Distributed Relational Algebra at Scale

Anonymous Author(s)*

ABSTRACT

Relational algebra forms a basis of primitive operations useful for applications in graphs, networks, program analysis, deductive databases, and logic. Despite its expressive power, relational algebra has not received the same attention in high-performance computing research as linear algebra, X, Y, or Z.

In this paper we present a set of efficient algorithms that tackle the problem of distributed, parallel relational algebra and use experiments from applications in graphs, program analysis, and datalog to evaluate our approach.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:

Anonymous Author(s). 2018. Distributed Relational Algebra at Scale. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Lorem ipsum dolar sit amat

2 RELATIONAL ALGEBRA

Relational algebra (RA) provides a basis of operations on relations (i.e., predicates, or sets of tuples) sufficient to implement a broad range of algorithms for databases and queries, data analysis, machine learning, graph problems, and constraint logic problems []. Scaling these underlying primitives, and finding an effective strategy for parallel communication to distribute them across multiple nodes, is thus a avenue for scaling and distributing algorithms for high-performance program analyses, deductive databases, among other applications. This section reviews the standard relational operations union, product, intersection, natural join, selection, renaming, and projection, along with their use in implementing two closely related example applications: graph problems and bottom-up datalog solvers.

The Cartesian product of two finite enumerations D_0 and D_1 is defined $D_0 \times D_1 = \{(d_0, d_1) \mid \forall d_0 \in D_0, d_1 \in D_1\}$. A *relation* $R \subseteq D_0 \times D_1$ is some subset of this product that defines a set of associated pairs of elements drawn from the two domains. For example, if R were the relation $(\geq)$ over natural numbers, both domains D_0 and D_1 would be $\mathbb{N}$ and the relation could be defined $(\geq) = \{(n_0, n_1) \mid n_0, n_1 \in \mathbb{N} \wedge n_0 \geq n_1\}$. Any relation R can also be viewed as a predicate P_R where $P_R(d_0, \ldots, d_k) \iff (d_0, \ldots, d_k) \in R$, or as a set of tuples, or as a database table.

We make some standard assumptions about relational algebra that differ from those of traditional set operations. Specifically, we assume that all our relations are sets of flat (first-order) tuples of natural numbers with a fixed, homogeneous arity. This means that the relation $(\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$ contains the tuple $(1, 2, 3)$, and not $((1, 2), 3)$. It also means that although our approach extends naturally to relations over arbitrary enumerable domains (such as integers, booleans, symbols/strings, lists of integers, etc)—we make the assumption that natural numbers may be used in the place of other enumerable domains when they are needed. Finally, this means that for operations like union or intersection, both relations must by union-compatable by having the same arity and column names.

... talk about names as indices?

2.1 Standard RA operations

...

Cartesian product. The product of two relations R and S is defined: $R \times S = \{(r_0, \ldots, r_k, s_0, \ldots, s_j) \mid (r_0, \ldots, r_k) \in R \wedge (s_0, \ldots, s_j) \in S\}$.

Union. The union of two relations R and R' may only be performed if both relations have the same arity but is otherwise set union: $R \cup R' = \{(r_0, \ldots, r_k) \mid (r_0, \ldots, r_k) \in R \vee (r_0, \ldots, r_k) \in R'\}$.

Intersection. The intersection of two relations R and R' may only be performed if both have k arity but is otherwise set intersection: $R \cap R' = \{(r_0, \ldots, r_k) \mid (r_0, \ldots, r_k) \in R \wedge (r_0, \ldots, r_k) \in R'\}$.

Projection. Projection is a unary operation that removes a column or columns from a relation—and thus any duplicate tuples that result from removing these columns. Projection of a relation R restricts R to a particular set of dimensions $\alpha_0, \ldots, \alpha_j$, where $\alpha_0 < \ldots < \alpha_j$, and is written $\Pi_{\alpha_0,\ldots,\alpha_j}(R)$. For each tuple, projection retains only stated columns: $\Pi_{\alpha_0,\ldots,\alpha_j}(R) = \{(r_{\alpha_0}, \ldots, r_{\alpha_j}) \mid (r_0, \ldots, r_k) \in R\}$.

Renaming. Renaming is a unary operation that renames (i.e., reorders) columns. Renaming columns can be defined in several different ways, including renaming all columns at once. We define our renaming operator, $\rho_{\alpha_i/\alpha_j}(R)$, to swap two columns, α_i and α_j where $\alpha_i < \alpha_j$—an operation that can be repeated to rename/reorder as many columns as desired:
$\rho_{\alpha_i/\alpha_j}(R) = \{(\ldots, r_{\alpha_j}, \ldots, r_{\alpha_i}, \ldots) \mid (\ldots, r_{\alpha_i}, \ldots, r_{\alpha_j}, \ldots) \in R\}$.

Selection. Selection is a unary operation that restricts a relation to tuples where a particular column matches a particular value. As with renaming, a selection operator may alternatively be defined to allow multiple columns to be matched at once, or to allow inequality

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-9999-9/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

or other predicates to be used in matching tuples. In our formulation, selection on multiple columns can be accomplished by repeated selection on a single column at a time. Selecting just those tuples from relation R where column α_i matches a value v is defined: $\sigma_{\alpha_i=v}(R) = \{(r_{\alpha_0}, \ldots, r_{\alpha_k}) \in R \mid r_{\alpha_i} = v\}$.

Selecting just those tuples from relation R where the values in columns α_i and α_j must match is defined: $\sigma_{\alpha_i=\alpha_j}(R) = \{(r_{\alpha_0}, \ldots, r_{\alpha_k}) \in R \mid r_{\alpha_i} = r_{\alpha_j}\}$.

Natural Join. Two relations can also be *joined* into one on a subset of columns they have in common. Join is a particularly important operation that combines two relations into one, where a subset of columns are required to have matching values, and generalizes both intersection and Cartesian product operations.

Consider an example of two tables in a database, one that encodes a system's users' `emails` (including their username, email address, and whether it's verified) and another that encodes successful `logins` (including a username, timestamp, and ip address):

emails

username	email	verified
samp	samwow@gmail.com	1
samp	samp9@uab.edu	0
karenk	karenk5@uab.edu	1

logins

username	timestamp	ipaddr
samp	1554291414	162.103.150.12
karenk	1554181337	171.31.15.120
karenk	1554219962	155.28.11.102
karenk	1554133720	171.31.15.120

A join operation on these two relations, written `users` ⋈ `logins`, yields a single relation with all five columns: username, email, passhash, timestamp, address. For columns the two relations have in common, the natural join only considers pairs of tuples from the two input relations where the values for those columns match, as in an intersection operation; for other columns, the natural join computes all possible combinations of their values as in Cartesian product. If both input relations share all columns in common, a join is simply intersection and if both input relations share no columns in common, a join is simply Cartesian product. For the above tables, the natural join is shown:

emails ⋈ logins

username	email	verified	timestamp	ipaddr
samp	samwow@...	1	...414	162...
samp	samp9@...	0	...414	162...
karenk	karenk5@...	1	...337	171...
karenk	karenk5@...	1	...962	155...
karenk	karenk5@...	1	...720	171...

For example, if we wanted to compute all email addresses and ip addresses that may be associated, we could compute the join of these two relations and then project the join down to these two attributes alone. Note that one row is removed because it becomes a duplicate after projection:

$$\Pi_{\texttt{email},\texttt{ipaddr}}(\texttt{emails} \bowtie \texttt{logins})$$

email	ipaddr
samwow@gmail.com	162.103.150.12
samp9@uab.edu	162.103.150.12
karenk5@uab.edu	171.31.15.120
karenk5@uab.edu	155.28.11.102

In this example, we've shown relations with associated attribute (column) names (e.g., `email`, `ipaddr`). In our formalization of relations, we treat columns as ordered and identified by their index instead—naturally a programming model, RDBMS, or API for relations will likely associate these indices with their symbolic names. As formalized, the `emails` relation would be a set of tuples:

$$R_{\texttt{emails}} = \{\ (0,0,1),\ (0,1,0),\ (1,2,1)\ \},$$

Where the attributes `username`, `email`, and `verified` are stored in columns 0, 1, and 2, respectively, the string "`samp`" is interned as username 0, the string "`karenk`" is interned as username 1, and the three emails are interned as emails 0, 1, and 2.

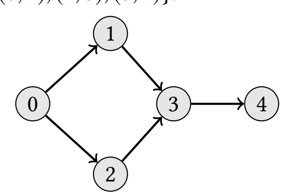
To formalize natural join as a operation on such a relation, we parameterize it by the number of indices that must match, assumed to be the first j of each relation (if they are not, a renaming operation must come first). The join of relations R and S on the first j columns is written $R \bowtie_j S$ and defined:

$$R \bowtie_j S = \{\ (r_0, \ldots, r_k, s_j, \ldots, s_m) \mid (\ldots, r_k) \in R \wedge (\ldots, s_m) \in S \wedge \bigwedge_{i=0..j-1} r_i = s_i\ \}$$

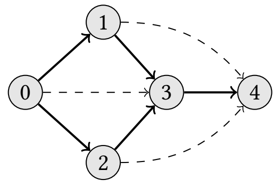
2.2 Application: transitive closure

One of the simplest common algorithms that may be implemented efficiently as a loop over high-performance relational algebra primitives, is computing the transitive closure of a relation or graph. Consider a relation $G \subseteq \mathbb{N}^2$ encoding a graph where each point $(a, b) \in G$ encodes the existence of an edge from node a to node b.

For example, consider graph G (shown below) where $G = \{(0,1),(1,3),(0,2),(2,3),(3,4)\}$.



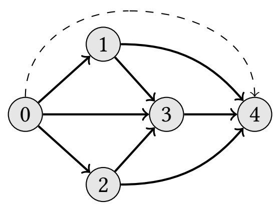
Renaming to swap the columns of G, results in a graph $\rho_{0/1}(G)$ where all arrows are reversed in direction. If this graph is joined with G on only the first column (meaning G is joined on its second columns with G on its first column), we get a set of triples (b, a, c)—specifically $\{(1,0,3),(2,0,3),(3,1,4),(3,2,4)\}$—representing paths of length two in the original graph where a leads to b which leads to c. Projecting out the first column yields pairs (a, c) encoding paths of length two from a to c in the original graph G. If this is unioned with the original G, we obtain a relation encoding paths of length one or two in G. This graph, $G \cup \Pi_{\alpha_1,\alpha_2}(\rho_{0/1}(G) \bowtie_1 G)$, is shown below with new edges (paths of length two) shown in dashes.



We can encapsulate this step in a function F_G which takes as input a relation T encoding a graph and returns the graph G unioned with T's edges extended with G's edges.

$$F_G(T) \stackrel{\Delta}{=} G \cup \Pi_{\alpha_1,\alpha_2}(\rho_{0/1}(G) \bowtie_1 T)$$

The graph shown above can be produced by $F_G(G)$ and the graph G is returned if the input graph T is empty: $F_G(\varnothing)$, or $F_G(\perp)$. If F_G is repeatedly applied, the results encodes ever longer paths through G. In this case for example, the graph $F_G(F_G(G))$ or ${F_G}^3(\perp)$ encodes the transitive closure of G—all paths in G reified as edges.



In the general case, for any graph G, there exists some $n \in \mathbb{N}$ such that ${F_G}^n(\perp)$ encodes the transitive closure of G. The transitive closure may be computed by repeatedly applying F_G in a loop until reaching an n where ${F_G}^n(\perp) = {F_G}^{n-1}(\perp)$ in a process called *fixed-point iteration*. In the first iteration, paths of length 1 are computed; in the second, paths of length 1 or two are computed, and so forth. After the longest path in G is found, just one additional iteration is necessary as a fixed-point check to confirm that the final graph has stabilized.

2.3 Application: Datalog

Computing transitive closure is a simple example of logical deduction. From paths of length 0 (an empty graph) and the existence of edges in graph G, we may deduce the existance of paths of length $0 \ldots 1$. From paths of length $0 \ldots n$ and the original edges in graph G, we may deduce the existance of paths $0 \ldots n+1$ edges long. The function F_G above performs a single round of this inference, finding paths one edge longer than any found previously and exposing new deductions for the next iteration of F_G to make. When the computation reaches its fixed point, a solution has been found because no further paths may be deduced from the available facts.

In fact, the function F_G is just an encoding in relational algebra of the transitivity propery itself, $T(a,b) \wedge T(b,c) \implies T(a,c)$, a logical constraint for which we desire a minimal solution. A graph T satisfies this property exactly when T is a fixed-point for F_T.

Solving logical problems in this way is precisely the strategy of *bottom-up logic programming*. Bottom-up logic programming begins with a set of facts (such as $T(a,b)$—the existence of an edge in a graph T) and a set of inference rules (such as $T(a,b) \wedge T(b,c) \implies T(a,c)$) and performs a fixed-point calculation, accumulating new facts that are immediately derivable, until reaching a minimal set of facts consistant with all rules.

Datalog is a bottom-up logic programming language supporting a restricted logic corresponding to first-order HornSAT—the satisfiability problem for conjunctions of Horn clauses. A *Horn clause* is a disjunction of atoms where all but one is negated: $a_0 \vee \neg a_1 \vee \ldots \vee \neg a_j$. By DeMorgan's laws we may rewrite this as $a_0 \vee \neg(a_1 \wedge \ldots \wedge a_j)$ and note that this is an implication: $a_0 \leftarrow a_1 \wedge \ldots \wedge a_j$. In first-order logic, atoms are predicates with universally quantified variables.

A Datalog program is a set of rules $P(x_0, \ldots, x_k) \leftarrow Q(y_0, \ldots, y_j) \wedge \ldots \wedge S(z_0, \ldots, z_m)$ and its input is a database of facts called the *extensional database* (EDB). Running the datalog program reifies the *intensional database* (IDB) which extends facts from the EDB with all facts transitively derivable via the program's rules.

In the typical notation of datalog, computing transitive closure of a graph is accomplished with just two rules:

```
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y), edge(y,z).
```

The first says that any edge implies a path (taking the role of the left operand of union in F_G), and the second says that any path (x,y) and edge (y,z) imply a path (x,z) (adding edges for the right operand of union in F_G).

Each Datalog rule may be encoded as a function F (between databases) where a fixed point for the function is guaranteed to be a database that satisfies the particular rule. Atoms in the body (premise) of the implication, where two columns are required to match, are refined using a selection operation; e.g., atom $S(a,b,b)$ is computed by RA $\sigma_{\alpha_1=\alpha_2}(S)$. Conjunction of atoms in the body of the implication is computed with a join operation: e.g., in the second rule above, this is the second column of `path` joined with the first of `edge`, or $\rho_{0/1}(\texttt{path}) \bowtie_1 \texttt{edge}$. These steps are followed by projection to only the columns needed in the head of the rule and any necessary column reordering. Finally, the resulting relation is unioned with the existing relation in the head of the implication to produce F's output, an updated database (e.g., with an updated `path` relation in the examples above).

Once a set of functions $F_0 \ldots F_m$, one for each rule, are constructed, Datalog evaluation operates by iterating the IDB to a mutual fixed point for $F_0 \ldots F_m$.

2.4 Implementation approaches

In our previous discussion of both transitive closure and Datalog, we have elided important optimizations and implementation details in favor of focusing on the main ideas of both. In practice, however, it is inefficient to perform multiple granular RA operations separately to perform a selection, reorder columns, join relations, project out unneeded columns, reorder columns again, etc, when iteration overhead can be eliminated and cache coherence improved by performing loop fusion. In practice, high-performance Datalog solvers perform all necessary steps at once, supporting a generalization of the operations we've discussed that can join, select, reorder variables, project, and union, all at once.

In addition, both transitive closure and Datalog, as discussed above, are using naïve fixed-point iteration, recomputing all previously discovered edges (resp. facts) at every iteration. Efficient implementations are *incrementalized*, only considering facts that can be extended to produce previously undiscovered facts. For example, when computing transitive closure, another relation T_Δ is used which only stores the longest paths—those discovered in the previous iteration. When computing paths of length n, in fixed-point iteration n, only new paths discovered in the previous iteration, paths of length $n-1$, need to be considered as shorter paths extended with edges from G yield paths which must have been discovered already. In the more general cases of Datalog and database theory, this optimization is known as *semi-naïve* evaluation.

Now, we review the two main approaches to encoding relations in a manner amenable to fast RA algorithms. Unsurprisingly, algorithms for computing join, union, selection, etc, depend greatly on the representation used for relations themselves.

Decision diagrams. One approach is the use of decision trees to encode relations. *Decision diagrams* (DDs) such as *binary decision diagrams* (BDDs) and its variants, zero-supressed binary decision diagrams (ZDDs) and algebraic decision diagrams (ADDs), are potentially compact representations of relations, predicates, sets of strings, or sets of sets. In a BDD, each variable (column) storing an integer is decomposed into one variable per bit: a relation R(a,b,c) where each column stores a 64bit integer is encoded as a set of binary strings, each 192 bits long. A BDD encodes the decision procedure of determining inclusion of a string (i.e., tuple, fact) in the set as a tree where each node has two subtrees, one for encoding string suffixes that follow a 0, and one for encoding string suffixes that follow a 1. The root node for a BDD encoding relation $R(a,b,c)$ has two subtrees, one for encoding a set of 191-bit strings with a 0 as the initial bit, and one for likewise encoding suffixes with a 1 as the leading bit. The children of leaf nodes are one of two special ($\perp$ and $\top$) nodes that indicate no strings exist with the encoded prefix or that all strings with the encoded prefix are present in the set.

Algorithms for performing RA on decision diagrams recursively merge nodes of the tree according to the operation being performed, taking time proportional to the size of the structures, and can be highly efficient when the tree is compact. Performant DD libraries like CUDD (CU decision diagram library) perform recursive interning under the hood to improve space efficiency []. In practice, decision diagrams can be highly compact or can blow up exponentially, depending largely on variable ordering (i.e., what determines which bits are near the root). A major practical downside of DDs has been the difficulty of automatically determining a efficient variable ordering given that a representation of domain-specific problems in decision diagrams (via an encoding in Datalog) has already thoroughly obfuscated the meaning behind each bit. Much work has gone into simply trying to learn, or dynamically adapt, a DD's variable ordering to the problem at hand.

Key-value stores. Another approach to encoding relations that has recently shown far greater scalability [], although it is both older and apparently less sophisticated, is to use a hash table, B-tree, prefix tree (trie), or other key-value store to maintain a set of tuples, with support for efficient iterators to directly loop over all facts in a relation when performing RA.

In this approach, a join becomes nested iteration over two or more relations to build up a set of output tuples that can be inserted into a new relation encoded as a key-value store. Unioning can be done by simply inserting tuples into an existing relation. Renaming, projection, and selection, can all be done trivially, on-the-fly, while performing a join, as these are as simple as reordering the variables of each tuple before an insertion, omitting values from a tuple before insertion, or omitting tuples that do not qualify before insertion.

In a join operation, it is inefficient to iterate over all tuples of two relations if any variables are unified (if the join is not equivalent to Cartesian product). Instead, only the first relation is iterated over in its entirety and an efficient selecting iterator is used to iterate over only those tuples in the second relation where unified variables match. This is accomplished by using a key-value store for tuple keys that is implemented using nested key-value stores for integer keys. Here again variable ordering becomes crucial as columns being selected on must come first. Unlike for DDs however, the variable ordering issue can be solved precisely and statically (at compile time) as the structure of RA operations being performed implies the necessary indices.

Consider the second Datalog rule implementing transitive closure, discussed previously, defining paths in terms of paths and edges. Each iteration of our function F for this rule can be implemented as the following pseudocode:

```
new_delta_path = {}
for [x,y] in delta_path.select_all():
   for [y,z] in edge.select("y", y):
      if path.insert([x,z]) == true:
         new_delta_path.insert([x,z])
delta_path = new_delta_path
```

3 HASH-TREE RELATIONAL ALGEBRA

This section discusses our implementation of distributed relational algebra. We employ a hybrid approach we call *hash-tree* relational algebra that consists of nesting B-tree key-value stores within a hash-table that can be partitioned across multiple cores or MPI nodes. As discussed in the previous section, the key-value store approach to encoding relations requires nested maps to support efficient join (and select) operations. In a typical join, it will be necessary to iterate over all tuples in a relation where particular columns match a value or values taken from the first operand of join—in which case the variable being matched should come first and be a key in the outermost B-tree. The relation is thus explicitly indexed on this column.

In our implementation, a relation $R(a,b)$ indexed on a is encapsulated in a type `Relation<Relation<void>>` which provides an interface to a B-tree mapping `uint64_t` keys, storing each a, to subrelations over just those values b that are paired with a particular a. In our approach, this nesting of B-trees is extended at the top-level by a hash table so that each value a is also hashed to one of *nproc* (the number of MPI processes being used to host the relation) *buckets*.

Hybrid join. The standard way to parallelize the key-value store approach to relational algebra on multi-core systems is to partition the iteration space of the outermost loop. For example, the Soufflè Datalog solver uses OpenMP to parallelize its join operations, first partitioning the outermost key-value store into multiple disjoint iterators—one for each available thread. Soufflè's join algorithm is nearly identical to our previous pseudocode for join, except that it adds an outer parallel for loop.

```
new_delta_path = {}
pfor part_iter in delta_path.partition():
```

```
        for [x,y] in part_iter:
            for [y,z] in edge.select("y", y):
                if path.insert([x,z]) == true:
                    new_delta_path.insert([x,z])
    delta_path = new_delta_path
```

3.1 Distributed hash-tree join

Our hybrid hash-tree structure for relations makes this partitioning explicit as physically separate relations stored in distinct hash-table buckets—each owned by a dedicated MPI process. Join operations then decompose into a separate join for each bucket, followed by hashing of output tuples, and a communication phase to insert these tuples into the output relation. In experiments designing our join operation, we found that, as hashing distributes key values evenly across buckets, MPI's all-to-all communication paradigm was actually most efficient for inserting output tuples in their receiving buckets (i.e., on processes hosting in the output relation).

In Figure 1 we show the process for computing a single iteration of a distributed transitive closure computation. T (i.e., T_Δ as we implement an incrementalized TC algorithm) is joined with G on a per-bucket basis (the diagram shows three color-coded buckets). Tuples in $T(x, y)$ are indexed on the second column (y) and tuples in $G(y, z)$ are indexed on the first column (y). This makes it possible to perform local (intra-bucket) joins as each tuple $(x, y) \in T$ is guaranteed to have all matching tuples $(y, z) \in G$ stored in the same bucket, managed by the same MPI process. Each resulting triple (x, y, z) has its middle column projected out on-the-fly as it is produced, and, as the resulting tuple (x, z) must be inserted into T, a relation indexed on its second column, each new edge is hashed and assigned to bucket *hash*(z)%*nprocs* in the output relation (T). As this bucket will likely not be managed by the local MPI process, a communication phase is required to actually perform insertion of each output tuple in its output relation. As each output tuple is produced it is staged in one of *nprocs* packets, ready to be sent across the network to the MPI process managing its bucket. Finally, an all-to-all communication phase is used to reorganize the output of the join operation—preparing T for subsequent fixed-point iterations. As tuples are received by their host process, they are inserted into the local B-tree structure, eliminating duplicates.

3.2 Distributed hash-tree union

We present two algorithms for distributed unions, naïve hash-tree union and buffered hash-tree union. As the name suggests buffered hash-tree union buffers data across all relations that need to be unioned before performing any communication or insertions and performs the union concurrently in a single step. While performing a union of n relations, naïve hash-tree union involves n epochs of communication and computation—one for each relation—as opposed to buffered hash-tree union that uses buffering to limit the number of communication and computation epochs to one.

Naïve hash-tree union is concurrent in processing each relation being unioned, but unions each relation in a separate step. Naturally, our buffered implementation has an extra memory overhead as opposed to the naïve implementation where all graphs that need to be unioned are processed one at a time, however it is more representative of real applications (such as Datalog, program analysis, etc) where each relation being unioned would have a set of nodes hosting it and tuples across multiple relations could all be transmitted and inserted into a new relation concurrently.

The input graphs can be read from files stored on the disk or can be read directly from memory. If the graphs are read from disk, the first phase is that of parallel I/O: processes access disjoint regions of the file to read an equal number of tuples in parallel. Once the tuples are read into memory, each process scans through its tuples and hashes them, grouping tuples into *nprocs* packets, ready to be sent across the network. The target process (i.e., outer hash-bucket) of a tuple is computed based on the hash value of its key. For instance, the target rank of a two column tuple (a, b), where a is the outer key and b the inner key, would be *hash*(a)%*nprocs*. We also perform preliminary deduplication, as these tuples are staged, before transmitting these batches of tuples to minimize communication overhead. The grouping step is followed by an all-to-all communication phase where tuples are sent to the appropriate processes (outer hash buckets). Once tuples arrive at a process, they are inserted into the local relation container. This step performs the important task of deduplication of tuples across relations.

4 EVALUATION

The ultimate goal of this section is to evaluate the performance of our hash-tree implementations of join, union and transitive closure at scale. We start by individually studying the computation and communication components of the RA operations. Computation is dominated by insertion of tuples and the major challenge faced is that of deduplication. We therefore study the efficacy of our btree-based relation container in the context of insertions. All our RA operations involve an all to communication phase, hence, we perform a detailed benchmark of MPI's all to all communication capability. Finally we benchmark the efficacy of distributed hash-tree union, parallel join and parallel transitive closure over a wider range of graphs.

4.1 Dataset and HPC platforms

We have performed our experiments using SuiteSparse Matrix Collection available at []. SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection), is a large and actively growing set of sparse matrices that arise in real applications. The Collection is widely used by the numerical linear algebra community for the development and performance evaluation of sparse matrix algorithms. For our experiments, we chose seven graphs (listed in Table **??**) representing a wide range in terms of the number of edges. Transitive closure of a graph with n edges can generate upto n^2 edges (a fully connected graph). The number of edges in the transitive closure of a graph depends on how connected the input graph is. For example, the transitive closure of our third graph with $2,100,225$ edges generated a graph with $276,491,930,625$ edges, amounting to 4.4 terabytes of data.

The experiments presented in this work were performed on Theta Supercomputer at the Argonne Leadership Computing Facility (ALCF). Theta is a Cray machine with a peak performance of X petaflops, Y compute cores, Z TiB of RAM, and A PiB of online disk storage. The supercomputer has Dragonfly network topology and a Lustre filesystem.

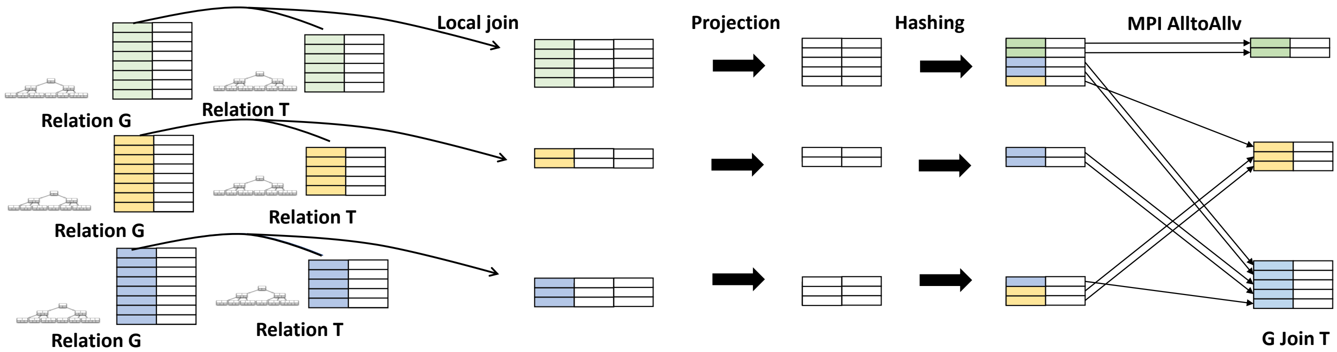


Figure 1: Schematic Diagram to show different phases of hash-tree join.

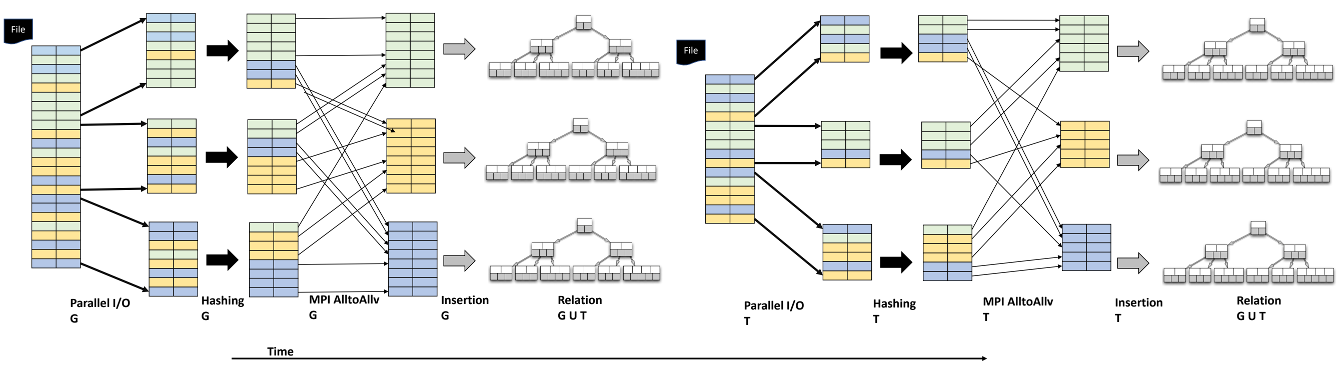


Figure 2: Schematic Diagram to show different phases of naive hash-tree union.

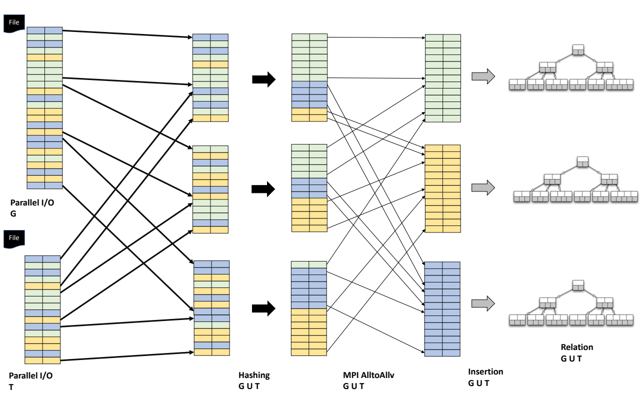


Figure 3: Schematic Diagram to show different phases of buffered hash-tree union.

4.2 Btree-Relation container

In this section we evaluate the efficacy of our relation container. We measure performance for two cases: insertions of unique tuples and insertions of tuples with duplicates. With the later set of experiments, every tuple had four duplicates. For both set of experiments, we compare two implementations of the relation class, one with a Btree back-end and the other with an hash map back-end. For the

Input graph edge count	Union	Join	Transitive Closure	Graph name
412148	✓			
2100225	✓			
6291408	✓			
59062957	✓			
136024430	✓	✓		
180292586	✓	✓		
240023949	✓			

Table 1: List of seven graphs used in our evaluation.

hash map, we used unordered_map from C++'s standard template library. The results can be seen in Figure 4. The X-axis corresponds to the total number of tuples being inserted and the Y-axis is the time taken for the task to finish. We observe that btree-based relation container outperforms the hash map based implementation for insertion of all tuple counts. Furthermore hash map based relation container fails to scale with insertion of very large number of tuples. For example hash based relation takes X seconds to insert Y tuples as opposed to only Z seconds taken by btree based relation. Similar results can be observed for insertion of tuples with duplicates. The btree based relation container successfully dedeuplicate tuples while maintaining high performance.

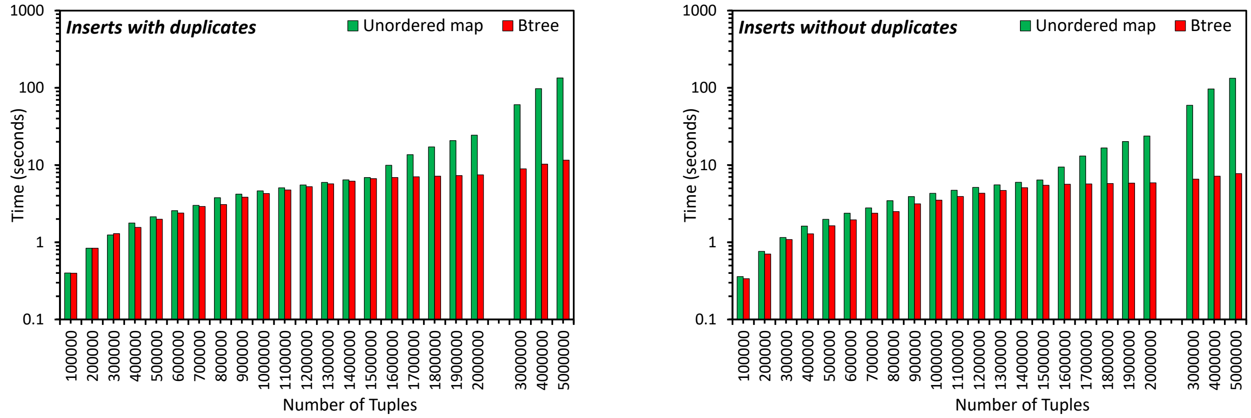


Figure 4: Performance evaluation of relation class implemented with btree and unordered map. (left) All tuples are distinct, (right) There are four duplicates of every tuple. Relation implemented with btree out-performs the unordered-map implementation for both cases.

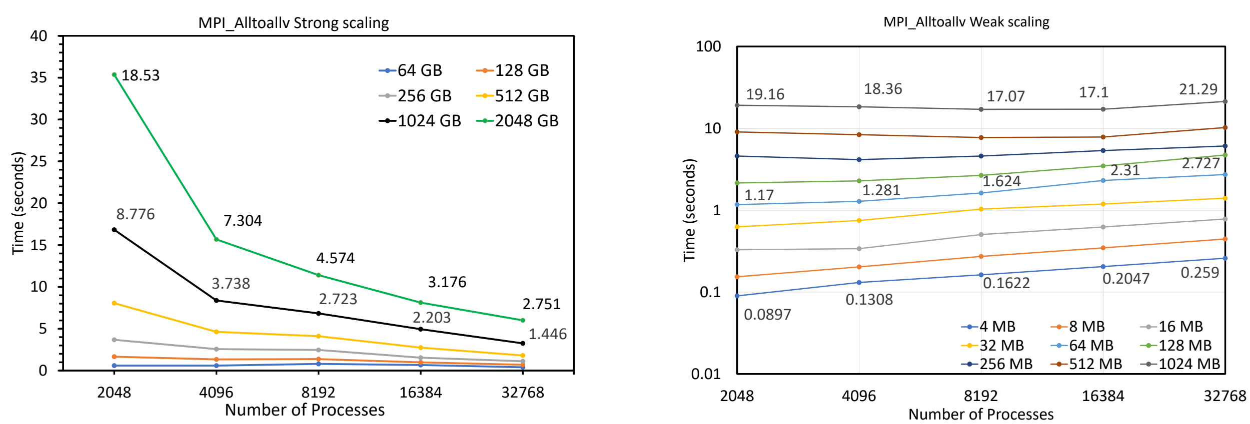


Figure 5: Strong (left) and Weak (right) scaling evaluation of MPI_alltoallv function of MPI.

4.3 MPI_All_to_Allv

All to all communication forms the core of all our RA algorithms (**refer to the algorithms**). We use MPI's MPI_Alltoallv function to facilitate all to all data communication. MPI_Alltoallv sends data from all to all processes where each process can send a different amount of data by providing displacements for the input and output data. In this section we study both weak and strong scaling characteristics of MPI_Alltoallv. For both set of experiments, we varied the number of processes from 2048 to 32768. We performed 9 set of weak scaling experiments. For these 9 set of experiments, the amount of data transmitted by each process ($data_{process}$) was varied from 4 megabytes (1^{st} set) to 1,024 megabytes (9^{th} set). For a n process run, every process transmits $data_{process}/n$ units of data to every other process. With strong scaling experiments, we performed 6 set of experiments, varying the total amount of data generated across processes ($data_{total}$) from 64 gigabytes (1^{st} set) to 2,048 gigabytes (6^{th} set). The amount of data generated by every process is the same, for example for a n process run, and $data_{total}$ total amount of data, every process produces $data_{total}/n$ units of data. A process then transmits $data_{total}/n^2$ units of data to every other process. The results of both weak and strong scaling experiments can be seen in Figure 5.

For both strong and weak scaling runs, we observe a decline in performance with overall decreasing workload. For instance, with strong scaling, the 6th set of experiments where total workload is 2, 048 gigabytes, we observe near perfect scaling when the number of processes is doubled from 2048 (18.5 seconds) to 4096 (7.3 seconds) to 8192 (4.5 seconds). After 8, 192 processes, although total time continues to come down with increasing process count, we observe that the rate becomes much slower. Furthermore, looking at the 6^{th} set of experiments where total workload is 64 gigabytes we observe relatively poor scaling characteristics across the entire process range. Both the observations can be attributed to an overall reduction in per-process workload. With less amount of data to transmit, total time is dominated by initialization costs as opposed to actual data transmission cost. For example, with total workload of 64 gigabytes, at 4, 096 processes, every process gets workload of only 16 ($64gigabytes/4096$) megabytes, and ends up sending 8 ($16megabytes/4096$) kilobytes of data to every other process. Similarly for weak scaling experiments as well, when the amount of data exchanged is substantial, we see almost perfect scaling. For example, when the amount of data transmitted by every process is 1024 megabytes, we observe almost perfect scaling, whereas when the amount of data sent by every process is small ($4megabytes$) we observe poor scaling.

Within the context of communication requirements of RA operations, we find the scaling trends of MPI_alltoallv to be encouraging. In general for a given workload (graph size for RA operations), there will always be a range of processes that exhibits good all to all scaling characteristics. We will have the challenging task to identify the right process count that balances the tradeoff between computation and communication. As we will see later in section 4.6 with larger per-process workload computation cost dominates as opposed to smaller per-process workload where total cost is dominated by communication.

4.4 Distributed Union

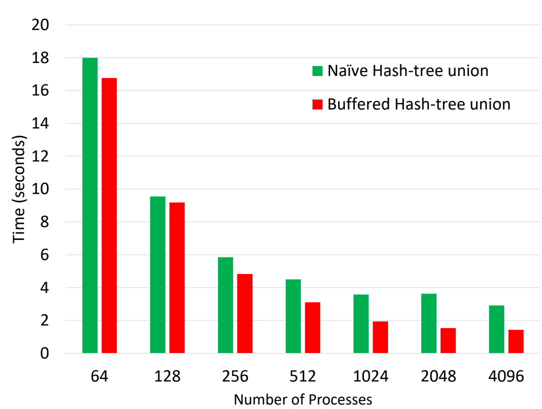


Figure 6: Strong scaling result of hash-tree Union.

We use strong scaling to benchmark the performance of our distributed hast-tree union. We measure the time to union all 7 graphs listed in table ??. The number of processes are varied from 64 to 4, 096. The total number of edges across all 7 graphs is 664, 659, 334 (9.9 gigabytes of data). Union of the 7 graphs result in a graph with 424, 592, 810 edges, indicating significant overlap among the graphs.

We benchmark the performance of both buffered-hash tree join and naive hash-tree join. Performance of both techniques can be seen in Figure 6. We observe two trends: at all core counts, buffered hash-tree union outperforms naive hash-tree unions. This trend can be attributed to an optimized data communication phase associated with buffered unions. Naive technique leads to communication involving small number of large sized data packets as opposed to buffered technique that involves communication with large number of small sized data packets. The other crucial trend is that the union phase scales only till 1024 cores, this can be attributes to increase in communication time at higher core counts, associated with movement of small sized data packets. This result is in corroboration with the trend seen in Section ??. At 2048 and 4096 processes, even though the insertion time gets reduced, the per-process workload becomes small impeding scalability of the communication phase.

It can be concluded that for this particular union task (7 graphs) 1024 is the ideal degree of parallelism, as that balances the communication and computation task best.

4.5 Distributed Join

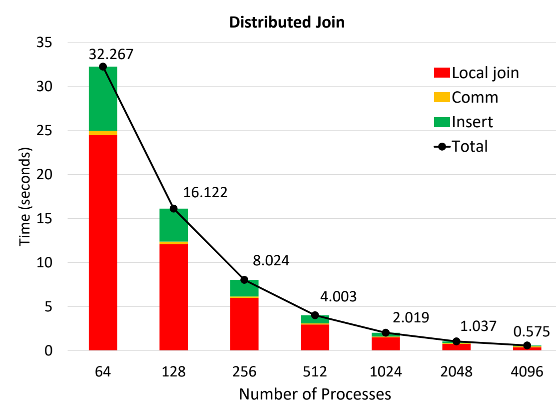


Figure 7: Strong scaling result of hash-tree join.

Similar to distributed union, we use strong scaling to benchmark the performance of distributed join as well. We perform join of two graphs with edges 136, 024, 430 and 180, 292, 586. The join operation yields a graph with X number of edges. The number of processes are varied from 64 to 4, 096. Once the two relations are initialized across all processes (after parallel I/O, hashing, communication and insertions), we initiate the join operation. We plot the result of join operation in Figure 7. We observe trends similar to distributed union. Join only scales upto X cores, after which communication cost starts to dominate and we observe overall decline in performance.

4.6 Transitive closure

Computing the transitive closure of a graph involves repeated join operations until a fixed point is reached. Iteration (i) adds new paths of length i, until a fixed point is reached and no new edges are added. We use the hash-join (**TODO: refer** to distribute the tuples across all processes. Every iteration comprises of four phases: 1) Join 2) network communication 3) insertion 4) checking for a fixed point. In the join phase every process concurrently computes the join of relations G and $T = \rho_{0/1}(G)$ which creates new edges. Following the semi-naive evaluation (discussed in Section **??**), the new edges are then added back to relation T. This step of adding newly created edges entail all to all communication. The new edges need to be re-hashed and sent to appropriate hash-buckets (processes). We accomplish this all-to-all communication using MPIâĂŹs MPI_alltoallv routine where newly created edges are transmitted to all the processes. Following the insertion phase, the newly created edges are inserted to the relation T. In the final step we check if the size of the relation T changed on any process, if it does then we have not yet reached a fixed point and we continue to another iteration of these 4 steps.

We performed a set of strong-scaling experiments to compute the transitive closure of graph with 412148 edgesâĂŤthe largest graph in the U. Florida Sparse Matrix set (Davis and Hu 2011). We used the Quartz supercomputer at the Lawrence Livermore National Laboratory (LLNL). For our runs, we varied the number of processes from 64 to 2048. A fixed point was attained after 2933 iterations, with the resulting graph containing 1676697415 edges. As can be seen in Figure 1, our approach takes 462 seconds at 64 cores and 235 seconds at 2048 cores, corresponds to an overall efficiency of 6.25of by the four major components (join, network communication, join, fixed-point check) of the algorithm. We observe (see Figure 2) that for all our runs the total time is dominated by computation rather than communication; insert and join together tended to take up close to 90as it shows that we are not bound primarily by the network bandwidth (at these scales and likely moderately higher ones) and it gives us the opportunity to optimize the computation phase

5 RELATED WORK

Lorem ipsum dolar sit amat

6 CONCLUSION

Lorem ipsum dolar sit amat

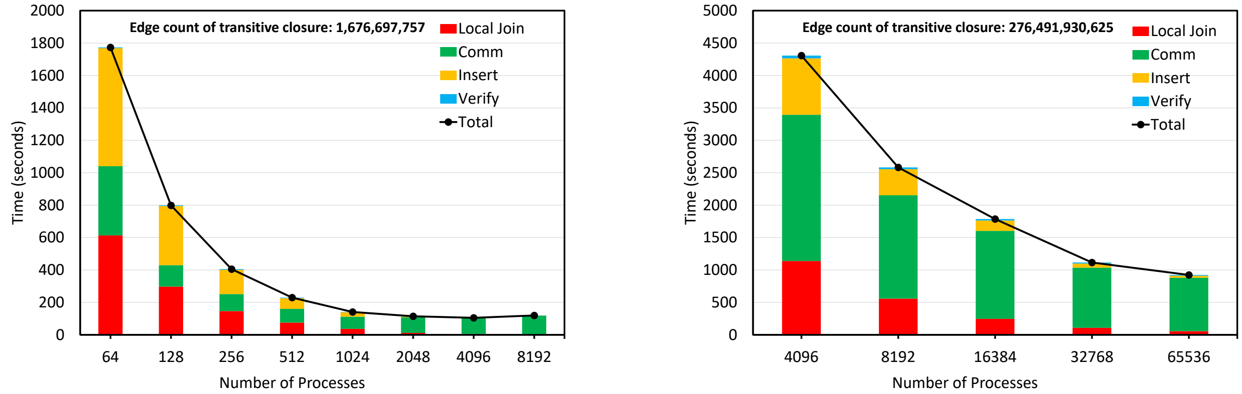


Figure 8: Transitive closure of a graph with X edges.