# 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. Relational Algebra at Scale. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 8 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, $\alpha_i$ and $\alpha_j$ 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 $\alpha_i$ 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 $\alpha_i$ and $\alpha_j$ 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` $\bowtie$ `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 $\bowtie$ 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.

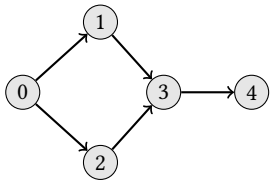
To formalize natural join as a operation on such a relation, we parameterize it by the number of indices that must match, assumed to be the first $j$ of each relation (if they are not, a renaming operation must come first). The join of relations $R$ and $S$ on the first $j$ columns is written $R \bowtie_j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

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

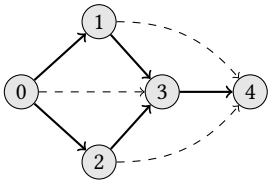
## 2.2 Application: transitive closure

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 $G = \{(0,1), (1,3), (0,2), (2,3), (3,4)\}$.



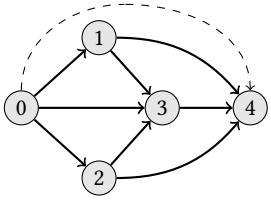

Renaming to swap the columns of G, results in a graph $\rho_{0/1}(G)$ where all arrows are reversed in direction. If this graph is joined with G on only the first column (meaning G is joined on its second columns with G on its first column), we get a set of triples $(b, a, c)$—specifically $\{(1,0,3), (2,0,3), (3,1,4), (3,2,4)\}$—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.

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

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 $F_G(\bot)$. 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 ${F_G}^3(\bot)$ 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 ${F_G}^n(\bot)$ 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 ${F_G}^n(\bot) = {F_G}^{n-1}(\bot)$ 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 *intensional database* (IDB). Running the datalog program yields the *extensional database* (EDB) which extends all facts from the IDB 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 $\rho_{0/1}$(`path`) $\bowtie_1$ `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 $F_0 \ldots F_m$.

# 3 IMPLEMENTATION

This section discusses implementing relational algebra efficiently and in parallel. We take a hybrid approach of nesting key-value stores within a hash-table that can be partitioned across multiple cores or nodes.

## 3.1 Representations

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

*Decision diagrams.* Decision diagrams such as binary decision diagrams (BDDs) and zero-supressed binary decision diagrams (ZDDs) are compact representations of relations as decision trees. Each variable (column) storing an integer is broken down 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.

*Key-value stores.* Another approach to encoding relations is to use a hash table, B-tree, prefix tree (trie), or other key-value store.

*Hybrid hash-table and b-tree.* Our approach is to use an efficient key-value store, but to

### 3.2 Hybrid Join

...

### 3.3 Distributed Join

...

### 3.4 Distributed Union

We present two algorithms for distributed unions. In the first approach we iterate through all graphs that needs to be unioned, and hash all the tuples

## 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.

### 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 hash map, we used unordered_map from C++'s standard template library. The results can be seen in Figure 1. 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.

| 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.**

### 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 2.

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

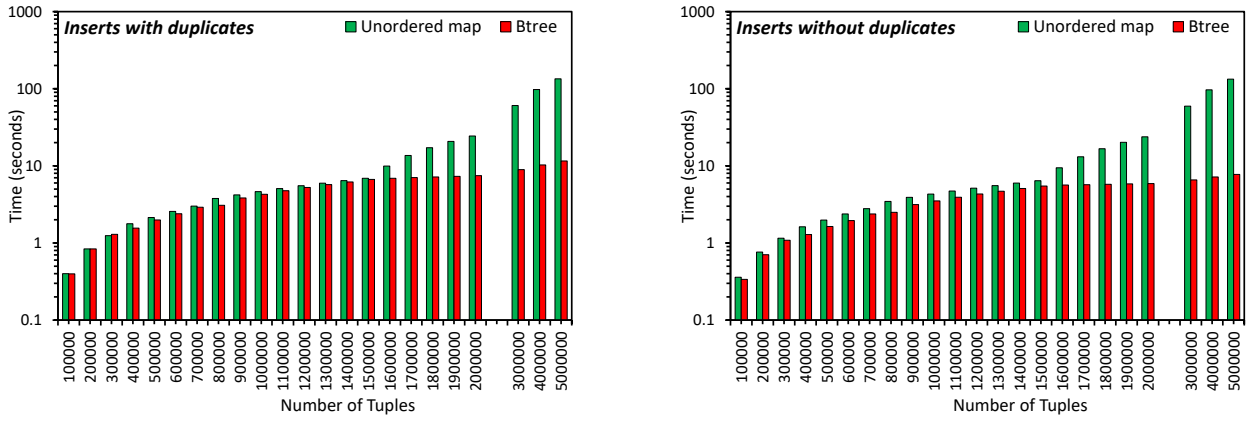



**Figure 1: Performance evaluation of relation class implemented with btree and unordered map. (left) All tuples are distinct, (right) There are four duplicates of every tuple. Relation implemented with btree out-performs the unordered-map implementation for both cases.**

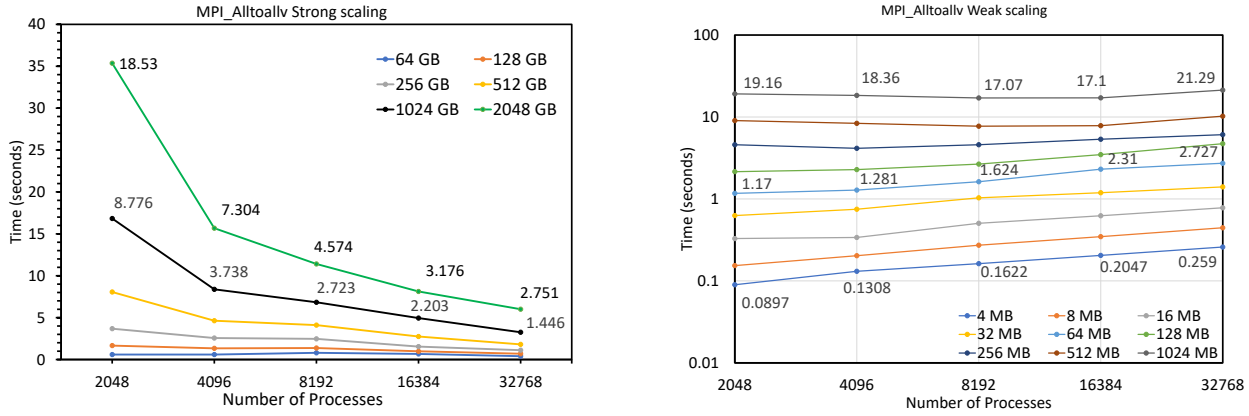



**Figure 2: Strong (left) and Weak (right) scaling evaluation of MPI_alltoallv function of MPI.**

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 ($64 gigabytes/4096$) megabytes, and ends up sending 8 ($16 megabytes/4096$) 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 ($4 megabytes$) 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

We use strong scaling to benchmark the performance of distributed union operation. We measure the timing to perform union of 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 seven graphs result in a graph with 424,592,810 edges, indicating significant overlap between the graphs.

We benchmark the performance of both union_x and union_y. Union_x involves seven epochs of communication and computation (one for every graph) as opposed to union_y that uses buffering to limit the number of communication and computation epochs to one. With union_x, all processes iterate through all seven graphs. The first phase is that of parallel I/O, where processes access disjoint regions of the file to read equal number of tuples in parallel. Once the tuples are read, every process scans through the tuples and groups them into *nprocs* (=*hashbuckets*) packets, ready to be sent across the network. Target process (hash-bucket) of a tuple is computed based on the hash outcome of its key. For instance, target rank of a two column tuple $(a, b)$ would be $hash(a)\%nprocs$. We also perform preliminary deduplication to eliminate duplicate tuples in the input graph. The scan step is followed actual by all to all communication phase where tuples are sent to the appropriate processes (hash buckets). Once tuples arrive at a process, they are inserted into the relation container. This step performs the important task of deduplication of tuples across the graphs.

With union_y instead of processing the graphs one after the other, we read all the graphs at once, buffer the tuples and follow it with one cumulative step of hashing, communication and insertion. This implementation deploying buffering restricts the number of communication epochs. Performance of both implementation can be seen in Figure 3. For both implementations, we present breakdown of time spent in each of the three phases: parallel I/O, all-to-all communication and insertions. We observe two trends: at all core counts, union_y outperforms union_x, this trend can be attributed to an optimized data communication phase. Union_y leads to communication involving small number of large sized data packets as opposed to union_x 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. This result is in corroboration with the trend seen in Section 4.1. At 2048 and 4096 processes, even though the insertion time gets reduced, the per-process workload becomes very 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

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 4. 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 rehashed 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

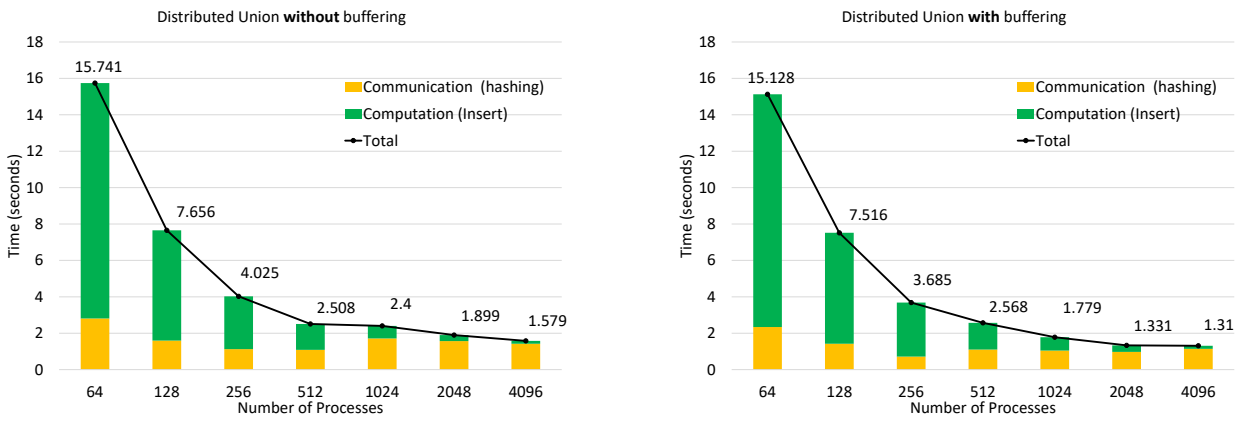



**Figure 3: Distributed union.**

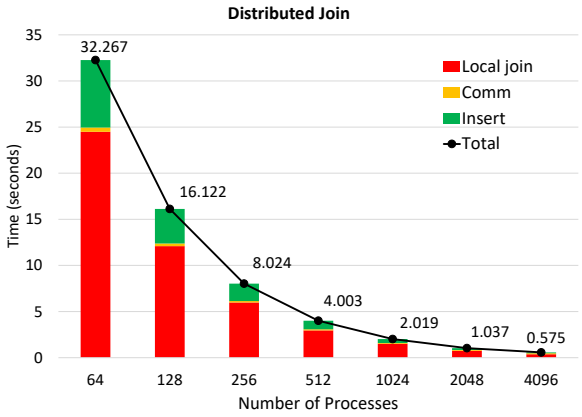



**Figure 4: Transitive closure of a graph with X edges.**

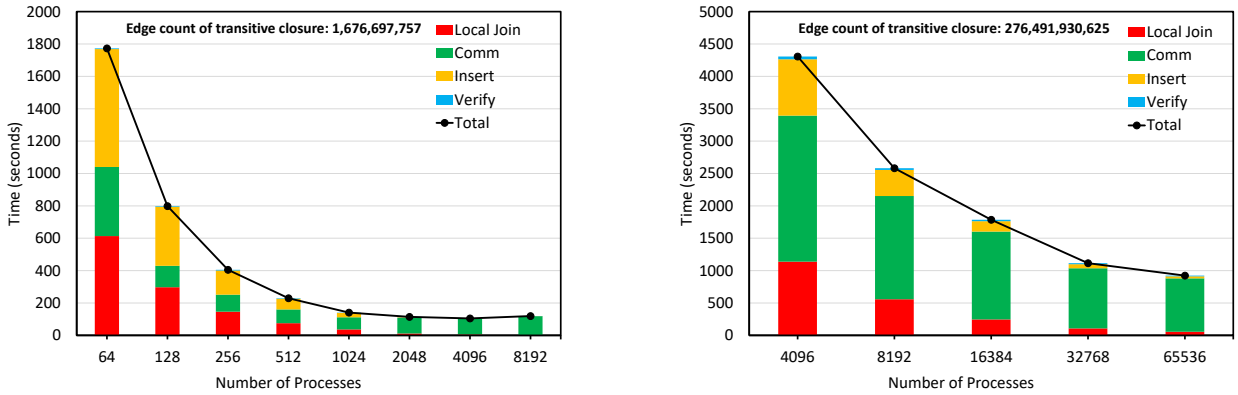



**Figure 5: Transitive closure of a graph with X edges.**