From NoSQL Accumulo to NewSQL Graphulo: Design and Utility of Graph Algorithms inside a BigTable Database

Dylan Hutchison[†] Jeremy Kepner[‡§◇] Vijay Gadepally[‡§] Bill Howe[†]

[†]University of Washington [‡]MIT Lincoln Laboratory
[§]MIT Computer Science & AI Laboratory [◇]MIT Mathematics Department

***Abstract*—Google BigTable's scale-out design for distributed key-value storage inspired a generation of NoSQL databases. Recently the NewSQL paradigm emerged in response to analytic workloads that demand distributed computation local to data storage. Many such analytics take the form of graph algorithms, a trend that motivated the GraphBLAS initiative to standardize a set of matrix math kernels for building graph algorithms. In this article we show how it is possible to implement the GraphBLAS kernels in a BigTable database by presenting the design of Graphulo, a library for executing graph algorithms inside the Apache Accumulo database. We detail the Graphulo implementation of two graph algorithms and conduct experiments comparing their performance to two main-memory matrix math systems. Our results shed insight into the conditions that determine when executing a graph algorithm is faster inside a database versus an external system—in short, that memory requirements and relative I/O are critical factors.**

I. Introduction

The history of data storage and compute is long intertwined. SQL databases facilitate a spectrum of streaming computation inside the database server in the form of relational queries [1]. Server-side selection, join, and aggregation enable statisticians to compute correlations and run hypothesis tests on larger datasets [2]. The high I/O cost of geospatial queries, financial transactions, and other more complicated computation motivated the concept of stored procedures for executing custom computation in databases as early as Sybase in the 1980s [3].

The NoSQL movement marks a shift away from in-database computation, relaxing some of the guarantees and services provided by SQL databases in order to provide a flexible schema and greater read/write performance as demanded by new applications such as website indexing [4]. The Google BigTable NoSQL design in particular forsook relational operators in order to allow arbitrary row and column names, allow uninterpreted values, and provide a clear model for data partitioning and layout, all at high performance that scales with a cluster of commodity machines [5]. Several databases follow BigTable's design, including Apache Accumulo, Apache HBase, and Hypertable.

BigTable follows a pull-based model of computation called the *iterator stack*: a collection of arbitrary classes through which key-value entries flow, beginning at a table's data sources (in-memory maps and files in a backing store) and flowing through the logic of each iterator class before sending entries to a client (or a file, in the case of a compaction) after the last iterator. The iterator stack was designed for relatively lightweight computation such as filtering outdated values and summing values with the same keys. As such, many applications use BigTable systems purely for storage and retrieval, drawing on separate systems to perform computation.

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

The NewSQL movement marks another shift, this time back toward SQL guarantees and services that promise efficient in-database analytics while retaining the flexibility and raw read/write performance of NoSQL databases [4]. Engineers have several reasons to consider computing inside databases rather than in external systems:

1) To increase data locality by co-locating computation and data storage. The savings in data communication significantly improves performance for some computations.
2) To promote infrastructure reuse, avoiding the overhead of configuring an additional system by using one system for both storage and computation. Organizations may have administrative reasons for preferring solutions that use known, already integrated systems rather than solutions that introduce unfamiliar or untested new systems.
3) To take advantage of database features such as fast selective access to subsets of data along indexed dimensions. In the case of distributed databases, algorithms running inside a database obtain distributed execution for free, in return for cooperating with the database's access path.

Is it possible to implement NewSQL-style analytics within a BigTable database? A positive answer could have broad implications for organizations that use a BigTable database to store and retrieve data yet use a separate system for analyzing that data. The extent of such implications depends on the relative performance of running analytics inside a BigTable database versus an external system.

In this work we examine analytics that take the form of graph algorithms. Analysts commonly interpret data as graphs due to easy conceptualization (entities as nodes, relationships as edges), visualization, and applicability of graph theory (e.g. centrality and clusters) for gaining data insight.

One way to represent graphs is by its adjacency or incidence matrix. Matrices are an excellent choice for a data structure

because they provide properties and guarantees derived from linear algebra such as identities, commutativity, and annihilators. These properties facilitate reasoning the correctness of algorithms and optimizations before any code is written, which can save considerable developer time [6].

The GraphBLAS specification is a set of signatures for matrix math kernels that are building blocks for composing graph algorithms [7]. These kernels provide the benefits of computing with matrices while remaining amenable to optimization. In particular, we show in Section II how a critical class of optimizations, kernel fusion, remains possible in the GraphBLAS abstraction.

We focus our work on Graphulo, an effort to realize the GraphBLAS primitives inside the Apache Accumulo database. In past Graphulo work we sketched several graph algorithms in terms of the GraphBLAS kernels [8] and detailed the Graphulo implementation of Sparse Generalized Matrix Multiply [9]. Our new contributions are:

- An extension of Graphulo's model for Accumulo server-side computation to all the GraphBLAS kernels in Section II, showing that it is possible to implement general graph analytics inside a BigTable system.
- The Graphulo implementation of algorithms to compute the Jaccard coefficients (Section III-A) and the k-truss decomposition (Section III-B) of an adjacency matrix.
- Insight into the conditions that determine when it profitable to execute graph algorithms inside a BigTable system, as learned from Section IV's performance evaluation of the Jaccard and k-truss algorithms inside Accumulo and in two external main-memory matrix math systems.

Our results corroborate an emerging theme: "no one size fits all" [10]; no single system is best for every algorithm. Our evidence shows that executing an algorithm inside a BigTable database achieves better performance when data does not fit in memory or the I/O cost, in terms of entries read and written, is within an order of magnitude. Algorithms that are iterative or otherwise read and write significantly more entries in an in-database implementation run faster in a main-memory system.

Writing an algorithm in terms of the GraphBLAS frees developers to execute on any GraphBLAS-enabled system. We offer the conditions presented here to aid developers in choosing the best system for execution.

II. GraphBLAS in Graphulo

Figure 1 depicts a template of the iterator stack Graphulo uses for server-side processing. Graphulo users customize the template by calling a `TwoTable` function in the Graphulo library. Because the TwoTable call accepts a large number of arguments in order to customize the iterator stack to any desired processing, Graphulo provides a number of simpler, more specialized functions such as `TableMult` for computing an *MxM*, `SpEWiseSum` for computing an *EwiseAdd*, and `OneTable` for computations that take a single input table.

Once a client configures the iterator stack by calling one of Graphulo's functions, Accumulo instantiates copies of the stack across the nodes in its cluster. This behavior follows

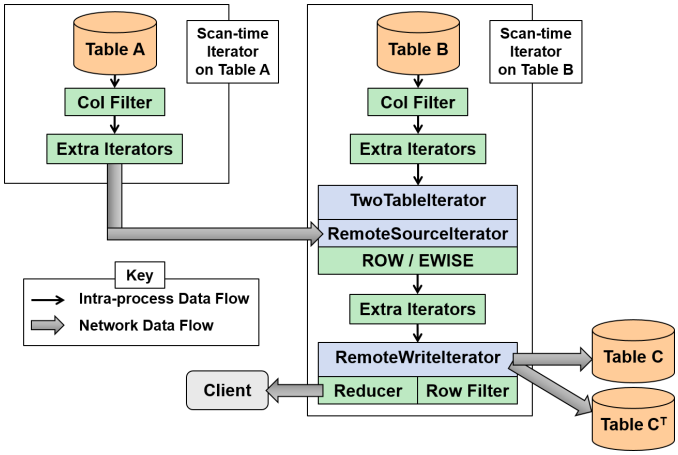

Fig. 1: Graphulo TwoTable template iterator stack. Every tablet server hosting tablets of $\mathbf{A}$ and $\mathbf{B}$ run this stack in parallel. Each GraphBLAS kernel is a special case of TwoTable.

GraphBLAS Kernel	Graphulo Implementation
BuildMatrix ($\oplus$)	Accumulo BatchWriter
ExtracTuples	Accumulo BatchScanner
MxM ($\oplus, \otimes$)	TwoTableIterator ROW mode, performing $\mathbf{A}^\mathsf{T}\mathbf{B}$
EwiseMult ($\otimes$)	TwoTableIterator EWISE mode
EwiseAdd ($\oplus$)	Similar to EwiseMult, with non-matching entries
Extract	Row and column filtering
Apply (f)	Extra Iterators
Assign	Apply with a key-transforming function
Reduce ($\oplus$)	Reducer module on RemoteWriteIterator
Transpose	Transpose option on RemoteWriteIterator

TABLE I: GraphBLAS implementation overview.

BigTable's design for horizontal distribution: *tables* are divided into *tablets*, each of which is hosted by a *tablet server*. Tablet servers execute a copy of the iterator stack that reads from each tablet of input tables $\mathbf{A}$ and $\mathbf{B}$ the tablet server hosts. The same distribution applies to output tables $\mathbf{C}$ and $\mathbf{C}^\mathsf{T}$; writes to $\mathbf{C}$ and $\mathbf{C}^\mathsf{T}$ ingest into each tablet server hosting their tablets. The whole scheme is embarrassingly parallel. However, it is important for developers to design the rows of their table schemas intelligently in order to evenly partition entries among their tablets and achieve good load balancing.

The boxes of Figure 1 in blue are the core components of Graphulo's iterator stack. The RemoteSourceIterator and RemoteWriteIterator extend the Accumulo client Scanner and BatchWriter, respectively, to read from and write to Accumulo tables inside an iterator stack. The TwoTableIterator merges two iterator stacks along a prefix of their keys, depending on whether it is used for a row-wise operation like matrix multiply (ROW mode) or an element-wise operation (EWISE mode).

Table I summarizes the GraphBLAS kernels and how parts of the TwoTable stack realize them. We discuss each kernel separately to show how they are individually implemented, but in practice kernel fusion is critical for performance.

To fuse a set of kernels is to execute them in one step, without writing the intermediary matrices between them. BLAS packages have a long history of fused kernels, stemming from the original GEMM call which fuses matrix multiplication, matrix addition, scalar multiplication, and transposition [11].

Kernel fusion is particularly important in the case of BigTable databases because writing out intermediary tables implies extra round trips to disk. It is possible to fuse kernels in Graphulo until a sort is required, since sorting is a blocking operation. Related research efforts are exploring additional GraphBLAS optimizations, such as decomposing kernels into finer-grained tasks that better utilize parallel architecture [12].

The symbols $\oplus$, $\otimes$, and f in Table I indicate user-defined functions supplied as parameters to the GraphBLAS kernels. In Graphulo these functions take the form of user-provided Java classes. Graphulo provides interfaces that make writing these operations easy provided they abide by a contract following the structure of a semi-ring: $0 \otimes a = 0$, $0 \oplus a = a$, $f(0) = 0$, associativity, and idempotence and distributivity in certain cases. Developers are free to insert general iterators or break these contracts, so long as they understand their role in Accumulo's distributed execution and lazy summing.

The following sections discuss Graphulo's representation of matrices, Graphulo's TwoTable iterators, and how those iterators implement the GraphBLAS kernels.

A. Matrices in Accumulo

We represent a matrix as a table in Accumulo. The row and column qualifier portions of an Accumulo key store the row and column indices of a matrix entry. Values are uninterpreted bytes that can hold any type, not necessarily numeric.

Other portions of the Accumulo key—column family, visibility, and timestamp—are available for applications to use. Many applications use the visibility portion of a key for cell-level security, restricting operations to run only on those entries a user has permission to see. Applications can define how visibilities propagate through the Graphulo iterators.

The column family provides a flexible grouping capability. For example, it is possible to store two matrices inside the same Accumulo table by distinguishing them with separate column families. This could be useful when the two matrices are frequently accessed together, since Accumulo would store them in the same tablets with alignment on rows. Applications may also leverage Accumulo's locality groups in order to store certain column families separately, which is similar to how Accumulo separately stores distinct tables but on a fine-grained, columnar level.

Zero-valued entries need not be stored in the Accumulo table in order to efficiently store sparse matrices. We treat the concept of "null" the same as the concept of "zero" for both storage and processing. However, it is not forbidden to store zero entries; a zero entry may spuriously arise during processing when 3, -5, and 2 are summed together, for example, though Graphulo prunes such entries as an optimization by default. We encourage developers not to rely on the presence or absence of zero entries for algorithm correctness.

B. BuildMatrix and ExtracTuples

The *BuildMatrix* and *ExtracTuples* GraphBLAS functions are the constructors and destructors of matrices from or to a set of "triples": a list of (row, column, value) entries.

Accumulo already supports these operations by means of its BatchWriter and BatchScanner client interfaces. Constructing a matrix is as simple as writing each tuple to an Accumulo table. Destructing a matrix is as simple as scanning a table.

If the list of triples for *BuildMatrix* contains multiple values for the same row and column, users commonly define a function $\oplus$ to store the sum of colliding values in the matrix. Accumulo combiner iterators achieve this behavior by lazily summing duplicate values according to $\oplus$ during compactions and scans.[1] Default behavior ignores all but one of them.

C. MxM

Graphulo matrix multiplication was previously explained in [9]. As a brief summary, Graphulo computes the matrix multiplication $\mathbf{AB}$ by placing a RemoteSourceIterator, TwoTableIterator, and RemoteWriteIterator on a BatchScan of table $\mathbf{B}$. The RemoteSourceIterator scans the transpose table $\mathbf{A}^\intercal$. The TwoTableIterator aligns entries from $\mathbf{A}^\intercal$ and $\mathbf{B}$ along the row portion of their keys, iterating both inputs in lockstep until a matching row is found. A user-defined Java class implementing a $\otimes$ function multiplies entries from matching rows according to the outer product algorithm. Partial products flow into the RemoteWriteIterator, which sends entries to the result table via an Accumulo BatchWriter. A user-defined iterator class implementing a $\oplus$ function lazily sums values on the result table during its next scan or compaction.

We call TwoTableIterator's workflow during *MxM* its ROW mode. More advanced uses of ROW mode may provide a user-defined strategy for processing two aligned rows of data. The Jaccard implementation described in Section III-A, for example, uses custom row processing to fuse the *EwiseAdd* of three *MxM* kernels by performing additional multiplications.

D. EwiseAdd and EwiseMult

The element-wise GraphBLAS kernels are similar to *MxM*, except that TwoTableIterator runs in EWISE mode and table $\mathbf{A}$ is not treated as its transpose. During EWISE mode, TwoTableIterator aligns entries from $\mathbf{A}$ and $\mathbf{B}$ along their row, column family, and column qualifier. The *EwiseMult* kernel passes matching entries to a user-defined $\otimes$ function.

The *EwiseAdd* kernel acts on both matching and non-matching entries. Non-matching entries pass directly to the RemoteWriteIterator. Matching entries pass to a "multiplication function" that implements the $\oplus$ addition logic.

E. Extract

The *Extract* kernel stores the subset of a matrix into a new matrix. Ranges of row and column indices specify the subset.

Graphulo implements *Extract* via row and column filtering on the ranges of indices. The client passes these ranges to Accumulo's tablet servers by transmitting serialized options attached to iterator configuration data. The RemoteWriteIterator handles row filtering by seeking the iterators above it

[1] BigTable systems do not run iterators on entries immediately as they are ingested in order to maximize write performance.

to only read entries from the indexed rows. Column filtering occurs at an iterator right after reading entries from the table.

The difference between row and column filtering is due to Accumulo's design as a row-store database. Whereas filtering rows is efficient since all data is accessed and stored by row, column filtering requires reading entries from all columns and discarding those outside the column indices. Future work could optimize column filtering for locality groups when present.

F. Apply and Assign

The *Apply* kernel applies a function f to every entry of a matrix. The kernel assumes that $f(0) = 0$ so that processing need only run on nonzero entries. Applying f in Accumulo takes the form of an extra iterator implementing f that can be placed at any point in an iterator stack. It is easy to fuse *Apply* with other kernels by including the iterator for f at appropriate locations.

Graphulo supports both stateless and stateful *Apply* functions f, with the caveat that stateful functions must cope with distributed execution wherein multiple instances of f run concurrently, each one seeing a portion of all entries.

The *Assign* kernel assigns a matrix to a subset of another matrix according to a set of row and column indices. An *Apply* iterator implements *Assign* by transforming the keys of entries to their corresponding keys in the new matrix.

G. Reduce

The *Reduce* kernel gathers information of reduced dimensionality about a matrix via a process similar to user-defined aggregation [13]. Often this information is a scalar, such as the set of unique values that occur in a matrix or the number of entries whose value exceeds a threshold. These scalars may be used inside control structures, such as how kTruss in Section III-B multiplies matrices until the number of partial products at each multiplication does not change, indicating convergence.

Many *Reduce* use cases transmit data to the client for control purposes rather than write data to a new table. In order to accommodate these cases and also facilitate fusing *Reduce* into other kernels, we integrated *Reduce* into the RemoteWriteIterator by coupling it with a "Reducer object" that processes every entry the RemoteWriteIterator sees.

The Reducer object implements the signature of a commutative monoid. It has a "zero" state that is the state of the Reducer before processing any entries. The reducer "sums in" entries one at a time via user-defined $\oplus$ logic. Once the RemoteWriteIterator finishes processing entries for the tablet it is running on, it asks the Reducer for the result of reducing all the entries the Reducer has seen, and it forwards that result to the client through the standard BatchScanner channel on which the client instantiated the whole TwoTable stack. The client combines local results from reducing each tablet to obtain the global result of *Reduce*.

The above scheme works because a *Reduce* call's result is typically small. It is feasible to fit the result into tablet server memory and transmit it to the client. If the result is not small, it may be wiser to store results at the server in a new table. *Reduce* calls that store results at the server, such as summing the columns of a matrix into a vector, can be implemented via *Apply* and a $\oplus$ iterator on the result table, or sometimes as an *MxM* with a constant matrix.

H. Transpose

The *Transpose* kernel switches the row and column of entries from a matrix. While *Transpose* could be implemented as an *Apply*, we found that it is used so frequently that it is worth making it a built-in option of the RemoteWriteIterator.

III. Algorithms

A. Vertex Similarity: Jaccard Coefficients

Vertex similarity is an important metric for applications such as link prediction [14]. One measure for similarity between two vertices is their Jaccard coefficient. This quantity measures the overlap of two vertices' neighborhoods in an unweighted, undirected graph. For vertices v_i and v_j where $N(v)$ denotes the neighbors of vertex v, the Jaccard coefficient is defined as

$$\mathbf{J}_{ij} = \frac{|\ N(v_i) \bigcap N(v_j)\ |}{|\ N(v_i) \bigcup N(v_j)\ |}$$

Gadepally et al formulated an algorithm in terms of the GraphBLAS to compute the Jaccard coefficients between all vertices for a graph represented as an unweighted, undirected adjacency matrix [8]. Algorithm 1 summarizes their formulation. They employ an optimization that restricts computation to the upper triangle of the adjacency matrix, taking advantage of the symmetry in the Jaccard coefficient definition. The notation $\mathrm{triu}(\cdot, 1)$ borrows MATLAB syntax for taking the strict upper triangle of a matrix.

Line 1 stores the degrees of each vertex in a vector $\mathbf{d}$. For the Graphulo implementation, we assume these degrees are pre-computed inside a separate degree table in Accumulo. Computing degree tables is often performed during data ingest, since fast access to degree information is useful for query planning, load balancing, filtering, and other analytics [15].

Graphulo computes the Jaccard coefficient matrix $\mathbf{J}$ in one fused *MxM* call that Figure 2 highlights. The inputs to the *MxM* are the lower and upper triangles $\mathbf{L}$ and $\mathbf{U}$ of $\mathbf{A}$, which we obtain by applying strict lower and upper triangle filters.

Given inputs $\mathbf{L}$ and $\mathbf{U}$, the TwoTableIterator computes[2] $\mathbf{L}^\intercal\mathbf{U} = \mathbf{U}\mathbf{U}$. We could compute the other required products $\mathbf{U}\mathbf{U}^\intercal$ and $\mathbf{U}^\intercal\mathbf{U}$ in a similar fashion with two separate multiplication calls. However, all the information we need to compute $\mathbf{U}\mathbf{U}^\intercal$ and $\mathbf{U}^\intercal\mathbf{U}$ is available during the computation of $\mathbf{U}\mathbf{U}$, which suggests an opportunity to fuse the three matrix multiplications together.

We implement the fusion of $\mathbf{U}\mathbf{U}^\intercal + \mathbf{U}^\intercal\mathbf{U} + \mathbf{U}\mathbf{U}$ by providing a custom row multiplication function to TwoTableIterator that computes all three products at once. In addition to multiplying pairs of entries in the Cartesian product of matching rows between $\mathbf{L}$ and $\mathbf{U}$ to compute $\mathbf{L}^\intercal\mathbf{U} = \mathbf{U}\mathbf{U}$, the custom function multiplies pairs of entries in the Cartesian product of

[2]Recall that Graphulo transposes the left input to *MxM* calls. The transpose of lower triangular matrix $\mathbf{L}$ is the upper triangular matrix $\mathbf{U}$.

Input: Unweighted, undirected adjacency matrix $\mathbf{A}$
Output: Upper triangle of Jaccard coefficients $\mathbf{J}$

1 $\mathbf{d} = \mathrm{sum}(\mathbf{A})$ *// pre-computed in degree table*
2 $\mathbf{U} = \mathrm{triu}(\mathbf{A}, 1)$ *// strict upper triangle filter*
3 $\mathbf{J} = \mathrm{triu}(\mathbf{UU} + \mathbf{UU}^\intercal + \mathbf{U}^\intercal\mathbf{U}, 1)$ *// fused MxM*
4 **foreach** *nonzero entry* $\mathbf{J}_{ij} \in \mathbf{J}$ **do**
5 | $\mathbf{J}_{ij} = \mathbf{J}_{ij}/(\mathbf{d}_i + \mathbf{d}_j - \mathbf{J}_{ij})$ *// stateful Apply on* $\mathbf{J}$
6 **end**

Algorithm 1: Jaccard
Comments describe the Graphulo implementation.

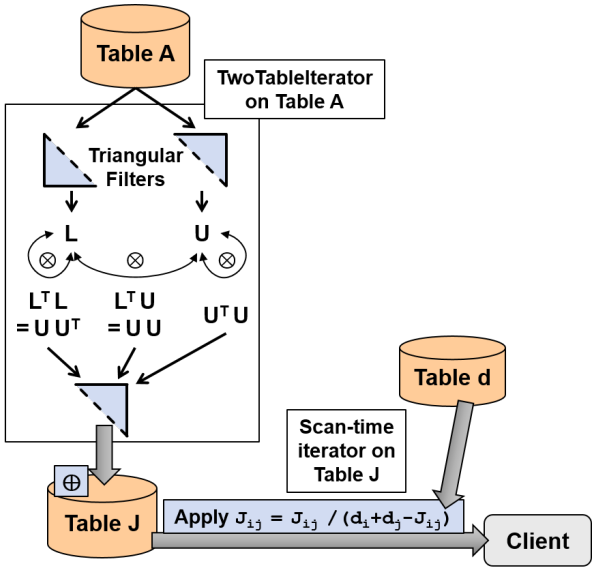

Fig. 2: Visualization of the Jaccard algorithm. The three '$\otimes$' symbols highlight the fused *MxM* kernel below them.

its left input $\mathbf{L}$ with itself to compute $\mathbf{L}^\intercal\mathbf{L} = \mathbf{UU}^\intercal$, as well as its right input $\mathbf{U}$ with itself to compute $\mathbf{U}^\intercal\mathbf{U}$. The additional multiplications also run on rows of $\mathbf{L}$ and $\mathbf{U}$ that do not match, as in the pattern of an *EwiseAdd* kernel. After passing the entries from all three matrix multiplications through a further strict upper triangle filter (line 3), the RemoteWriteIterator sends them to the result table $\mathbf{J}$ with an iterator that sums colliding values.

After the *MxM* operation completes, indicating that all partial products from the three matrix multiplications have written to table $\mathbf{J}$, we add a stateful *Apply* iterator to the scan scope of $\mathbf{J}$ that computes lines 4–6. The iterator performs a kind of broadcast join, scanning the degree table $\mathbf{d}$ into the memory of tablet servers hosting $\mathbf{J}$'s tablets in order to efficiently compute line 5. Holding $\mathbf{d}$ in memory is feasible because it is significantly smaller than the original table $\mathbf{A}$.

As a final optimization to *MxM* that applies when multiplying a matrix with itself, we used the "deep copy" feature of Accumulo iterators to duplicate the iterator stack on $\mathbf{A}$ and use it for both inputs to TwoTableIterator. This optimization eliminates the need for a RemoteSourceIterator, which saves entry serialization and coordination with Accumulo's master. While it did not significantly increase performance during Section IV's single-node evaluation, we anticipate the deep copy technique will have larger impact in a multi-node setting.

Input: Unweighted, undirected adjacency matrix $\mathbf{A}_0$, integer k
Output: Adjacency matrix of k-truss subgraph $\mathbf{A}$

1 $z' = \infty, \mathbf{A} = \mathbf{A}_0$ *// table clone*
2 **repeat**
3 | $z = z'$
4 | $\mathbf{B} = \mathbf{A}$ *// table clone*
5 | $\mathbf{B} = \mathbf{B} + 2\mathbf{AA}$ *// MxM with* $a \otimes b = 2$ *if* $a, b \neq 0$
6 | $\mathbf{B}(\mathbf{B} \,\%\, 2 == 0) = 0$ *// filter on* $\mathbf{B}$
7 | $\mathbf{B}((\mathbf{B} - 1)/2 < k - 2) = 0$ *// filter on* $\mathbf{B}$
8 | $\mathbf{A} = |\mathbf{B}|_0$ *// Apply on* $\mathbf{B}$*; switch* $\mathbf{A} \leftrightarrow \mathbf{B}$
9 | $z' = \mathrm{nnz}(\mathbf{A})$ *// Reduce, gathering* nnz *at client*
10 **until** $z == z'$ *// client controls iteration*

Algorithm 2: kTruss
Comments describe the Graphulo implementation.

B. Truss Decomposition

The k-truss of an unweighted, undirected simple graph is the largest subgraph in which every edge is part of at least $k-2$ triangles. Computing the k-truss is useful for focusing large graph analysis onto a cohesive subgraph that has a hierarchical structure one can vary with k [16].

Gadepally et al formulated a GraphBLAS algorithm to compute the k-Truss of a graph represented by an unweighted, undirected incidence matrix [8]. The algorithm iteratively deletes edges that are part of fewer than $k-2$ triangles until all edges are part of at least $k-2$ triangles. Iteration is a particularly challenging feature for an Accumulo implementation since Accumulo stores intermediary tables on disk.

We adapted the algorithm to run on a graph's adjacency matrix in Algorithm 2. The nnz call in line 9 computes the number of nonzero entries in a matrix. The product $\mathbf{AA}$ in line 5 computes the number of triangles each edge is part of.

The '%' symbol in line 6 indicates remainder after integer division, and so line 6 deletes even values. Because $\mathbf{A}$'s nonzero values are odd (specifically, the value 1) and because partial products from $2\mathbf{AA}$ are even (specifically, the value 2), line 6 effectively filters out entries from $\mathbf{B}$ that are not present in $\mathbf{A}$. Line 7 deletes edges that are part of fewer than $k-2$ triangles after "undoing" the $+1$ from $\mathbf{A}$ and the $\times 2$ from $2\mathbf{AA}$. The $|\mathbf{B}|_0$ in line 8 indicates the zero norm of $\mathbf{B}$, which sets nonzero values to 1.

We specifically designed Algorithm 2 to minimize the number of intermediary tables in order to minimize round-trips to disk. The insight that allows us to reduce the number of intermediary table writes, from two in a naive formulation to one as presented here, is finding a way to distinguish edges that are part of at least $k-2$ triangles but not present in the original graph from edges present in the original graph. A naive method to distinguish edges is by computing the *EwiseMult* $\mathbf{A} \otimes \mathbf{B}$ where $\otimes$ is "logical and". We eliminate the *EwiseMult* by playing tricks with parity in lines 5–7 as described above.

The Graphulo kTruss implementation uses two temporary tables $\mathbf{A}$ and $\mathbf{B}$. We initialize $\mathbf{A}$ by cloning input table $\mathbf{A}_0$.[3]

Each iteration begins with cloning $\mathbf{A}$ into $\mathbf{B}$ as in line 4. Implementing line 4 with a table clone is an optimization that avoids rewriting the entries in $\mathbf{A}$. Graphulo constructs an *MxM* iterator stack to multiply $\mathbf{A}$ with itself[4] and sum the result into $\mathbf{B}$ as in line 5. The *MxM* stack includes a $\otimes$ function that evaluates to 2 on nonzero inputs, as well as an extra iterator following $\otimes$ that filters out entries along the diagonal as another optimization, which is correct since the k-truss is defined on simple graphs. A standard $\oplus$ iterator on $\mathbf{B}$ sums partial products.

After the *MxM* stack completes, indicating that $\mathbf{B}$ contains every partial product, Graphulo places an additional iterator on $\mathbf{B}$ after the $\oplus$ iterator to filter out entries that are even (line 6) or fail the k-Truss condition (line 7). A final iterator on $\mathbf{B}$ sets nonzero values to 1 as in line 8.

A *Reduce* call computes $\mathrm{nnz}(\mathbf{A})$ in line 9 by counting entries. The algorithm has converged and may terminate per line 10 once $\mathrm{nnz}(\mathbf{A})$ does not change between iterations.

The temporary tables switch roles between iterations, deleting the old $\mathbf{A}$ before switching $\mathbf{A}$ with $\mathbf{B}$ and again cloning $\mathbf{A}$ into $\mathbf{B}$. After the last iteration, we rename the new $\mathbf{A}$ to the designated output table via a clone and delete.

IV. Performance

In this section we conduct an experiment to (1) provide a single-node evaluation of Graphulo on the Jaccard and 3Truss (fixing $k = 3$ in kTruss) algorithms and (2) gain insight into when it is profitable to execute the algorithms inside Accumulo versus an external system.

We compare Graphulo to *main-memory* external systems because they are among the best options for computing on a subgraph (e.g. cued analytics [17]), a use case databases accelerate by creating table indexes. Such subgraphs lie on the threshold of fitting into memory wherein main-memory computation is feasible; we therefore choose problem sizes that also lie on the threshold of fitting in memory. We consider both sparse and dense matrix systems because dense systems generally perform orders of magnitude faster but have severe memory constraints, whereas sparse systems offer intermediate performance in exchange for handling larger matrices.

Specifically, we choose two open-source main-memory matrix math systems for comparison: the dynamic distributed dimensional data model (D4M) MATLAB library for sparse matrices [18] and the Matrix Toolkits Java (MTJ) library for dense matrices [19]. D4M provides an associative array API that maps to databases like Accumulo and calls MATLAB's sparse matrix functions [20]; D4M has frequently been used to analyze subsets of database tables that fit in memory. We choose D4M for the role of sparse matrix math because it is one of the best GraphBLAS alternatives to Accumulo for single-node computation; GraphMat [21] (single-node) as well as CombBLAS [22] and GraphPad [23] (distributed) are also candidates. MTJ is one of many Java libraries that provide high-performance matrix math [24]. We choose MTJ for the role of dense matrix math for its off-the-shelf ease of use.

Following our past experimental setup [9], we evaluate the performance of Jaccard and 3Truss by measuring runtime while varying two parameters: problem size and computational resources. Varying these parameters allows us to gauge both weak scaling (varying problem size while holding computational resources constant) and strong scaling (varying computational resources while holding problem size constant).

We control problem size via the size of input matrices, and we control computational resources via the number of tablets on input and output tables. The number of tablets controls how many threads Accumulo uses for reading, writing, and iterator processing. In our single-node setup, this relates to how well Accumulo uses the 8 cores it has available. We limit the number of tablets to 1 or 2, since Accumulo runs enough threads to fully use all 8 cores at 2 tablets.

We set the number of tables on input tables by compacting them prior to each experiment. For 2 tablet experiments, we choose a split point that evenly divides each input table and pre-split output tables on the same splits.[5] In order to prevent Accumulo from creating additional splits during larger SCALE experiments (17 for Jaccard, 16 for 3Truss), we increased the `table.split.threshold` parameter to 5 GB.

Most users deploy Accumulo on a sizable cluster. Graphs as large as 70 trillion edges (1.1 PB) have been analyzed by clusters as large as 1200 machines and 57.6 TB collective memory [25]. This work evaluates single-node performance as a proof of concept and an indicator for how multi-node performance might scale.

We conduct the experiments on a Linux Mint 17.2 laptop with 16 GB RAM, two dual-core Intel i7 processors, and a 512 GB SSD. Atop single-instance Hadoop 2.4.1 and ZooKeeper 3.4.6, we allocated 2 GB of memory to an Accumulo tablet server with room to grow to 3 GB, 1 GB for native in-memory maps and 256 MB for data and index cache. We ran a snapshot build of Accumulo 1.8.0 at commit `9aac9c0` in order to incorporate a bugfix that affects Graphulo's stability [26].

We use the Graph500 unpermuted power law graph generator [27] to create random input adjacency matrices whose first rows are high-degree "super-nodes" and whose subsequent rows exponentially decrease in degree. Power law distributions are widely used to model real world graphs [28].

The generator takes SCALE and EdgesPerVertex parameters, creating matrices with 2^{SCALE} rows and EdgesPerVertex $\times$ 2^{SCALE} entries. We fix EdgesPerVertex to 16 and use SCALE to vary problem size. In order to create undirected and unweighted adjacency matrices without self-loops, we merge the generated matrix with its transpose, ignore duplicate

[3] Cloning Accumulo tables is a cheap operation because no data is copied; Accumulo simply marks the cloned table's Hadoop RFiles as shared with the clonee after flushing entries in memory to disk.

[4] Recall that $\mathbf{A}^\intercal = \mathbf{A}$ because $\mathbf{A}$ is the adjacency matrix for an undirected graph. The product $\mathbf{A}^\intercal\mathbf{A}$ is the same as $\mathbf{AA}$.

[5] In the case of Graphulo kTruss, all intermediary tables use the same splits as the input table since cloning a table preserves its splits.

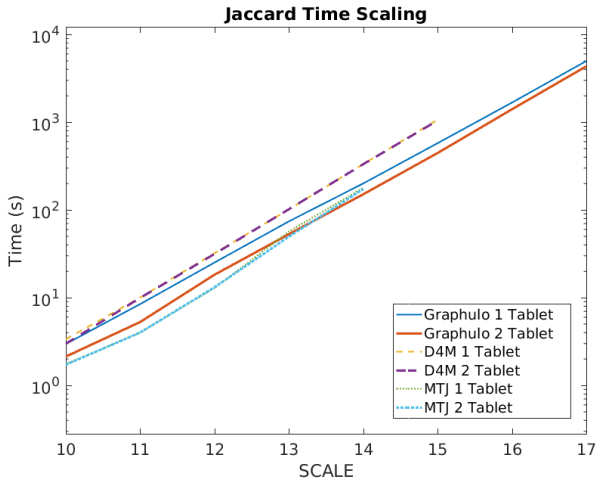

Fig. 3: Jaccard experiment processing time.

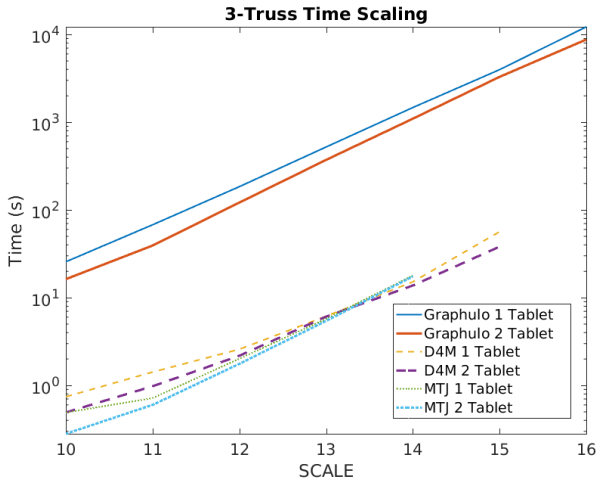

Fig. 4: 3Truss experiment processing time.

SCALE	nnz($\mathbf{A}$)	nnz(Jaccard($\mathbf{A}$))	Partial Products	Graphulo Overhead	Graphulo 1 Tablet	Graphulo 2 Tablets	D4M 1 Tablet	D4M 2 Tablets	MTJ 1 Tablet	MTJ 2 Tablets
10	2.10×10^4	2.15×10^5	1.01×10^6	4.7x	2.97	2.14	3.36	2.99	1.76	1.72
11	4.52×10^4	7.07×10^5	3.10×10^6	4.4x	8.46	5.29	1.01×10^1	9.96	3.99	4.01
12	9.67×10^4	2.18×10^6	9.29×10^6	4.3x	2.52×10^1	1.83×10^1	3.22×10^1	3.16×10^1	1.29×10^1	1.32×10^1
13	2.04×10^5	6.75×10^6	2.71×10^7	4.0x	7.42×10^1	5.30×10^1	1.02×10^2	1.02×10^2	5.68×10^1	4.96×10^1
14	4.26×10^5	2.02×10^7	7.77×10^7	3.8x	2.01×10^2	1.51×10^2	3.34×10^2	3.33×10^2	1.79×10^2	1.73×10^2
15	8.83×10^5	6.07×10^7	2.22×10^8	3.7x	5.77×10^2	4.46×10^2	1.07×10^3	1.05×10^3		
16	1.82×10^6	1.77×10^8	6.20×10^8	3.5x	1.68×10^3	1.41×10^3				
17	3.73×10^6	5.16×10^8	1.72×10^9	3.3x	4.99×10^3	4.34×10^3				

TABLE II: Jaccard experiment statistics. Graphulo is competitive and better scales due to low overhead.

SCALE	nnz($\mathbf{A}$)	nnz(3Truss($\mathbf{A}$))	Partial Products	Graphulo Overhead	Graphulo 1 Tablet	Graphulo 2 Tablets	D4M 1 Tablet	D4M 2 Tablets	MTJ 1 Tablet	MTJ 2 Tablets
10	2.10×10^4	2.03×10^4	5.94×10^6	293.3x	2.57×10^1	1.63×10^1	0.74	0.49	0.49	0.28
11	4.52×10^4	4.35×10^4	1.22×10^7	280.7x	6.78×10^1	3.93×10^1	1.42	0.98	0.72	0.60
12	9.67×10^4	9.20×10^4	5.45×10^7	592.7x	1.84×10^2	1.21×10^2	2.58	2.18	2.02	1.76
13	2.04×10^5	1.93×10^5	1.59×10^8	825.5x	5.22×10^2	3.72×10^2	6.16	6.09	5.74	5.44
14	4.26×10^5	3.99×10^5	4.55×10^8	1140.6x	1.47×10^3	1.10×10^3	1.52×10^1	1.38×10^1	1.79×10^1	1.75×10^1
15	8.83×10^5	8.20×10^5	1.30×10^9	1582.5x	3.97×10^3	3.29×10^3	5.65×10^1	3.82×10^1		
16	1.82×10^6	1.67×10^6	3.62×10^9	2167.0x	1.22×10^4	8.77×10^3				

TABLE III: 3Truss experiment statistics. D4M and MTJ execute faster, assuming sufficient memory, due to high overhead. Graphulo overhead is defined as how many times more entries Graphulo writes into Accumulo than D4M or MTJ. Runtimes are listed in seconds.

entries, and filter out the diagonal. These modifications slightly change the input graphs' edge count; see the nnz($\mathbf{A}$) column in Tables II and III for exact counts.

Figures 3 and 4 plot the runtime for Graphulo, D4M, and MTJ on 1 and 2 tablets and various problem sizes. D4M's sparse matrices and MTJ's dense matrices exceed our machine's memory at SCALE 16 and 15, respectively.

Figure 3 indicates competitive Graphulo performance on the Jaccard algorithm. Graphulo always outperforms D4M and runs on par with MTJ for problem sizes that D4M and MTJ can hold in memory. Figure 4 shows an order of magnitude faster D4M and MTJ performance on the 3Truss algorithm.

In order to analyze the disparity in performance between Jaccard and 3Truss, we present additional experimental information in Tables II and III. The "nnz(Jaccard($\mathbf{A}$))" and "nnz(3Truss($\mathbf{A}$))" columns list the number of nonzero entries in the result from their algorithms. D4M and MTJ write exactly this many entries into Accumulo because they compute the complete result in memory and insert it as is.

Recall that a partial product is a multiplied value $a * b$ computed during a matrix multiplication. The "Partial Products" column in Tables II and III lists the total number of partial products computed during each algorithm (not including entries filtered out by Jaccard's strict upper triangle filter and kTruss's no-diagonal filter). Graphulo writes exactly this many entries into Accumulo because it implements the outer product matrix multiply algorithm; Graphulo writes individual partial products to Accumulo and defers summing them to a $\oplus$ iterator that runs during compactions and scans. In contrast, D4M and MTJ pre-sum partial products before writing to Accumulo in a manner similar to the inner product algorithm. Unfortunately an inner product formulation that pre-sums partial products

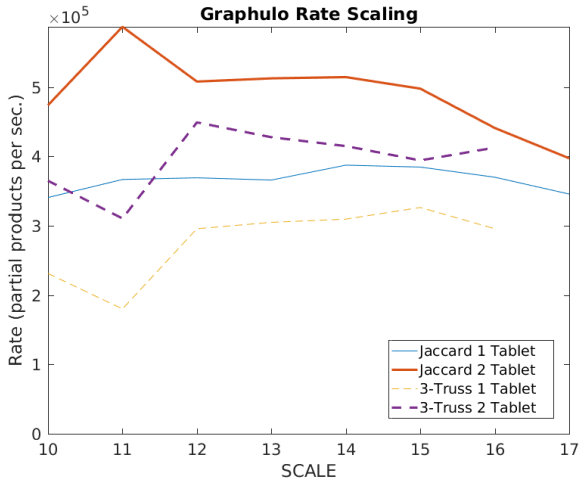

Fig. 5: Graphulo Processing Rate during Jaccard, 3Truss.

is infeasible for Graphulo because, unlike a main-memory system, Graphulo does not assume it can hold a whole table in memory and would therefore need to re-read an input table many times, including many re-reads from disk.

We define the *Graphulo overhead* as how many times more entries Graphulo writes into Accumulo than D4M or MTJ. For reference, the Graphulo overhead of multiplying two power law matrices is 2.5–3x, decreasing with matrix size [9]. Table II shows that Jaccard has a similar Graphulo overhead of 3–5x, also decreasing with matrix size. At this low an overhead, it makes sense that Graphulo outperforms D4M and MTJ because the performance gain from computing inside Accumulo outweighs the cost of writing additional entries.

Table III shows a drastically different Graphulo overhead for 3Truss of 280–2200x, increasing with matrix size. The source of the additional overhead is that Graphulo writes out an intermediary table at each iteration, including all partial products of $\mathbf{AA}$ before applying the filter in lines 6 and 7 of Algorithm 2. D4M and MTJ do not need to write intermediary values since they hold them in memory. At this high an overhead, D4M and MTJ are better places to execute 3Truss.

In terms of weak and strong scaling, Graphulo actually performs quite well if we account for its overhead. Figure 5 plots Graphulo's processing rate during Jaccard and 3Truss. We express processing rate as the number of partial products written to Accumulo divided by algorithm runtime so that we can compare processing rate to Accumulo insert rates.

For reference, the largest cited peak single-node write rates into Accumulo are on the order of 400k per second [29]. We believe the reason why the Jaccard 2-tablet rate exceeds the single-node record is due to its fused *MxM* call: writing partial products from 3 multiplications while reading a single table.

Both algorithms show relatively horizontal lines, indicating constant processing rate as problem size varies. Two-tablet rates are about 1.5 times greater than one-tablet rates, indicating a potential for strong scaling that a multi-node experiment could further evaluate in future work.

V. Related Work

One way to characterize the difference between Jaccard and kTruss's performance inside Accumulo is as the difference between data- and state-parallel algorithms. A *data-parallel* operation easily runs on data partitioned among several threads or machines. A *state-parallel* operation runs more efficiently on data gathered in one location, either because of data correlation, the need for global calculation, or the presence of iteration which induces communication between every round.

Many distributed databases, Accumulo included, efficiently run data-parallel workloads but are poorly suited to state-parallel workloads. The solution proposed in Section 4.3 of the UDA-GIST framework [30]—to gather the entire state into a single node during the iterative portion of the workflow—is essentially the solution that D4M and MTJ implement.

The Graphulo *Reduce* implementation is similar in design to the GLADE aggregation framework for implementing generalized linear aggregates (GLAs) inside parallel databases [31]. Both follow a pattern of accumulating information inside a thread-local "reducer object" and coalescing the objects from multiple threads and nodes by serializing and merging them, in order to gather a fully reduced result at one location.

Multi-platform query optimizers such as Rheem [32] and Musketeer [33] evaluate equivalent implementations of an algorithm on different data processing environments, in order to select the best one relative to a cost model for execution. It seems clear that a multi-platform optimizer could evaluate and compare two plans for executing a graph algorithm, one using Graphulo and one using a main-memory system. Less clear is whether the overhead conditions described in Section IV, under which execution on one system outperforms the other, could be encoded into a cost model. Should such a cost model exist, we could spare developers the burden of analyzing algorithms in order to determine the system that executes them best. The BigDAWG polystore framework takes a different approach, introducing 'scope' clauses for developers to manually specify the system for executing portions of an algorithm and 'cast' clauses for moving data between system [34].

Cheung et al applied a program synthesis technique called *verified lifting* in order to recognize recognize fragments of legacy code which operate on data obtained from a database query and could be pushed into the database in order to achieve better performance [35]. Similar to multi-platform query optimizers, verified lifting facilitates automatic rewriting of algorithm fragments to execute on the best-suited system, assuming an accurate cost model. Their difference is that optimizers take higher-level queries as input, usually in a form that can be parsed into a logical algebra, whereas verified lifting abstracts a high-level specification from general-purpose code.

PipeGen is an initiative to reduce the cost of data transfer between systems by generating efficient binary transfer protocols and injecting them into systems' import and export facilities [36]. Should efforts like PipeGen gain momentum, computing in specialized external systems may gain feasibility for algorithms with otherwise prohibitive data transfer cost.

VI. Conclusion

In this work we detail how Graphulo's design enables executing the GraphBLAS kernels inside the Accumulo database. We cover the implementation of two graph algorithms and show how to optimize them for in-database execution via kernel fusion. Experiments comparing their performance to two main-memory matrix math systems show that I/O, in terms of database reads and writes, is a critical factor for determining whether an algorithm executes best inside a database or in an external system, assuming enough available memory.

In future work we aim to extend our analysis to a multi-node setting and include additional systems such as Spark and CombBLAS. Characterizing the traits of algorithms that determine their performance on different data processing systems is an exciting first step toward robust cross-system algorithm optimization, using each system where it performs best.

Acknowledgment

The authors thank Brandon Myers and Zachary Tatlock for their energetic feedback, Timothy Weale for his engineering camaraderie, the whole Graphulo team bridging Seattle and Cambridge, and Alan Edelman, Sterling Foster, Paul Burkhardt, and Victor Roytburd. This material is based on work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1256082.

References

[1] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[2] S. P. Ghosh, "Statistical relational tables for statistical database management," *IEEE Transactions on Software Engineering*, no. 12, pp. 1106–1116, 1986.

[3] M. Stonebraker and J. Hellerstein, "What goes around comes around," *Readings in Database Systems*, vol. 4, 2005.

[4] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. Capretz, "Data management in cloud environments: NoSQL and NewSQL data stores," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, no. 1, p. 22, 2013.

[5] 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.

[6] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin, "Summingbird: A framework for integrating batch and online mapreduce computations," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1441–1451, 2014.

[7] J. Kepner, D. Bader, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the graphblas: Seven good reasons," *Procedia Computer Science*, vol. 51, pp. 2453–2462, 2015.

[8] 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)*. IEEE, 2015.

[9] D. Hutchison, J. Kepner, V. Gadepally, and A. Fuchs, "Graphulo implementation of server-side sparse matrix multiply in the Accumulo database," in *High Performance Extreme Computing Conference (HPEC)*. IEEE, 9 2015.

[10] M. Stonebraker and U. Çetintemel, "One size fits all: an idea whose time has come and gone," in *International Conference on Data Engineering (ICDE)*. IEEE, 2005, pp. 2–11.

[11] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[12] M. M. Wolf, J. W. Berry, and D. T. Stark, "A task-based linear algebra building blocks approach for scalable graph analytics," in *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015.

[13] S. Cohen, "User-defined aggregate functions: bridging theory and practice," in *International Conference on Management of Data (SIGMOD)*. ACM, 2006, pp. 49–60.

[14] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American society for information science and technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[15] 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)*. IEEE, 2013.

[16] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[17] A. Reuther, J. Kepner, P. Michaleas, and W. Smith, "Cloud computing where ISR data will go for exploitation," in *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2009.

[18] 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 *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012, pp. 5349–5352.

[19] "Matrix toolkits java," https://github.com/fommil/matrix-toolkits-java.

[20] J. Kepner, J. Chaidez, V. Gadepally, and H. Jansen, "Associative arrays: Unified mathematics for spreadsheets, databases, matrices, and graphs," *New England Database Day (NEDB)*, 2015.

[21] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[22] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, pp. 496–509, 2011.

[23] M. J. Anderson, N. Sundaram, N. Satish, M. Patwary, T. L. Willke, and P. Dubey, "GraphPad: Optimized graph primitives for parallel and distributed platforms," in *International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2016.

[24] H. Arndt, M. Bundschus, and A. Naegele, "Towards a next-generation matrix library for java," in *International Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2009, pp. 460–467.

[25] P. Burkhardt and C. A. Waring, "A cloud-based approach to big graphs," in *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015.

[26] "Batchwriter locator cache out-of-sync when shared with tserver," https://issues.apache.org/jira/browse/ACCUMULO-4229.

[27] 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.

[28] V. Gadepally and J. Kepner, "Using a power law distribution to describe big data," in *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015.

[29] 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," *High Performance Extreme Computing Conference (HPEC)*, 2014.

[30] K. Li, D. Z. Wang, A. Dobra, and C. Dudley, "UDA-GIST: an in-database framework to unify data-parallel and state-parallel analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 557–568, 2015.

[31] F. Rusu and A. Dobra, "Glade: a scalable framework for efficient analytics," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 1, 2012.

[32] D. Agrawal, S. C. A. E. Z. Kaoudi, M. Ouzzani, P. P. J.-A. Quiané-Ruiz, and N. T. M. J. Zaki, "Road to freedom in big data analytics," pp. 479–484, 2016.

[33] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: all for one, one for all in data processing systems," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.

[34] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The bigdawg polystore system," *SIGMOD Record*, vol. 44, no. 2, pp. 11–16, 2015.

[35] A. Cheung, A. Solar-Lezama, and S. Madden, "Optimizing database-backed applications with query synthesis," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 3–14, 2013.

[36] B. Haynes, A. Cheung, and M. Balazinska, "Pipegen: Data pipe generator for hybrid analytics," in *Proceedings of the Seventh Symposium on Cloud Computing*. ACM, 2016.