Graphulo: Server-side Matrix Multiply on Accumulo Tables

Dylan Hutchison[1], Jeremy Kepner[1,2,3], Vijay Gadepally[1,2], Adam Fuchs[4]

[1]MIT Lincoln Laboratory BeaverWorks
[2]MIT Computer Science and Artificial Intelligence Laboratory
[3]MIT Mathematics Department
[4]Sqrrl

***Abstract*—todo**

I. Introduction

Accumulo is well-known as a high performance NoSQL database with respect to ingest and scans [1]. The next step past ingesting and scanning is computing—running enrichment, algorithms and analytics.

Conducting computation within a database like Accumulo offers several advantages, assuming we can efficiently map it to the database's programming model: *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" like a stored procedure 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 load balancing tablets. Why not reuse these services already deployed in production clusters that use Accumulo, rather than install and maintain new distributed systems for computation?

One family of computation commonly run on database data is linear algebra. Researchers in the GraphBLAS Forum [2] have identified a set of primitive operations that form a 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 [3].

In this paper we focus on implementing SpGEMM, a core kernel at the heart of GraphBLAS. In fact, most 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 using the row and column of the second; we can realize SpRef when selecting a row or column subset by multiplying on the left or right with an identity matrix containing those rows or columns; and we can realize Reduce by multiplying on the left or right by a dense vector of all ones. Outside SpGEMM usage ranges from graph search [4] to table joins [5] and plenty others described in the introduction of [6].

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 [7]. Because Accumulo tables may span Hadoop clusters, we never assume a table fits in memory and instead take a streaming approach that distributes with Accumulo's infrastructure.

We are particularly interested in SpGEMM for queued analytics, that is, analytics on a selected table subset. Queued analytics allow us to maximally take advantage of Accumulo as a database by quickly accessing subsets of interest, whereas whole-table analytics usually perform better on parallel file systems such as Lustre or Hadoop. We therefore prioritize low latency over high throughput, in the best case enabling analysts to manipulate Accumulo data interactively.

We review iterator stacks, Accumulo's model of server-side computation, in Section I-A. We formally define matrix multiplication and compare inner and outer product SpGEMM mehtods in Section II-A, ultimately settling on outer product for our TableMult implementation. We show TableMult's design as Accumulo iterators in Section II-B, and we test its scalability with experiments in Section III. We conclude with a discussion on design alternatives in Section IV.

Dylan Hutchison is the corresponding author, reachable at dhutchis@mit.edu.

This material is based upon work supported by the National Science Foundation under Grant No. DMS-1312831. 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.

A. Primer: Accumulo Iterators

Accumulo's server-side programming model is the *iterator stack*, a list of objects that implement the `SortedKeyValueIterator` (SKVI) interface. At a high level, the iterator stack is like a collection of datastreams originating at Accumulo's data sources (Hadoop RFiles and cached in-memory maps), converging together in merge-sorts, flowing through each of the stack's iterators and at the end, transferring to the client over the network. Each iterator performs some operation on the stream along the way such as filtering out entries that fail a filter function, all while maintaining data emission in sorted order.

Developers add custom logic for server-side computation by creating new SKVIs and plugging them into an appropriate position in the iterator stack datastream. In return for fitting their computation in the SKVI paradigm, developers gain distributed parallelism for free as Accumulo runs the iterator stack on every relevant tablet simultaneously.

SKVIs are reminiscent of built-in Java iterators in that they have methods to return a current entry (`getTopKey` and `getTopValue`) and proceed to the next entry (`next`) until no more entries remain (`hasTop`). On the other hand, SKVIs have more power than iterators in that they may initalize state inside an `init` method, to which Accumulo passes options of the form `Map<String,String>` from the client, and that SKVIs have a `seek` method that jumps to the beginning of a passed-in range. System SKVIs at the top of an iterator stack perform actual disk seeks.

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 [8] may lax this restriction for future Accumulo versions.

Iterators are most commonly used for "reduction" operations, in the sense of transforming or eliminating entries passing through. The Accumulo community generally discourages "generator" iterators that emit new entries not present in the original data source, not because generator iterators are impossible but because they are easy to misuse and violate SKVI constraints by emitting entries out of order or relying on state that fails when Accumulo chooses to destroy and re-create an iterator stack between method calls.

II. 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 $\mathbf{C}$ as

$$\mathbf{C}(i,j) = \bigoplus_{k=1}^{m} \mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$$

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 common inner product method runs the following:

```
for $i = 1:n$
    for $j = 1:p$
        emit $\mathbf{A}(i,:) \cdot \mathbf{B}(:,j)$
```

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

```
for $i = 1:n$
    for $j = 1:p$
        for $k = 1:m$
            emit $\mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$
```

Inner product has an advantage of generating output "in order," meaning that all partial products needed to compute a particular element $\mathbf{C}(i,j)$ are generated consecutively by the third-level loop. We may apply the $\oplus$ operation immediately after each third-level 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: the second-level loop that scans over all of input matrix $\mathbf{B}$ repeats for each row of $\mathbf{A}$ from the top-level loop iteration. Under our assumption that we cannot fit $\mathbf{B}$ entirely in memory, multiple passes over $\mathbf{B}$ translates to multiple Accumulo scans that each require a disk read. We found in our performance tests that multiple scans over $\mathbf{B}$ delivered over an order of magnitude worse performance, taking over 100 seconds to multiply SCALE 11 inputs whereas the outer product method ran in under 8 seconds.

Outer product matrix multiply runs the following:

```
for $k = 1 : m$
    emit $\mathbf{A}(:, k) \times \mathbf{B}(k, :)$
```

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

```
for $k = 1 : m$
    for $i = 1 : n$
        for $j = 1 : p$
            emit $\mathbf{A}(i, k) \otimes \mathbf{B}(k, j)$
```

Outer product emits partial products in unsorted order. This is due to moving the i and j loops that determine partial product position below the top-level k loop.

On the other hand, outer product only requires a single pass over both input matrices. This is because the top-level k loop fixes a dimension of both $\mathbf{A}$ and $\mathbf{B}$. Once we finish processing a full column of $\mathbf{A}$ and row of $\mathbf{B}$, we never need to read them again (i.e., we never need to restart the top-level k loop).

In terms of memory usage, outer product works best when we can fit one of the matching row and column in memory. If neither fits in memory, then we could still run the algorithm by re-reading rows of $\mathbf{B}$ while streaming through columns of $\mathbf{A}$ as the loops suggest. Running through columns of $\mathbf{A}$ instead is possible by symmetry of i and k. We may implement the "no memory assumption" streaming appraoch in Accumulo by using `deepCopy` SKVI methods to store "pointers" to the beginning of rows from table $\mathbf{B}$ (or columns of table $\mathbf{A}$), perhaps at the cost of extra disk reads.

Because k runs along the second dimension of $\mathbf{A}$ and Accumulo uses a row-oriented data layout, we implement TableMult to operate on $\mathbf{A}$'s transpose $\mathbf{A}^\intercal$.

B. TableMult Iterators

We implement SpGEMM with three server-side iterators placed on a BatchScan of Table $\mathbf{B}$: RemoteSourceIterator, TwoTableIterator and RemoteWriteIterator.

The key idea behind the TableMult iterators is that they divert normal dataflow by opening a BatchWriter, redirecting entries out-of-band to a sink table via Accumulo's standard ingest channel that does not require sorted order. The scan itself emits no entries except for a smidgeon of "monitoring entries" that inform the client about TableMult progress. We also enable multi-table iterator dataflow by opening Scanners that read outside Accumulo tables out-of-band. Scanners and BatchWriters are standard tools for Accumulo clients; by creating them inside Accumulo iterators, we enable all the functionality of client-side processing within the Accumulo server.

In the case of SpGEMM, we use a Scanner on table $\mathbf{A}^\intercal$ and a BatchWriter on result table $\mathbf{C}$. Figure 1 illustrates TableMult's data flow.

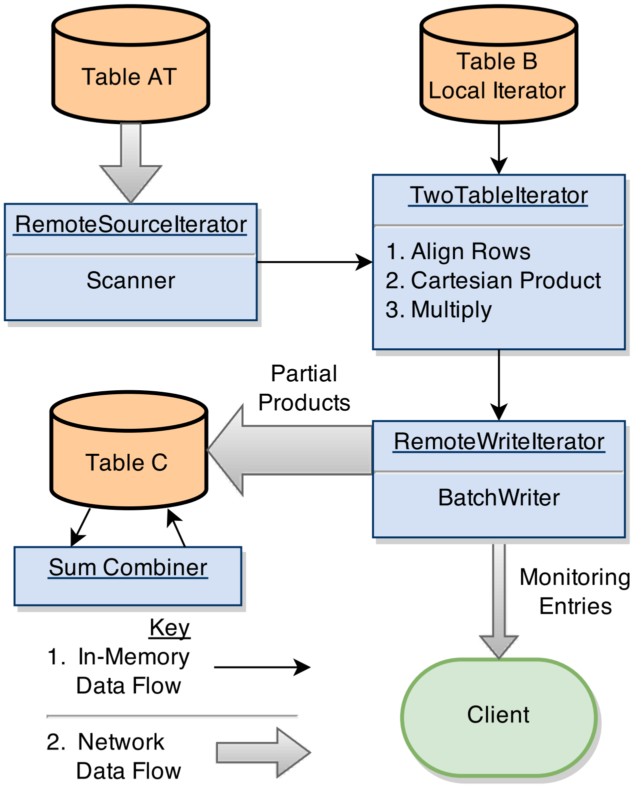

Fig. 1: Data flow through the TableMult iterator stack

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$\backslash$0), [d,d$\backslash$0), [f,$+\infty$)." The comma used as a delimitter in the example may be any character not present in a row key, and we disallow rows consisting of a single ':'. We do not include column family or deeper fields in our encoding and leave them empty, 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 requires scanning every column and will affect performance if bottlenecked on table reads. 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 experiments in Section III multiply whole table inputs.

2) TwoTableIterator: TwoTableIterator reads from two iterator sources, one for $\mathbf{A}\mathrm{T}$ 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 $\mathbf{A}^{\mathsf{T}}$ and $\mathbf{B}$ until they 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 $\otimes$ and emit results.

In implementing the $\otimes$ operation as ordinary real number multiplication and $\oplus$ as addition, 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 BigDecimal anyway since writing result entries is our bottleneck, not iterator processing.

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 thread 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 every entry from 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 calling objects' finalize method before it garbage collects them, but in our experience, the JVM called finalize in every non-extreme circumstance and we guarantee correctness even if finalize 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 TableMult over a subset of the input tables' rows that consists of many disjoint ranges, such as 1M "singleton" ranges spanning one row each. It is inefficient to flush the BatchWriter before returning from each seek call, which happens once per disjoint scan range, and a known Accumulo issue could even crash the tablet server [9]. 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 range objects to the `setRanges` BatchScanner method for table $\mathbf{B}$. The RemoteWriteIterator can then traverse all 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 handles the portion of ranges that intersects with the range of keys in its tablet.

We include an option to BatchWrite the result table's transpose in place of or alongside the result table. Writing the transpose enables maintenance of transpose indexing for those using the D4M Schema [10], as well as facilitating chaining TableMults one after another.

4) Lazy $\oplus$: We lazily perform the $\oplus$ portion of matrix multiplication (i.e., summing 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 subsets of a result element's partial products and on any ordering of them, which is chaotic by the outer product's nature. If we truly had an $\oplus$ operation that required seeing all partial products at once, we would have to either gather partial products at the client or initiate a full major compaction.

Our choice to defer the $\oplus$ operation

5) Monitoring: RemoteWriteIterator never emits 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 clients would have are scan and write rates from the Accumulo monitor, whether a scan is running, idle or queued from the tablet server, and what partial products are written to the result table so far from 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 $\mathbf{A}^\intercal$ 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 entries from the BatchWriteIterator to the client immediately as they are generated. We may restore the previous parameter value (default 512KB) once the TableMult compeletes. If a proposed change [11] 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 $\mathbf{B}$.

III. Performance

We evaluate TableMult with two variants of an experiment. First we gauge weak scaling by measuring rate of computation as problem size increases. We define problem size as the number of nodes in random input graphs and rate of computation as the number of partial products processed per second. Second we gauge strong scaling by repeating the first experiment with all tables split into two tablets, allowing Accumulo to scan and write to them in parallel.

We use D4M as a baseline to compare Graphulo TableMult performance 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 Assoc multiplication is highly optimized and runs very quickly, particularly because the entire input 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 TableMult 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 because TableMult uses a streaming outer product algorithm that does not require storing input tables in memory. An alternative D4M implementation would mirror TableMult's streaming outer product algorithm, enabling D4M to run on larger problem sizes at the cost of worse performance. We therefore imagine the whole-table D4M implementation as an upper bound on the best performance achievable when performing multiplication on an Accumulo table outside Accumulo's infrastructure.

We use the Graph500 random graph generator [12] to create random input matrices. The generator creates graphs with a power law structure, such that the first vertex has high degree and subsequent vertices have exponentially decreasing degree. 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 experiment for a given SCALE and either one or two tablets.

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 Graphulo TableMult multiplying the transpose of the first input table with the second.
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:
    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 experiments 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 add the following threads:

1) Table $\mathbf{A}^\intercal$ server-side scan thread

2) Table $\mathbf{A}^{\mathsf{T}}$ client-side scan thread,
running from a RemoteSourceIterator
3) Table $\mathbf{B}$ server-side scan/multiply thread,
running the TableMult iterator stack
4) Table $\mathbf{B}$ client-side scan thread,
running from the client initiating; mostly idle
5) Table $\mathbf{C}$ server-side write thread,
running with a Combiner implementing $\oplus$
6) Table $\mathbf{C}$ client-side write thread,
running from a RemoteWriteIterator
7) Table $\mathbf{C}$ server-side minor compaction threads

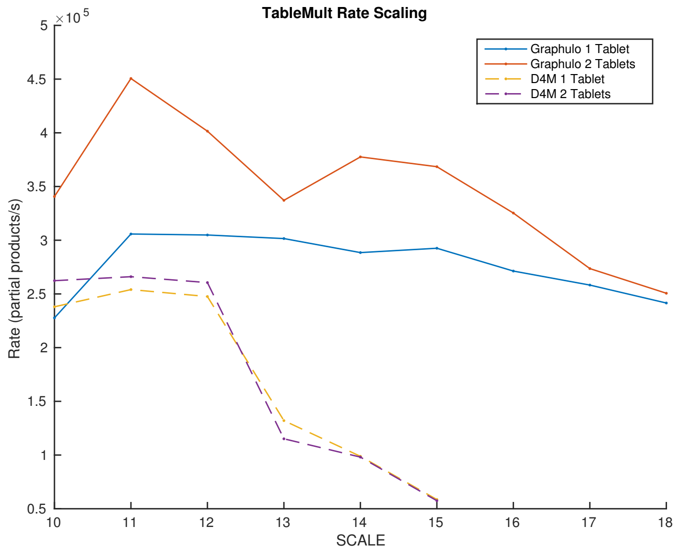

Fig. 2: Data flow through the TableMult iterator stack

Figure 2 displays test results. We could not run the D4M comparison past SCALE 15 because the input and output tables do not fit in memory.

In terms of weak scaling, 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. Graphulo roughly achieve weak scaling, although the two-tablet Graphulo curve shows some instability.

One reason we see a slightly downward trend in rate 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.

In terms of strong scaling, the best results we could achieve are two-tablet rates at double the one-tablet rates for every problem size. Our experiment shows that Graphulo two-tablet multiply rates perform as much as 1.5x better than one-tablet rates. This may be due to very high processor contention as a result of the 14 threads that may run concurrently with two tablets; in fact, processor usage hovered around 100% for all four laptop cores throughout the two-tablet experiments. We expect better strong scaling results once we move our experiment to a larger Accumulo cluster that can handle more degrees of parallelism.

IV. Discussion

A. *TableMult Design Alternatives*

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 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.

We also believe there is room to reconsider the inner product SpGEMM formulation from our initial design because they bottleneck on different operations: inner product bottlenecks heavily on scanning whereas outer product bottlenecks on writing. See Figure 3 for a pictorial comparison. At the expense of multiple passes over input matrices, inner product emits entries more efficiently in that emitted entries are in order and partial products can be summed immediately, reducing the number of entries needed to write to Accumulo to the minimum possible. Outer product reads input matrices in a single pass but emits entries out of order and has much less change to pre-sum partial products, requiring writing individual partial products to the result table. For our experiment on power law input graphs, Table I shows that outer product writes 2.5 to 3 times more entries than the minimum possible.

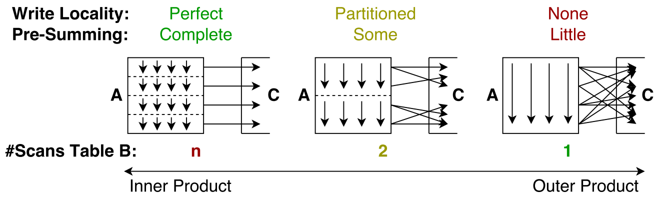

Fig. 3: Tradeoffs between Inner and Outer Product

Is it possible to blend inner and outer product algorithms and achieve better performance than either method alone? A definitive answer is future research, though we sketch one possible bridge in Figure 3. Suppose we partition the rows of $\mathbf{A}$ (columns of $\mathbf{A}^{\mathsf{T}}$) into two halves and run outer product on each separately. Each outer product requires scanning all of $\mathbf{B}$ once, for a total of two passes. In exchange we increase write locality: the top half outer product only writes to the top half of $\mathbf{C}$ and the bottom half outer product only writes to the bottom half of $\mathbf{C}$. Selecting the number of partitions along rows of $\mathbf{A}$

			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	3.53	227,700.33	3.38	238,023.69	2.36	340,721.66	3.06	262,328.46
11	2,361,580	814,644	7.72	305,789.27	9.29	254,040.68	5.24	450,554.23	8.87	266,050.05
12	6,816,962	2,430,381	22.36	304,868.98	27.53	247,568.13	16.97	401,529.20	26.16	260,553.36
13	19,111,689	7,037,007	63.37	301,550.36	144.73	132,047.17	56.68	337,136.98	166.01	115,121.57
14	52,656,204	20,029,427	182.52	288,486.51	532.91	98,808.44	139.46	377,562.34	536.79	98,094.46
15	147,104,084	58,288,789	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		
17	1,086,789,275	677,175,416	4,208.24	258,252.42			3,972.13	273,603.14		
18	2,937,549,526	966,262,286	12,161.74	241,540.15			11,720.44	250,634.66		

TABLE I: Numerical results and parameters for Figure 2

determines balance between inner product (partitions between every row) and outer product (zero partitions).

A challenge for any hybrid algorithm is mapping it to Accumulo infrastructure. We chose outer product because it most naturally fits into Accumulo, using for example iterators for one-pass streaming computation, BatchWriters to handle unsorted entry emission and Combiners to defer summation. We may realize greater performance by considering data distribution among tablet servers, for example, but such a consideration would require accessing and perhaps manipulating Accumulo's internal state and non-public API. We suggest this paper's approach as a balance between top performance and implementation stability.

B. TableMult in Algorithms

A 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.

V. Related Work

Buluç and Gilbert have studied SpGEMM rigorously, comparing parallel message passing algorithm implementations, proving theoretical bounds and plotting performance [13]. Most of the algorithms they present including Sparse SUMMA algorithms use 2D block decompositions. Unfortunately, 2D decompositions are quite difficult in Accumulo and message passing between servers even more so. In this work we use a 1D decomposition along rows with no tablet server communication other than shuffling partial products to result tablets via a BatchWriter.

Our outer product method could have been implemented in MapReduce on Hadoop or its successor YARN [14]. 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.

To measure performance, we used techniques similar to those from Google Dapper [15]. 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.

VI. Conclusions

In this work we showcase the design of TableMult, a Graphulo implementation of the SpGEMM GraphBLAS linear algebra kernal server-side on Accumulo tables. We compare inner and outer approaches for SpGEMM and show how outer product is better suited to the Accumulo iterator environment. Performance experiments show ideal weak scaling and hint at good strong scaling, although repeating our experiments on a larger cluster is necessary to confirm.

Current research is to implement the remaining GraphBLAS kernels and develop algorithms calling them, ultimately delivering a Graphulo linear algebra library as a pattern for server-side computation to the Accumulo community.

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] R. Sen, A. Farris, and P. Guerra, "Benchmarking apache accumulo bigdata distributed table store using its continuous test suite," in *Big Data (BigData Congress), 2013 IEEE International Congress on*. IEEE, 2013, pp. 334–341.

[2] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson *et al.*, "Standards for graph algorithm primitives," *arXiv preprint arXiv:1408.0393*, 2014.

[3] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for nosql databases," in *International Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2015 IEEE International*. IEEE, 2015.

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

[5] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton, "Mad skills: new analysis practices for big data," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1481–1492, 2009.

[6] A. Buluç and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," 2010.

[7] 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.

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

[9] ——. (2015) Scanning with many singleton ranges crashes tserver. [Online]. Available: https://issues.apache.org/jira/browse/ACCUMULO-3710

[10] J. Kepner, C. Anderson, W. Arcand, D. Bestor, B. Bergeron, C. Byun, M. Hubbell, P. Michaleas, J. Mullen, D. O'Gwynn *et al.*, "D4m 2.0 schema: A general purpose high performance schema for the accumulo database," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–6.

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

[12] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *Cyberinfrastructure Technology Watch*, vol. 2, pp. 1–10, 2006.

[13] A. Buluc and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[15] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," *Google research*, 2010.