding author and can be reached at dhutchis [at] mit.edu

This material is based upon work supported by the National Science Foundation under Grant No. DMS-1312831. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

and proceed to the next entry (`next`) until no more entries remain (`hasTop`). Iterators may hold state initialized in `init`, to which Accumulo hands options of the form `Map<String,String>` passed from the client.

Arguably, the most critical component of an iterator is its `seek` method, which instructs an iterator to jump to the beginning of a passed-in range. System iterators at the top of an iterator stack perform actual disk seeks and cache locations in memory when seeked.

During a scan, Accumulo constructs an iterator stack for each tablet whose keys overlap some portion of the scan range. These iterator stacks may run in parallel, and each is seeked to the range of keys in the current tablet, instersected with the scan range. When any call to the iterator stack returns, Accumulo may choose to destroy the iterator stack and later re-create it, passing a new seek range starting at the last key returned from `getTopKey`, exclusive. Accumulo does this when it needs to switch data sources (such as RFiles) after a compaction, when a client stops requesting data, or out of fairness to other concurrent scans.

Iterators do not have full lifecycle control in that there is no `close` method that allows an iterator to clean up its state before being destroyted. The only safe way for an iterator to use state requiring cleanup, such as opening a file or starting a thread, is for the iterator to clean up its state before returning from a method call. Ideas discussed in [4] may lax this restriction for future Accumulo versions.

III. TableMult Design

A. Matrix Multiplication

Given input matrices $\mathbf{A}$ of size $n \times m$, $\mathbf{B}$ of size $m \times p$, and operations $\oplus$ and $\otimes$ for scalar summation and multiplication, the matrix multiplication $\mathbf{A} \oplus.\otimes \mathbf{B} = \mathbf{C}$, or more shortly $\mathbf{AB} = \mathbf{C}$, defines the entries of result matrix C as

$$\mathbf{C}_{ij} = \bigoplus_{k=1}^{m} \mathbf{A}_{ik} \otimes \mathbf{B}_{kj}$$

We call intermediary results of $\otimes$ operations *partial products*.

In the case of sparse matrices, we only perform $\oplus$ and $\otimes$ operations where both operands are nonzero, an optimization stemming from the requirement that 0 is an additive identity of $\oplus$ such that $a \oplus 0 = 0 \oplus a = a$, and that 0 is a multiplicative annhilator of $\otimes$ such that $a \otimes 0 = 0 \otimes a = 0$. Sparse arithmetic is impossible without these conditions, since in that case zero elements could generate nonzero results.

In Accumulo, zero elements are entries that do not exist in a table. Entries that actually contain the value zero may exist from an operation producing a zero value (such as summing partial products 5, -3 and -2). We currently deliver no special treatment to these entries and plan on adding an optional feature that removes them when encountered.

There are two general patterns for performing matrix multiplication: inner product and outer product. We study each in terms of how it implements the $\otimes$ component of matrix multiplication, deferring the $\oplus$ operation to run on output generated from applying $\otimes$. In the pseudocode, ':' stands for "all positions" in the spirit of Matlab notation.

The more commonly used inner product method performs the following:

```
foreach $n$
    foreach $p$
        emit $\mathbf{A}_{n:} \cdot \mathbf{B}_{:p}$
```

where the operation $\cdot$ is inner (also called dot) product on vectors, which we may unfold as

```
foreach $n$
    foreach $p$
        foreach $m$
            emit $\mathbf{A}_{nm} \otimes \mathbf{B}_{mp}$
```

Inner product has an advantage of generating output "in order," meaning that all partial products needed to compute a particular element $\mathbf{C}_{np}$ are generated consecutively by the third-level for loop. We may apply the $\oplus$ operation immediately after each third-level for loop and obtain an element in the result matrix. This means that inner product is easy to "pre-sum," an Accumulo term for applying a Combiner locally before sending entries to a remote but globally-aware table combiner. It is also adventageous that inner product generates entries sorted by row and column, which allows inner product to be used in standard iterator stacks that require sorted output.

Despite its order-preserving advantages, we chose not to implement inner product because it requires multiple passes over input matrix $\mathbf{B}$. This is because the second-level for loop, which runs over all the columns of matrix $\mathbf{B}$, must run for each row of matrix $\mathbf{A}$ (first-level for loop). Performance measurements proved this a major bottleneck since, in the case of Accumulo and with the assumption that we cannot fit all of matrix $\mathbf{B}$ in memory, it mandates multiple reads from disk.

The outer product method performs the following:

```
foreach $m$
    emit $\mathbf{A}_{:m} \times \mathbf{B}_{m:}$
```

where the operation $\times$ is outer (also called tensor or Carteisan) product on vectors, which we may unfold as

```
foreach $m$
    foreach $n$
        foreach $p$
            emit $\mathbf{A}_{nm} \otimes \mathbf{B}_{mp}$
```

Outer product emits partial products in chaotic, unsorted order. This is due to moving the n and p for loops, both responsible for determining partial product position, below the top-level m for loop.

On the other hand, outer product only requires a single pass over both input matrices. This is because the top-level m for loop fixes a dimension of both matrix $\mathbf{A}$ and $\mathbf{B}$. Once we finish processing a full column of $\mathbf{A}$ and row of $\mathbf{B}$, the column and row are finished; there is no need to read them again (i.e., we never need to restart the top-level for loop).

We may speed the outer product algorithm by storing rows of table $\mathbf{B}$ in memory, which eliminates the need to re-read rows of $\mathbf{B}$ for each run of the third-level for loop. In the case of Accumulo, we can even eliminate the need to hold a row in memory by using deepCopy SKVI methods to store "pointers" to the beginning of the current row m of table $\mathbf{B}$. However, this strategy may come at the cost of extra disk seeks, and so we leave testing its performance to future work, for now storing the current row of table $\mathbf{B}$ in memory.

Since m is the second dimension of matrix $\mathbf{A}$, we implement SpGEMM to operate on the transpose $\mathbf{A}^\intercal$ of input table $\mathbf{A}$.

B. TableMult Iterators

We use three new server-side iterators to implement SpGEMM: RemoteSourceIterator, TwoTableIterator and RemoteWriteIterator. We place these iterators on a scan of Table $\mathbf{B}$. The scan itself emits no entries except for a smidgeon of "monitoring entries" that inform the client about TableMult progress. Instead, the scan on table $\mathbf{B}$ reads from table $\mathbf{AT}$ by opening a Scanner and writes to result table C by opening a BatchWriter, all within the scan's iterator stack. See Figure 1 for an illustration.

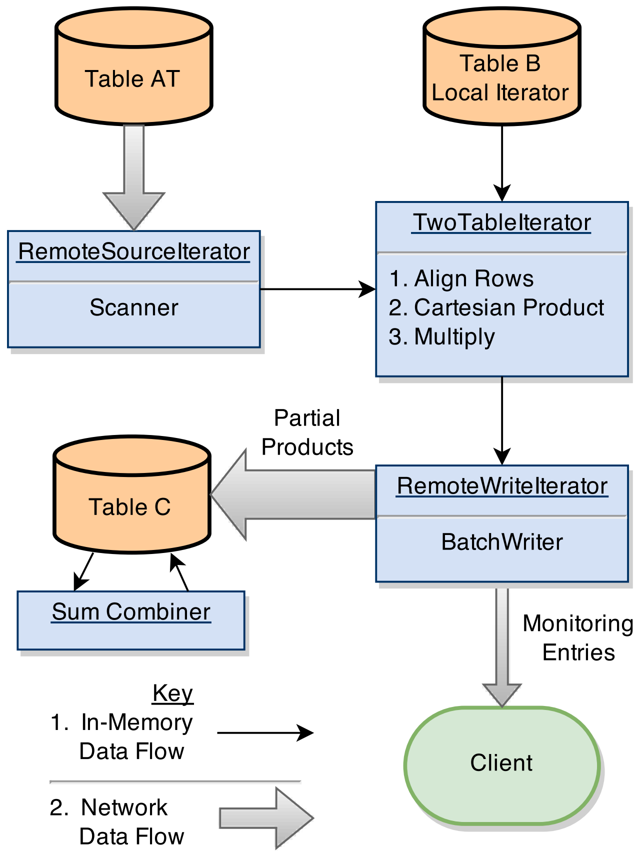

Fig. 1: Data flow through the TableMult iterator stack

1) RemoteSourceIterator: RemoteSourceIterator contains a Scanner that scans an Accumulo table (not necessarily in the same cluster) with the same range it is seeked to. Clients pass connection information, including zookeeper hosts, timeout, username and password, to a RemoteSourceIterator via iterator options in the form of a `Map<String,String>`.

A client may also specify rows and columns to scan via a rowRanges and colFilter option, if not scanning the whole table. We encode the rowRanges option in "D4M syntax," consisting of delimitted row keys and the ':' character to indicate ranges between row keys. In our encoding, a,:,b,d,f,:," translates to the ranges "[a,b\0), [d,d\0), [f,+∞)." The comma used as a delimitter in the example may be any character not present in a row key. Rows consisting of a single ':' are not allowed. The keys of the ranges are empty aside from the row, which is ok for our purpose since SpGEMM operates on the Cartesian product of entire rows.

ColFilter is defined similarly to rowRanges, except that we do not allow ranges to occur in the colFilter because Accumulo does not index columns. We may enable ranges anyway in the future by using an Accumulo Filter iterator subclass to only return the entries within a column range set. Such a feature would require scanning of every column, which does affect performance if bottlenecked on table reads instead of writes. We encourage users to maintain a tranpose table for cases requiring column indexing.

Scanning a subset of rows in a table is crucial for queued analytics, since those analaytics operate on a graph subset. However, for simplicity of performance evaluation, our performance measurements do not use rowRanges or colFilter and instead multiply whole table inputs.

2) TwoTableIterator: TwoTableIterator reads from two iterator sources, one for $\mathbf{AT}$ and one for $\mathbf{B}$, and performs the core operations of the outer product algorithm in three phases:

1) Align Rows. Read entries from table $\mathbf{AT}$ and $\mathbf{B}$ until they both reside at the beginning of a matching row, or until one runs out of entries. We skip non-matching rows as they would be multiplied by an all-zero row in the opposite table and generate all zero patrial products.
2) Cartesian product. Read both matching rows into an in-memory data structure. Initialize an iterator that emits pairs of entries from the the matching rows' Carteisan product.
3) Multiply. Pass pairs of entries to the $\otimes$ function and emit the result.

In implementing the $\otimes$ operation as ordinary real number multiplication, we decode Accumulo value byte arrays into Strings and then into `java.math.BigDecimal` objects. BigDecimal enables multiplication to work for very large and very precise real-valued numbers, though it may be cumbersome for small integer multiplication. We use it anyway since it is not our primary bottleneck. We use a similar BigDecimal decoding for our $\oplus$ summing operation.

3) RemoteWriteIterator: RemoteWriteIterator writes entries to a remote Accumulo table using a BatchWriter created on `init`. Entries do not have to be in sorted order because

Accumulo sorts incoming entries as part of its standard ingest process. Like RemoteSourceIterator, the client passes connection information for the remote table via iterator options.

Barring extreme events such as exceptions in the iterator stack or tablet server death, we designed RemoteWriteIterator to maintain correctness, such that entries generated from RemoteWriteIterator's source will be written to the remote table once. One way to accomplish this is by performing all BatchWriter operations within a single function call before ceding thread contol back to the Accumulo tablet server. We do this in the `seek` method, streaming all entries from the RemoteWriteIterator's source (within the seek range) into a BatchWriter at once, by repeatedly calling the `getTop` and `next` methods of its source and `flush`ing afterward.

We completely close the BatchWriter in a `finalize` method. The JVM does not guarantee that objects' `finalize` method will be called before it is garbage collected, but in our experience, `finalize` has been called in every non-extreme circumstance and we guarantee correctness even if it is not called. Alternatives such as using `java.lang.ref` classes and closing BatchWriters in place of flushing them are open.

A performance concern remains in the case of performing a TableMult over a subset of the input tables' rows that consists of many disjoint ranges, such as 1M "singleton" ranges that span one row each. It is inefficient to flush the BatchWriter before returning from each seek call, which happens once per disjoint scan range. In order to accomodate this case, we "transfer seek control" over the desired row range subset from the Accumulo tablet server to the RemoteWriteIterator by passing the range objects through iterator options encoded as a D4M range string (see above), as opposed to the usual method of passing the range objects to the `setRanges` method of the BatchScanner on table $\mathbf{B}$. The RemoteWriteIterator can then traverse all of the ranges in the desired subset before returning from a seek. In the case of multiple tablets for table $\mathbf{B}$, the RemoteWriteIterator running on each tablet only handles the portion of ranges that intersects with the range of keys in its tablet.

We include an option to BatchWrite the transpose of the result table in place of or alongside the result table. This enables maintenance of transpose indexing for users using the D4M Schema, as well as facilitating TableMult chaining of one TableMult after another.

4) Lazy $\oplus$: We lazily perform the $\oplus$ portion of matrix multiplication (i.e., summing of partial products) by placing a Combiner subclass implementing $\oplus$ logic on the result table at scan, minor and major compaction scopes. Thus, $\oplus$ occurs sometime after the RemoteWriteIterator writes partial products to the result table, yet necessarily before any entry from the result table may be seen such that we always achieve correctness. This could happen in part when Accumulo flushes the result table to a new RFile, when Accumulo compacts RFiles together, or when a client scans the result table.

The key algebraic requirement for implementing $\oplus$ inside a Combiner is that $\oplus$ must be associative and commutative. These properties allow us to perform $\oplus$ on any ordering of the partial products, which is chaotic by the nature of the outer product algorithm and often applied to some portion of the partial products.If we truly had an $\oplus$ operation that required seeing all partial products at once, we would have to either gather the partial products at the client or perform a full major compaction.

5) Monitoring: RemoteWriteIterator never emits any entries to the client by default. One downside of this approach is that clients cannot precisely track the progress of a TableMult operation. The only information they would have are scan and write rates from the Accumulo monitor and what partial products are written to the result table so far by scanning the result table.

We therefore implemented a monitoring option on RemoteWriteIterator that occassionally emits entries back to the client at "safe" points in the middle of a TableMult, that is, at points when the iterator stack may recover its state if Accumulo destroys, later re-creates and re-seeks it to a range starting at its last emitted key (exclusive). Stopping after emitting the last value in the outer product of two rows is naturally safe, since we may place that row key in the emitted monitoring key and know, in the event of an iterator stack rebuild, to proceed to the next matching rows. We are also experimenting with an encoding that permits stopping in the middle of an outer product by encoding the column family and qualifier of the rows in the outer product in the emitted monitoring key.

We control the frequency of emitting monitoring entries by specifying iterator options in terms of how many entries before emitting or how long a duration between emitting monitor entries. We encode the number of entries processed so far in the monitoring entry value. Between the key and value, the client can see how far the TableMult operation progressed in terms of number of emitted entries and progress in scanning row keys from tables $\mathbf{AT}$ and $\mathbf{B}$.

In order for the client to receive monitoring entries before the TableMult completes, we must change the `table.scan.max.memory` paramter of table $\mathbf{B}$ to as small a value as possible, say one byte. This forces Accumulo to send back entries from the BatchWriteIterator immediately as they are released. We may restore the previous parameter value (default 512KB) once the TableMult compeletes. If a proposed change [5] is applied to future Accumulo versions, we will be able to change the tablet server scan cache on a per-scan basis rather than a per-table basis that affects concurrent scans on table B.

Put Java signature of TableMult call?

IV. Performance

We evalutate our TableMult implementation with a weak scaling experiment, measuring rate of computation as problem size increases. We define problem size as the number of nodes in random input graphs measure rate of computation as the number of partial products processed per second. We also perform the same tests when input and output tables are split into two tablets each, which allows Accumulo to scan and write to them in parallel.

We use D4M as a baseline to compare our Graphulo TableMult implementation against, because a user's next best alternative to TableMult is to read the input graphs from Accumulo to a client, compute the matrix product at the client, and insert the result back into Accumulo.

D4M stores tables in Matlab as Associative Array objects, written as Assocs for short. D4M Associative Array multiplication is highly optimized and runs very quickly, considering the entire tables are stored in memory. D4M's bottleneck is therefore on reading data from Accumulo and especially on writing results back to Accumulo. We consequently expect TableMult to perform better than D4M because it avoids the need to transfer data out of Accumulo in order to process it.

We also expect TableMult to succeed on larger graph instances than D4M, since TableMult uses a streaming outer product algorithm that does not require storing input tables in memory. An alternative D4M implementation is to mirror TableMult's streaming outer product algorithm. This would enable D4M to run on larger problem sizes but decrease performance. We therefore imagine the whole-table D4M implementation as an upper bound on the best performance achievable when performing the multiplication with a client outside Accumulo's infrastructure.

We use the Graph500 random graph generator [**?**] to create random input matrices. The generator creates graphs with a power law structure, such that node degree is very high for the first row of the input table (or in terms of graphs, first vertex) has many edges and further rows have an exponentially decreasing number of edges. The graph generator takes a SCALE and EdgesPerVertex parameter and creates graphs with 2^{SCALE} vertices and EdgesPerVertex $\times$ 2^{SCALE} edges. We fix EdgesPerVertex to 16 and use SCALE to vary problem size.

The following procedure outlines our performance test for a given SCALE and number of tablets (1 or 2).

1) Generate two random graphs with different random seeds and insert them into Accumulo tables using D4M.
2) In the case of two tablets, identify an optimal split point for each input graph and set the input graphs' table splits equal to that point. "Optimal" here means a split point that nearly evenly divides an input graph into two tablets.
3) Create an empty output table and pre-split it with the first input table's split. The split will not be optimal for the output table because the matrix product has a different degree distribution than that of the input tables, but it is close enough for the purposes of our test.
4) Compact the input and output tables so that Accumulo redistributes the tables' entries into the assigned tablets.
5) Run and time the Graphulo TableMult operation, multiplying the transpose of the first input with the second input table.
6) Create, pre-split and compact a new result table for the D4M comparison as in step 3 and 4.
7) Run and time the D4M equivalent of TableMult, using the following steps:
   a) Scan both input tables into D4M Associative Array objects in Matlab memory.
   b) Convert the string values from Accumulo into numeric values for each Assoc.
   c) Multiply the transpose of the first Assoc with the second Assoc.
   d) Convert the result Assoc back to String values and insert it into Accumulo.

We conduct the tests on a laptop with 16GB RAM and 2 Intel i7 processors at 3GHz running Ubuntu 14.04 linux. We use single-instance Accumulo 1.6.1, Hadoop 2.6.0 and ZooKeeper 3.4.6. We allocate 2GB of memory to the Accumulo tablet server initially, allowing increases in 500MB step sizes, 1GB for Accumulo's native in-memory maps and 256MB for data and index caches.

We chose not to run with more than two tablets per table because it would result in too many threads a single laptop could handle. Each additional tablet can potentially result in the following threads:

1) Table AT server-side scan thread
2) Table AT client-side scan thread (running from a RemoteSourceIterator)
3) Table B server-side scan/multiply thread (running the TableMult iterator stack)
4) Table B client-side scan thread (running from the client iniating the operation; mostly idle)
5) Table C server-side write thread (running with a combiner implementing $\oplus$)
6) Table C client-side write thread (running from a RemoteWriteIterator)
7) Table C server-side minor compaction threads

We look forward to extending our test to a larger Accumulo cluster that can handle more degrees of parallelism.

Figure 2 displays test results. The best results we could achieve are flat horizontal lines, indicating that we maintain the same level of operations per second as problem size increases.

We could not run the D4M test comparison past SCALE 15 because we cannot fit the input and output tables in memory.

One reason we see a decrease in performance results at larger problem sizes is that Accumulo needs to minor compact the result table in the middle of the TableMult. This in turn triggers the $\oplus$ Combiner on the table, which sums paartial products written to the result table so far. Thus, one explanation for the rate decrease is that our rate estimate in terms of partial products per second does not include the summing operations Accumulo must perform when it needs to minor compact.

V. Discussion

Our initial design operated the iterator stack on a full major compaction. We chose to operate the iterator stack on a scan instead because major compactions experience a delay on the order of seconds before Accumulo schedules them, slightly

Fig. 2: Data flow through the TableMult iterator stack

bumping latency, and because opening a BatchWriter inside an iterator at major compaction presents a small chance for deadlock. Deadlock may occur if the major compaction iterators triggered enough minor compactions such that they exhaust every thread in the compaction thread pool. This leaves open the chance that a major compaction thread would block on a minor compaction thread in order for the BatchWriter to write entries, while in turn the minor compaction thread blocks on the major compaction since major compactions take a higher thread pool priority than minor ones.

One frequent computaional pattern is to create intermediary tables in the middle of an algorithm that are not needed once the algorithm completes. For example, we may have a series of TableMult operations for which only the final TableMult is needed as output. It is therefore wasteful to write the intermediary TableMult results to disk if we have room to store them in memory. Sometimes one can restructure an algorithm to minimize the use of intermediary tables, but a better solution would be to realize a notion of in-memory "temporary tables" in Accumulo. We leave constructing this notion to future work.

Similarly, it is also useful to *pipeline* TableMults in an algorithm by starting the process that acts on the result of a TableMult before the TableMult finishes. Unfortunately, the outer product algorithm cannot guarantee that all partial products for a particular element are written to the result table before the algorithm finishes, since it writes results in chaotic order. The inner product algorithm would be easier to pipeline. We therefore treat TableMult operations as barriers for operations acting on a TableMult's result.

We implemented TableMult logic inside three separate iterators connected via the SortedKeyValueIterator interface. One technique for increasing performance is *loop fusion*, combining separate components into one that performs everything in one pass. We chose to keep our iterators separate because it opens them to reuse in other server-side operations and the iterator's processing (including necessary decoding and encoding) is not our bottleneck. We will be bound by BatchWrite time no matter how well we optimize the iterator processing.

VI. Related Work

Our outer product method could also have been implemented in MapReduce [?] or its successor YARN [?]. In fact, there is a natural analogy from how we process data using Accumulo infrastructure to methods in MapReduce.

In the map phase, we map matching rows of the input tables to a list of partial products generated from their outer product. We realize the map phase in Accumulo via the Scanners and the TwoTableIterator. In the shuffle phase, we send partial products to the correct tablet of the result table via machinery in the Accumulo BatchWriter (using data in the metadata table). In the reduce phase, partial products are lazily combined by Accumulo combiners that implement the $\oplus$ operation.

We have a hunch that a MapReduce implementation using the AccumuloInputFormat and AccumuloOutputFormat will outperform our implementation in terms of throughput for large input tables (i.e. whole-table analytics) but not latency for moderate input tables (i.e. queued analytics), due to spinup delay MapReduce jobs experience. In either case, using MapReduce requires features in Hadoop clusters outside the Accumulo ecosystem, which may not be an option for some user environments. It would be interesting to directly compare a MapReduce implementation in the future.

Cannon's algorithm, other SpGEMM

Stored Procedures a la MySQL

To measure performance, we used techniques similar to those from Google Dapper [**?**]. We instrument sections of code inside try-finally statements, recording the start time before entering those sections and ensuring we record stop times inside the finally portion. We store total time spent inside these code section "spans" inside thread-local storage, along with statistics on how many times we enter a span and the minimum and maximum amount of time spent inside a span. These statistics showed us what code was being traversed more often than expected and what portions of code took the longest.

VII. Conclusions

Acknowledgment

The authors wish to thank the entire Graphulo team at MIT CSAIL and MIT Lincoln Laboratory. We also thank the GraphBLAS contributors and National Science Foundation for their generous ongoing support of this program.

References

[1] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout *et al.*, "Achieving 100,000,000 database inserts per second using accumulo and d4m," *IEEE High Performance Extreme Computing*, 2014.

[2] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.

[3] J. Kepner and V. Gadepally, "Adjacency matrices, incidence matrices, database schemas, and associative arrays," in *International Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014.

[4] D. Hutchison. (2015) Iterator redesign. [Online]. Available: https://issues.apache.org/jira/browse/ACCUMULO-3751

[5] J. Vines. (2012) Scanner should support batch size specified in bytes. [Online]. Available: https://issues.apache.org/jira/browse/ACCUMULO-261

Performance Numbers

			Graphulo 1 Tablet		D4M 1 Tablet		Graphulo 2 Tablets		D4M 2 Tablets	
SCALE	#PartialProducts	nnz(C)	Time (s)	Rate	Time (s)	Rate	Time (s)	Rate	Time (s)	Rate
10	804,989	269,404	2.79	288,319.84	2.94	273,575.94	2.14	375,408.75	2.96	271,817.07
11	2,361,580	814,644	7.59	310,885.56	8.85	266,699.94	5.22	452,262.67	8.94	264,132.83
12	6,816,962	2,430,381	21.38	318,844.63	27.30	249,633.32	15.39	442,786.38	37.25	182,983.58
13	19,111,689	7,037,007	57.82	330,486.24	155.58	122,834.05	54.16	352,846.59	180.27	106,014.96
14	52,656,204	20,029,427	160.45	328,167.63	586.62	89,761.16	136.73	385,107.45	594.49	88,572.50
15	147,104,084		502.86	292,532.77	2,510.38	58,598.13	399.24	368,452.34	2,559.24	57,479.52
16	400,380,031	163,481,262	1,475.81	271,294.29			1,230.81	325,297.94		

TABLE I: Numerical results and parameters for Figure 2