Load-balancing Parallel Relational Algebra

Sidharth Kumar and Thomas Gilray

University of Alabama at Birmingham
{sid14,gilray}@uab.edu

Abstract. Relational algebra (RA) comprises a basis of important operations, sufficient to power state-of-the-art reasoning engines for Datalog and related logic-programming languages. Parallel RA implementations can thus play a significant role in extracting parallelism inherent in a wide variety of analytic problems. In general, bottom-up logical inference can be implemented as fixed-point iteration over RA kernels; relations dynamically accumulate new tuples of information according to a set of rules until no new tuples can be discovered from previously inferred tuples and relevant rules (RA kernels). While this strategy has been quite successful in single-node contexts, it poses unique challenges when distributed over many-node, networked clusters—especially regarding how the work-load is balanced across available compute resources.
In this paper, we identify two fundamental kinds of load imbalance and present a strategy to address each. We investigate both spacial load imbalance—imbalance across each relation (across compute nodes)—and temporal load imbalance–imbalance in tuples produced across fixed-point iterations. For spacial balancing, we implement refinement and consolidation procedures. For temporal balancing, we implement a technique that permits the residual workload from a busy iteration to roll over to a new iteration. In sum, these techniques permit fully dynamic load-balancing of relational algebra that is robust to changes across time.

Keywords: Parallel Relational Algebra · Load Balancing · Logic Programming · Message Passing Interface · All-to-all communication.

1 Introduction

Relational algebra (RA) comprises an important basis of operations. It can be used to implement a variety of algorithms in satisfiability and constraint solving [16], graph analytics [19], program analysis and verification [15], deductive databases [14], and machine learning [17]. Many of these applications are, at their heart, cases of logical inference; a basis of performant relational algebra is sufficient to power state-of-the-art forward-reasoning engines for Datalog and related logic-programming languages.

Quite recently, some efforts [4, 12] have explored methods for exploiting the massive parallelism available on modern clusters in a single relational-algebra operation, making it possible to extract data-parallelism across individual operations for relations at scale. Instead of only decomposing tasks in a broader

logical inference problem, such approaches could permit extreme scaling for problems involving only a small number of distinct tasks in a principled manner. A fundamental problem, such approaches must contend with, is that of inherent imbalance possible among the relation data. For example, a *join* of two relations (an operation that generalizes both Cartesian product and intersection by matching only specified columns) may most naturally be decomposed across many processes or threads by grouping like keys on like processes, permitting most of the join to be parallelized completely. If a relation is imbalanced among its keys (exhibits *"key skew"*), this imbalance will also be represented in the decomposition of the join operation, which is highly undesirable for performance.

In this paper, we discuss both the problem of dynamic changes in spacial imbalance, where operations on relations become imbalanced due to key skew in the relation itself, and the problem of temporal imbalance, where operations on relations may vary significantly in their output when repeated. While past work has given mention of replication-based strategies for remediating spacial imbalance [4] and has implemented *static* replication strategies [12], no existing approach offers a solution that is robust to arbitrary changes in relation balance across time, or to sudden explosions in operation output.

We make three novel contributions to the literature on effectively parallelizing relational algebra:

- We explore spacial load imbalance in relational algebra and present two techniques for balancing relations across MPI processes at scale.
- We explore temporal load imbalance in relational algebra and present an iteration-buffering technique for mitigating against the effects of explosions in workload and guarding against resultant failures.
- We present an evaluation of our two approaches, together and in isolation, using random, real world, and corner-case relation topologies, illuminating a space of tunings and demonstrating effectiveness using 256−32,768 processes.

In Section 2, we describe background and related work on relational algebra. In Section 3, we discuss our implementation using MPI, describe two kinds of load imbalance, and present three techniques for mitigating such load imbalance. In Section 4, we present an evaluation of our approaches using the Theta supercomputer and discuss tuning the framework and our observations.

2 Parallel Relational Algebra

This section reviews some standard relational operations such as union, product, intersection, natural join, selection, renaming, and projection, along with their use in implementing algorithms for bottom-up logical inference engines. We discuss existing approaches to parallelizing relational algebra (RA) on single-node systems, and then on many-node systems.

We make a few typical assumptions about relational algebra that diverges from those of traditional set operations. Specifically, we assume that all relations are sets of flat (first-order) tuples of integers with a fixed, homogeneous arity.

Although our approach may extend naturally to relations over any enumerable domains (e.g., booleans, symbols/strings, etc), we also assume that such values are interned and assigned a unique enumerated identity. Several RA operations are especially standard and form a simple basis for computation. *Cartesian product* is defined as for sets except only yields flat first-order tuples and never nested tuples. The *Union* and *Intersection* of two relations are defined as for sets except that these operations requires both relations to have the same arity.

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 relation R restricts R to a particular set of columns $\alpha_0, \ldots, \alpha_j$, where $\alpha_0 < \ldots < \alpha_j$, and is written $\mathit{\Pi}_{\alpha_0,\ldots,\alpha_j}(R)$. For each tuple, projection retains only the specified columns. *Renaming* is a unary operation that renames columns (i.e., reorders columns, as column names are just their index). Renaming columns can be defined in different ways, including renaming all columns at once. We define a 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 may be repeated to rename/reorder any number of columns. In practice, our implementation offers a combined projection and reordering operation that generalizes these two operations more effeciently.

$$\mathit{\Pi}_{\alpha_0,\ldots,\alpha_j}(R) \triangleq \{(r_{\alpha_0}, \ldots, r_{\alpha_j}) \mid (r_0, \ldots, r_k) \in R\}$$
$$\rho_{\alpha_i/\alpha_j}(R) \triangleq \{(\ldots, r_{\alpha_j}, \ldots, r_{\alpha_i}, \ldots) \mid (\ldots, r_{\alpha_i}, \ldots, r_{\alpha_j}, \ldots) \in R\}$$

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

G

0	1
A	**B**
A	**C**
B	**D**
C	**D**
D	**E**

G joined with G

0	1	2
A	**B**	**D**
A	**C**	**D**
B	**D**	**E**
C	**D**	**E**

$\rho_{0/1}(\rho_{0/1}(G) \bowtie_1 G)$

Fig. 1: Graph G and the result of G-join-G.

Consider a relation G, shown in Figure 1, as a table, and at the top of Figure 2, as a graph. Joining G on its second column with G on its first column yields a new relation, with three columns, encoding all paths of length 2 through the graph G, where each path is made of three nodes in order.

To formalize natural join as an operation on such a relation, we parameterize it by the number of columns 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:

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

Note that to compute the join of G on its second column with G on its first column, we first have to reverse G's columns, computing $\rho_{0/1}(G)$, so we may then compute a join on one column: $\rho_{0/1}(G) \bowtie_1 G$. To present the resulting paths of length two in order again, we may use renaming to swap the join column back to the middle position, as shown in Figure 1. Our implementation (detailed in Section 3) provides more general operations that make this administrative renaming occur only implicitly, on the fly.

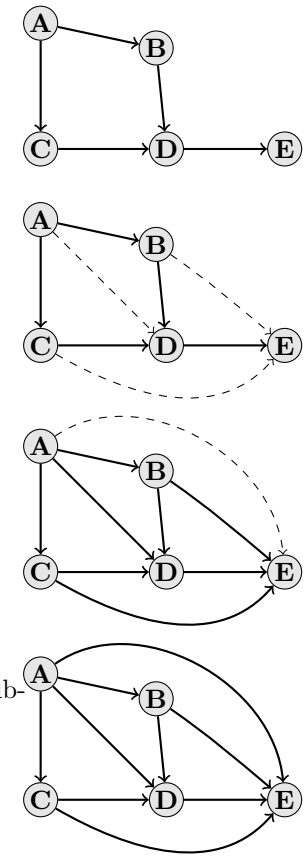


Fig. 2: Each iteration of computing transitive closure for a small example relation G.

2.1 Motivation: Logical Inference

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* (TC) of a relation or graph. For example, consider our example graph $G \subset \mathbb{N}^2$ where each symbol **A** through **E** has been encoded or interned as an integer: $G = \{(0_{\mathbf{A}}, 1_{\mathbf{B}}), (1_{\mathbf{B}}, 3_{\mathbf{D}}), (0_{\mathbf{A}}, 2_{\mathbf{C}}), (2_{\mathbf{C}}, 3_{\mathbf{D}}), (3_{\mathbf{D}}, 4_{\mathbf{E}})\}$ (a subscript shows each integer's interpretation as a symbol or vertex name). 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), via $\rho_{0/1}(G) \bowtie_1 G$, we get a set of triples (b, a, c)—specifically $\{(1_{\mathbf{B}}, 0_{\mathbf{A}}, 3_{\mathbf{D}}), (2_{\mathbf{C}}, 0_{\mathbf{A}}, 3_{\mathbf{D}}), (3_{\mathbf{D}}, 1_{\mathbf{B}}, 4_{\mathbf{E}}), (3_{\mathbf{D}}, 2_{\mathbf{C}}, 4_{\mathbf{E}})\}$—that encode paths of length two in the original graph where a leads to b which leads to c. Projecting out the initial column, b, with $\Pi_{1,2}(\rho_{0/1}(G) \bowtie_1 G)$ yields pairs (a, c) encoding paths of length two from a to c in the original graph G. (Note that this projection step not only removes a column, but a row as well as $(1_{\mathbf{B}}, 0_{\mathbf{A}}, 3_{\mathbf{D}})$ and $(2_{\mathbf{C}}, 0_{\mathbf{A}}, 3_{\mathbf{D}})$ are duplicates if not differentiated by their middle, b, node). If we compute the union of this graph with the original G, we obtain a relation encoding paths of length one or two in G. This graph, $G \cup \Pi_{1,2}(\rho_{0/1}(G) \bowtie_1 G)$, is second from the top in Figure 2 with new edges styled as dashed lines.

We can encapsulate this step in a function Extend_G which takes a graph T, and returns T's edges extended with G's edges, unioned with G.

$$\mathit{Extend}_G(T) \triangleq G \cup \Pi_{1,2}(\rho_{0/1}(T) \bowtie_1 G)$$

The original graph G, at the top of Figure 2, is yielded for $\mathit{Extend}_G(\bot)$, and the graph below it is returned for $\mathit{Extend}_G{}^2(\bot)$, and the graph below that is

returned for $Extend_G{}^3(\bot)$, etc. As $Extend_G$ is repeatedly applied from an empty input, each result encodes ever longer paths through G, as shown. In this case for example, the graph $Extend_G{}^3(\bot)$ encodes the transitive closure of G—all paths in G reified as edges. One final iteration, computing $Extend_G{}^4(\bot)$, is required to check that the process successfully reached a fixed point for $Extend_G$.

In the general case, for any graph G, there exists some $n \in \mathbb{N}$ such that $Extend_G{}^n(\bot)$ encodes the transitive closure of G. The transitive closure may be computed by repeatedly applying $Extend_G$ in a loop until reaching an n where $Extend_G{}^n(\bot) = Extend_G{}^{n-1}(\bot)$ in a process of *fixed-point iteration*. In the first iteration, paths of length one are computed; in the second, paths of length one 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 in this process of path inference.

Computing transitive closure is a simple example of logical inference. From paths of length zero (an empty graph) and the existence of edges in graph G, we deduce the existence of paths of length $0 \ldots 1$. From paths of length $0 \ldots n$ and the original edges in graph G, we deduce the existence of paths of length $0 \ldots n+1$. The function $Extend_G$ above performs a single round of inference, finding paths one edge longer than any found previously and exposing new deductions for a next iteration to make. When the computation reaches a fixed point, the solution has been found as no further paths may be deduced from available facts. In fact, the function $Extend_G$ is a quite-immediate encoding, in relational algebra, of the transitivity property itself, $T(a,c) \Longleftarrow G(a,c) \vee T(a,b) \wedge G(b,c)$, a logical constraint for which we desire a least solution. T satisfies this property *exactly* when T is a fixed-point for $Extend_G$ and the transitive closure of G.

This kind of logical inference forms the semantics of *Datalog*, a bottom-up logic-programming language supporting a restricted logic corresponding roughly to first-order HornSAT—the SAT problem for conjunctions of Horn clauses [2]. 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 initial facts called the *extensional database* (EDB). Running the datalog program makes explicit the *intensional database* (IDB) which extends facts in the EDB with all facts transitively derivable via the program's rules. In the usual Datalog notation, computing transitive closure of a graph is accomplished with two rules:

```
T(x,y) <- G(x,y).          T(x,z) <- T(x,y), G(y,z).
```

The first rule says that any edge, in G, implies a path, in T (taking the role of the left operand of union in $Extend_G$ or the left disjunct in our implication); the second rule says that any path (x, y) and edge (y, z) imply a path (x, z) (adding edges for the right operand of union in $Extend_G$). Other kinds of graph mining problems, such as computing triangles or k-cliques, can also be naturally implemented as Datalog programs [18]. Our primary motivation for developing distributed RA is as a back-end for an Datalog-like logic-programming language.

2.2 Implementing Parallel Relational Algebra

In our discussion of both TC computation and Datalog generally, we have elided important optimizations and implementation details in favor of formality regarding the main ideas of both. In practice, 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 fusing these operations. In addition, both transitive closure, and Datalog generally, as presented above, are using naïve fixed-point iteration, recomputing all previously discovered edges (i.e., facts) at every iteration. Efficient implementations are *incrementalized* and only consider facts that can be extended to produce so-far undiscovered facts. For example, when computing transitive closure, another relation T_Δ is used which only stores the longest paths in T—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 necessarily yield paths which have been discovered already. This optimization is known as *semi-naïve evaluation* [2]. Each non-static relation (such as T) is effectively partitioned into three relations: T_{full}, T_Δ, and T_{new}. T_{full} stores all facts discovered more than 1 iteration ago; T_Δ stores all facts that were newly discovered in the previous iteration, and is joined with G each iteration to discover new facts; and T_{new} stores all these facts, newly discovered in the current iteration. At the end of each iteration, T_Δ's tuples are added to T_{full}, T_Δ's pointer is swapped with the pointer to T_{new}, and T_{new} is emptied to prepare for the next iteration.

```
1   pfor(auto it = part.begin(); it<part.end();++it){
2     try{for(const auto& env0 : *it) {
3       const Tuple<RamDomain,2> key({{env0[1],0}});
4       auto range = rel_1_edge->equalRange_1(key,
                                  READ_OP_CONTEXT(rel_1_edge_op_ctxt));
5       for(const auto& env1 : range) {
6         if(!(rel_2_path->contains(Tuple<RamDomain,2>({{env0[0],env1[1]}}),
                                   READ_OP_CONTEXT(rel_2_path_op_ctxt)))) {
7           Tuple<RamDomain,2> tuple({{static_cast<RamDomain>(env0[0]),
                                       static_cast<RamDomain>(env1[1])}});
8           rel_4_new_path->insert(tuple,
                                  READ_OP_CONTEXT(rel_4_new_path_op_ctxt));
9         }
10      }
11    }} catch(std::exception &e){SignalHandler::instance()->error(e.what());}
12  }
```

Fig. 3: The join of a TC computation, as implemented by Soufflé.

The state of the art evaluating Datalog is perhaps best embodied in the Soufflé engine [8–10,15]. Soufflé systematically optimizes the RA kernals obtained from an input Datalog program, partially evaluating and staging the resulting RA for the task at hand. Soufflé also performs a strongly-connected-component analysis to extract separate inference tasks connected in a dependency (directed, acyclic) graph—stratifying SCC evaluation. RA itself is performed using a series of nested loops that utilize efficient data-structures to iterate over the tuples of

a relation, iterate over tuples that match a subset of column-values, and insert new tuples. Figure 3 shows a portion of the exact C++ code produced by Soufflé (v1.5.1) for the two-rule TC program shown above (with added indentation only).

To compute $\rho_{0/1}(T_\Delta) \bowtie_1 G$, first the outer relation (the left-hand relation—in this case T_Δ) is partitioned so that Soufflé may process each on a separate thread via OpenMP (line 1 in Figure 3). For each partition, a loop iterates over each tuple in the current partition of T_Δ (line 2) and computes a selection tuple, `key`, representing all tuples in G that match the present tuple from T_Δ in its join-columns (in this case the second column value, `env0[1]`). This selection tuple is then used to produce an iterator selecting only tuples in G whose column-0 value matches the particular tuple `env0`'s column-1 value. Soufflé thus iterates over each $(x, y) \in T_\Delta$ and creates an iterator that selects all corresponding $(y, z) \in G$. Soufflé iterates over all matching tuples in G (line 5), and then constructs a tuple (x, z), produced by pairing the column-0 value of the tuple from T_Δ, `env0[0]`, with the column-1 value of the tuple from G, `env1[1]`, which is inserted into T_{new} (line 8) only if it is not already in T_{full} (line 6).

Given this architecture, Soufflé achieves good performance by using fast thread-safe data-structures, template specialized for common use cases, that represent each relation extensionally—explicitly storing each tuple in the relation, organized to be amenable to fast iteration, selection, and insertion. Soufflé includes a concurrent B-tree implementation [9] and a concurrent blocked prefix-tree implementation [10] as underlying representations for relations.

2.3 Distributed Parallel Relational Algebra

The double-hashing approach, with local hash-based joins and hash-based distribution of relations, is the most commonly used method to distribute join operations over many nodes in a networked cluster computer. This algorithm involves partitioning relations by their join-column values so that they can be efficiently distributed to participating processes [5, 6]. The main insight behind this approach is that for each tuple in the outer relation, all relevant tuples in the inner relation must be hashed to the same MPI process or node, permitting joins to be performed locally on each process.

Recently, radix-hash join and merge-sort join algorithms have been evaluated using this approach [4]. Both these algorithms partition data so that they may be efficiently distributed to participating processes and are designed to minimize inter-process communication. One-sided RMA operations remotely coordinate distributed joins and parallelize communication and computation phases. Experiments for this work scaled join operations to 4,096 nodes, and reached extremely high peak tuples/second throughput, but this work does not address materializing and reorganizing relations for subsequent iterations—challenges required to implement fixed-point algorithms over RA. In addition, this work only considers uniform (perfectly balanced) relations, citing balancing of relations as future work and does not represent realistic workloads because each key has exactly one matching tuple in each relation being joined. A key advantage

of this approach is that radix-hash join and merge-sort join, used on each process, support acceleration via AVX/SIMD instructions and exhibit good cache behavior [3, 11].

Another recent approach proposes adapting the representation of imbalanced relations by using a two-layered distributed hash-table to partition tuples over a fixed set of *buckets*, and, within each bucket, to a dynamic set of *subbuckets* which may vary across buckets [12]. Each tuple is assigned to a bucket based on a hash of its join-column values, but within each bucket tuples are hashed on non-join-column values, assigning them to a local subbucket, then mapped to an MPI process. This permits buckets that have more tuples to be split across multiple processes, but requires some additional communication among subbuckets for any particular bucket. This work presents a static refinement strategy that is used before fixed-point iteration to decide how many subbuckets to allocate per-bucket, and compares two approaches to mapping subbuckets to processes. This implementation does not address dynamic refinement across fixed-point iterations; as relations accumulate new tuples, the difference between the largest subbucket and the smallest subbucket can grow or diminish.

3 Balancing Distributed Relational Algebra

In this section, we extend previous approaches to efficiently distributing relational algebra by developing strategies that mitigate load-imbalance in a fully dynamic manner. First, we describe the architecture of our join operation in detail to ground this discussion. Following [12], we distribute each relation across a fixed number of logical *buckets* (chosen to match the number of MPI processes in our experiments). Each bucket has a variable number of *subbuckets*, that can increase as needed for buckets containing disproportionately large numbers of tuples. Each subbucket belongs to just one bucket and is hosted by a single MPI process, but a single MPI process may host any number of subbuckets.

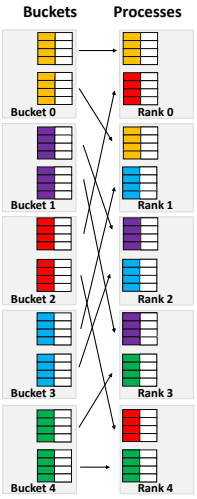


Fig. 4: Round-robin mapping of subbuckets to processes.

To distribute subbuckets to managing processes, we use a round-robin mapping scheme. The example in Figure 4 shows the round-robin mapping of subbuckets to processes where there are 5 buckets with 2 subbuckets each and 5 MPI processes. This process requires a very small amount of added communication, but ensures that no process manages more than one subbucket more than any other.

Locally, subbuckets store tuples using B-trees (an approach used by Soufflé), which carries several advantages over the double-hashing approach's use of hash tables. Crucially, hash-tables can lead to a resizing operation that delays synchronization.

Figure 5 shows a schematic diagram of our join algorithm in the context of an incrementalized TC computation. A join operation can only be performed

for two *co-located relations*: two relations each keyed on their respective join columns that share a bucket decomposition (but not necessarily a subbucket decomposition for each bucket). This ensures that the join operation may be performed separately on each bucket as all matching tuples will share a logical bucket; it does not, however, ensure that all two matching tuples will share the same subbucket as tuples are assgined to subbuckets (within a bucket) based on the values of non-join columns, separately for each relation.

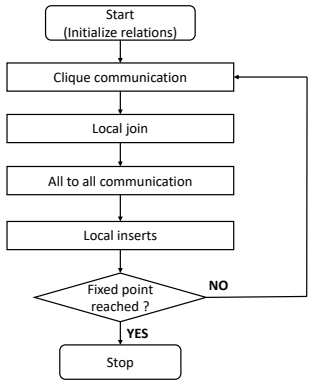
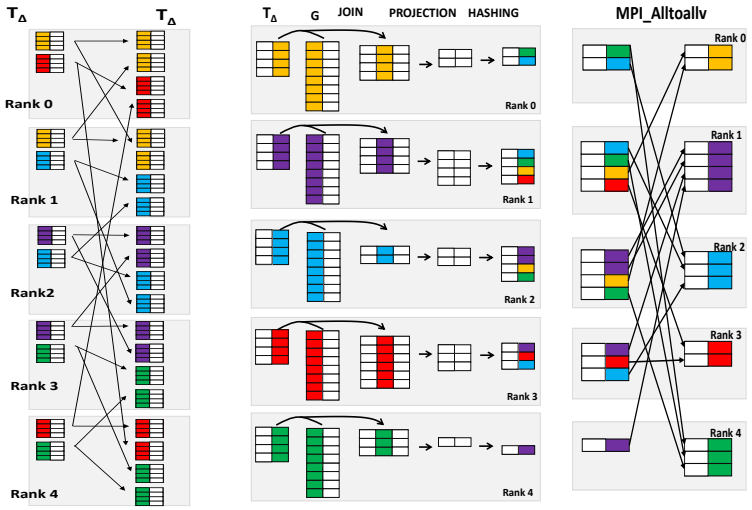


Fig. 5: Major steps in our join algorithm, in the context of TC.

The first step in a join operation is therefore an *intra-bucket communication* phase within each bucket so that every subbucket receives all tuples for the outer relation across all subbuckets (while the inner relation only needs tuples belonging to the local subbucket). Following this, a *local join* operation (with any necessary projection and renaming) can be performed in every subbucket, and, as output tuples may each belong to an arbitrary bucket in the output relation, an MPI *all-to-all* communication phase shuffles the output of all joins to their managing processes (preparing them for any subsequent iteration). Finally, upon receiving these output tuples from the previous join, each process inserts them into the local B-tree for T_{new}, propagates T_{Δ} into T_{full} and T_{new} becomes T_{Δ} for the next iteration along with a empty T_{new}. If no new tuples have been discovered, globally, a fixed point has been reached and iteration may halt.

Intra-bucket communication (shown in Figure 6a) uses MPI point-to-point communication to shuffle all tuples from each subbucket of the outer relation (in the case of T-join-G in TC, T_{Δ}) to all subbuckets of the inner-relation (in the case of TC, G), which will subsequently perform local, per-subbucket joins. It may seem appealing to fuse the final all-to-all communication phase among buckets with the intra-bucket communication of the next iteration, sending new tuples (for T_{Δ} in the next iteration) directly to all subbuckets of G; however, doing this fusion forgoes an opportunity for per-subbucket deduplication and yields meaningful slowdowns in practice.

The local join phase proceeds in a fully parallel and unsynchronized fashion. Each process iterates over its subbuckets, performing a single join operation for each. Our join is implemented as a straightforward tree-based join as shown in Figure 6b. In this diagram, colors are used to indicate the hash value of each tuple as determined by its join-column value. The outer relation's local tuples are iterated over, grouped by key values. For each key value, a lookup is performed to select a portion of the tree storing the inner relation's local tuples where all tuples have a matching key value (in this case on the first column of G). For two sets of tuples with matching join-column values, we effectively perform a Cartesian product computation, producing one tuple for all output pairs. Each output



(a) (b) (c)

Fig. 6: (a) Intra-bucket communication; each subbucket of T_Δ sends its data to all subbuckets of G. (b) Local, per-subbucket joins (including projection and re-hashing). (c) All to all communication.

tuple has any needed projection and renaming of columns performed on-the-fly; in this case, the prior join columns that matched are projected away. These output tuples are temporarily stored in a tree, to perform local deduplication, and are then staged for transmission to new managing subbuckets in their receiving relation. After the join, each output tuple belongs to T_{new} (T_Δ in the next iteration) and must be hashed on the final column to determine which bucket it belongs to, and on all other columns to determine which subbucket within that bucket. While we follow Soufflé in implementing B-tree-based joins on each process, other approaches may be able to take better advantage of AVX instructions and on-chip caching [3,11]. We plan to investigate alternatives in the future and believe them to be largely orthogonal to our paradigm for decomposition, communication, and balancing of relations.

Next, an all-to-all communication (shown in Figure 6c) phase transmits materialized joins to their new bucket-subbucket decomposition in T_{new}. After being hashed on their new join column value to assign each to a bucket, and on all non-join-column values to assign each to a subbucket, the managing process for this subbucket is looked up in a local map and tuples are organized into buffers for MPI's `All_to_allv` synchronous communication operation. When this is invoked, all tuples are shuffled to their destination processes.

Finally, after the synchronous communication phase, T_Δ is locally propagated into T_{full}, which stores all tuples discovered more than 1 iteration ago. New tuples are checked against this T_{full} to ensure they are genuinely new facts (paths in G), and are inserted into a B-tree for T_{new} on each receiving process to perform remote deduplication. At this point, the iteration ends, T_{new} becomes T_Δ for

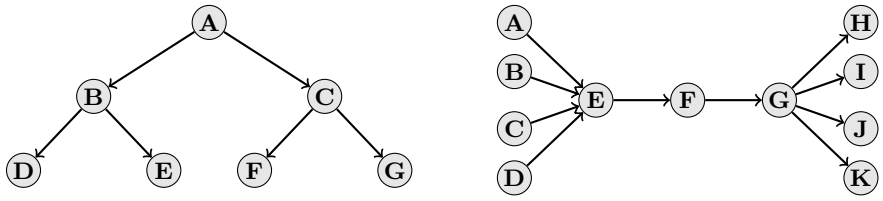


Fig. 7: A complete binary tree with height 2 and down-directed edges (left); a bowtie graph with width 4 and length 3 (right).

the subsequent iteration, and an empty T_{new} is allocated. If no new tuples were actually discovered in the previous iteration, a fixed-point has been reached and no further iterations are needed as the database as stabilized.

3.1 Two Kinds of Load-imbalance

We consider two kinds of load-imbalance and how they might occur and change across iterations of a transitive closure computation: *spacial load imbalance*, when a relation's stored tuples are mapped unevenly to processes, and *temporal load imbalance*, when the number of output tuples produced varies across iterations.

Consider the class of relations that encode complete binary trees of height H where directed edges face either strictly downward or upward. The left side of Figure 7 shows an example of a downward-facing complete binary tree with height 2. If a downward-facing relation in this class is keyed on its first column, there is no load imbalance as each key has exactly two tuples (two children per parent); if it is keyed on its second column, there is likewise no load imbalance as each key has exactly one tuple (one parent per child). If we use an up-directed binary tree instead, these are reversed; either way, the relation is initially balanced. Now what happens when we compute its TC?

Direction	T	G
Up	$O(2^{H-D})$	$O(1)$
Down	$O(D)$	$O(1)$

Fig. 8: Worst-case imbalance for T and G in TC computation (for complete binary tree topology).

The TC of a down-directed complete binary tree of height H (keyed on column 0) has significant spacial imbalance. The root node has $O(2^H)$ tuples (edges) hosted on its process, while nodes at depth $H-1$ have only 2. If the relation is keyed on the second column (or if we use an up-directed tree), then there is a natural imbalance that increases linearly with depth. In a TC computation, as relation T is keyed on its second column, not the first, a down-directed tree exhibits the more moderate imbalance; for an upward-facing complete binary tree, T has a worst-case exponential imbalance ratio. The worst-case imbalance ratios for T and G are summarized in Figure 8.

The complete binary tree topology graphs are perhaps corner cases for relation imbalance, however such relations can occur in the wild, and even more moderate degrees of imbalance can cause relational algebra to slow down or crash in practice. Relational algebra that is suitable for arbitrary workloads

must handle arbitrary degrees of spacial imbalance gracefully, and if used within a fixed-point loop (as is the case for general logical inference applications), relations must support dynamic spacial refinement that is efficient enough to handle arbitrary changes in imbalance across time—both increases and decreases.

Now consider the bowtie topology shown on the right side of Figure 7. Each bowtie-topology graph has a width W and length L, and is formed by connecting W nodes each to the starting node of a string of L nodes, connected on the far side to another W nodes each. What happens when computing the TC of an arbitrary bowtie relation? The first iteration, a join between a bowtie relation and itself, yields $2W + L - 1$ new edges; in fact, at every iteration until the last, the worst-case join output is in $O(W + L)$. At the final iteration, however, the number of output tuples suddenly becomes quadratic in the width of the bowtie, $O(W^2)$, as each of the leftmost nodes are paired with each of the rightmost nodes. This illustrates a case of temporal imbalance—a large bowtie can produce fewer than $100K$ tuples one iteration and more than $1B$ tuples the next.

A general-purpose system for relational algebra should also be robust to unexpected surges in the per-iteration workload, adapting itself to dynamic changes in the overall workload across time. While bowtie graphs represent corner cases, it is common to see join output change significantly from iteration to iteration when computing TC of real-world graphs as well (see Table 1).

3.2 Three Techniques for Adaptive Load-balancing

Now we describe three techniques that, used in conjunction, can remediate both kinds of imbalance illustrated in the previous section: bucket refinement, bucket consolidation, and iteration roll-over. *Bucket refinement* is a dynamic check for each bucket to see if its subbuckets are significantly heavier than average, triggering a refinement in which new subbuckets are allocated to support this larger number of tuples. *Bucket consolidation* occurs only if there are a significant number of refined buckets, and consolidates buckets into fewer subbuckets when spacial imbalance has lessened. Finally, *iteration roll-over* allows particularly busy iterations to be interrupted part-way, with completed work being processed immediately and with the residual workload from the iteration "rolling over".

Bucket refinement is one of two techniques we use to address natural spacial imbalance among the keys of a relation. Refinement is used to check for disproportionately *heavy* subbuckets (those with more than the average number of tuples), and to spread this load across an increased number of subbuckets. Checking for needed refinement is a lightweight, but non-trivial step, so we only perform this imbalance check every N iterations (where N is an adjustable parameter). In our experiments, we use both $N = 2$ and $N = 10$ but observed only small a difference in performance. To check for refinement, the heaviest subbucket in each bucket is compared with the average subbucket size across all buckets; when the ratio is greater than 3-to-1, we refine this bucket, quadrupling its subbucket count from 1 to 4, from 4 to 16, from 16 to 64, etc; the subbucket count in each bucket is always maintained as a power of 4. This additional allocation of subbuckets extends the round-robin mapping maintained in lock-step

on all processes by transmitting a small amout of meta-data during the global all-to-all phase. An immediate point-to-point communication is triggered especially to distribute three-quarters of the tuples from each subbucket in a refined bucket to processes hosting newly allocated subbuckets.

Bucket consolidation is a complementary technique for combining previously split subbuckets when spacial load imbalance has again lessened. The imbalance check for bucket consolidation is guarded by a global check to see if greater than 60% of buckets have been refined to 4 or more subbuckets. When this is the case, all buckets containing subbuckets with a below-average tuple-count are consolidated into $\frac{1}{4}$ as many subbuckets. This process uses the same communication machinery as bucket refinement; a special point-to-point communication is used to redistribute tuples into a smaller number of buckets, all of which are freshly allocated using our round-robin allocation scheme to prevent pathological cases.

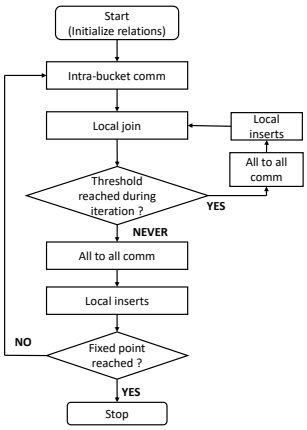


Fig. 9: The major steps in our join algorithm with iteration roll-over added.

Iteration roll-over guards against severe cases of temporal imbalance which can slow evaluation, through thrashing of memory, or crash a process. As in the case of our bowtie topology, the shape of a graph can cause a sudden explosion of work in a single iteration. This requires our algorithm to be on-guard for temporal imbalance at every iteration, as opposed to spacial imbalance where we may avoid some overhead by checking for imbalance intermittently. As each local join is processed, grouped by key-column values, a count of output tuples is maintained and at each new key-column value we check to see if it has passed some fixed *threshold value* (a tunable parameter—we experiment with several threshold values). When the threshold has been exceeded, we stop computing the join and transmit the partial join output to destination processes for remote deduplication early.

This step is shown in Figure 9. When a threshold value is reached during the local-join phase, an all-to-all communication is triggered, followed by local inserts in each destination subbucket. Then, instead of the iteration ending (with propagation from each R_Δ to R_{full} and from each R_{new} to R_Δ), the previous iteration continues exactly where it left off. We may also think of this an an *inner iteration* as opposed to the normal *outer iterations* of semi-naïve evaluation. Each inner iteration batches the threshold value in output tuples to promote cache coherence and prevent overflow.

4 Evaluation

We begin by studying the impact of spatial and temporal load balancing in isolation. Following this, we analyze the impact of both forms of load balancing, jointly, on real-world graphs and at scale.

We performed our experiments for this work on the Theta Supercomputer [1] at the Argonne Leadership Computing Facility (ALCF). Theta is a Cray machine with a peak performance of 11.69 petaflops, 281,088 compute cores, 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM and 10 PiB of online disk storage. We performed our experiments using the SuiteSparse Matrix Collection [7].

4.1 Spatial load balancing

We evaluate the performance of spatial load-balancing, in Figure 10, by computing the transitive closure of eight balanced binary-tree graphs (depths: 21, 23, 25, 27, for each edge direction: *up* and *down*). We run all these experiments at 512 cores, both with and without spatial load balancing turned on. The transitive closure for the graphs (in order of increasing tree depth) generated 39,845,890, 176,160,770, 771,751,938 and 3,355,443,202 edges, respectively (taking 21, 23, 25 and 27 iterations to converge). Note that both up-directed (`UP`) and down-directed (`DOWN`) graphs (of equal depth) produce the same number of edges.

We observed dynamic load balancing lead to a roughly 2× improvement for `UP` graphs. As an example, for the graph with depth 27, load balancing led the total runtime to go down from 463 seconds to 209 seconds. In our experiments, we set the load-imbalance check to be performed every other iteration, as this is the sole feature under consideration; for all four graphs, however, actual re-balancing (refinement of buckets) occurred only five times each, with the cumulative number of sub-buckets increasing dynamically from 512 to 1088 in every case.

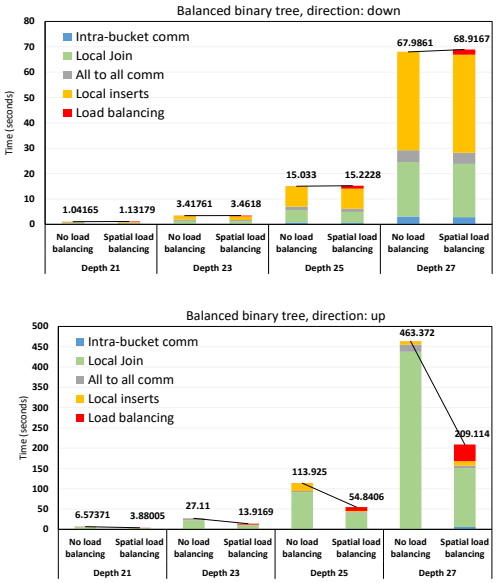


Fig. 10: TC computation for complete binary trees (depths 21, 23, 25 and 27) for up (top) and down (bottom) pointing edges with and without load balancing.

On the other hand, load balancing does not yield any improvement for `DOWN` graphs. This is despite the fact that computing the TC `UP` and `DOWN` graphs produces the same number of edges and takes the same number of iterations to converge in both cases. What differs is how tuples are distributed among keys (values for the join column); with linear imbalance in the `DOWN` case and exponential imbalance in the `UP` case. We note that TC for `UP` graphs can be computed as efficiently as `DOWN` graphs if we change our iterated join from T-join-G to G-join-T, but this optimization requires a priori knowledge of the final graph's topology, which

is likely unavailable. Our approach aims to be as relation agnostic as is possible, so that arbitrary logical inference tasks may be scaled effectively.

It may be surprising that `DOWN` graphs do not show some lesser need for dynamic re-balancing as they evolve from being perfectly balanced to being linearly imbalanced. This would be the case if each key were mapped to a unique bucket.Since keys are hashed to a smaller number of buckets, however, we only observe a 1.001 imbalance ratio for height-25 `DOWN` trees and we observe a 204.8 inter-bucket imbalance ratio for height-25 `UP` trees.This means hashing keys to buckets has a modest ameliorating effect on imbalance that can be sufficient, but not in cases of severe imbalance.

4.2 Temporal load balancing

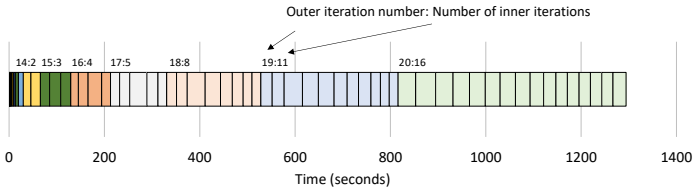


Fig. 11: Breakdown of time taken to finish 20 iterations (paths of length 20) using temporal load balancing.

Temporal load balancing is a key safety feature, without which it can become impossible to make meaningful progress due to continuous page faults. We demonstrate this particular use case for an extreme scenario, where thresholding acts as a critical component. We use a very large graph `Hardesty3` [7] (40,451,631 edges) that generates an overwhelming number of edges at an accelerating pace. Without thresholding, a process gets overwhelmed by the computational workload and runs out of memory. We applied a modified version of the transitive closure problem where, instead of trying to reach the fixed point, we restricted our computation to run only 20 iterations. (At the end of iteration 20, we have computed all paths of up to length 20.) We ran our experiments at 32,768 cores, both with and without temporal load balancing. Without load balancing, we were only able to complete iteration 16, whereas with load balancing we were able to finish all 20 iterations. The number of edges generated at the end of 20 iterations was 913,419,562,086 (13.3 Terabytes). We have plotted a breakdown of time taken during every iteration in Figure 11. We observed temporal load balancing was used for all iterations after the 14th iteration, the 19$^{\text{th}}$ and 20$^{\text{th}}$

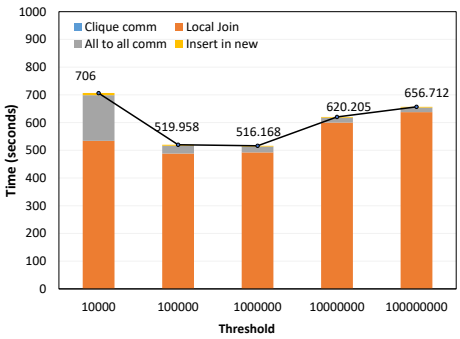


Fig. 12: Time to compute TC for bow-tie topology graph with varying thresholds.

iteration for example was broken into 11 and 16 inner iterations respectively. Also, it can be seen that the aggregate time taken increases significantly with every iteration. For this experiments, we used a threshold of 8,000,000. It took 1,256 seconds to finish running 20 iterations.

Temporal balancing can also be used as an optimization technique for extreme topology graphs such as the bowtie (see Figure 7). To demonstrate this, we used a bow-tie graph with a width of 10,000 vertices and length of 10 vertices. This graph generates $10{,}000 \times 10{,}000$ edges in the 10^{th} iteration, when all vertices on the left side of the bowtie each discover paths to all vertices on the right side of the bowtie. For the first 10 iterations, the number of edges produced every iteration is roughly 20,000 whereas the number of edges generated in the 10^{th} iteration is 100,000,000, leading to a temporal imbalance ratio of about 5,000. We run our experiments at 256 cores with 5 different threshold values: 10,000, 100,000, 1,000,000, 10,000,000, and 100,000,000. The transitive closure of the graph generated 400,440,055 edges. While the number of outer iterations is 10 for all thresholds, the number of inner iterations varied as 20,020, 3,343, 402, 49 and 13. Note that small threshold values lead to an unnecessarily increased number of inner iterations and hence an increased number of all to all communication epochs. Smaller threshold values also lead to more-optimized local join phases, as the nested B-tree data structures holding relations do not grow very large, leading to better cache coherence while performing lookups and inserts.

We plot our results in Figure 12. We observe a best timing of 516 seconds for a threshold of 1,000,000 tuples. At this threshold, we achieve a good balance between the extra time taken for all-to-all communication phases versus the time saved during each local join phase. Lower thresholds make the problem bounded by communication (all-to-all phase) whereas higher thresholds make the problem bounded by computation (local join phase). At larger process counts, we observed better performance for larger threshold values. For example, at 8,192 cores the transitive closure of graph `sgpf5y6` with edge count 831,976 took 384, 559 and 590 seconds for threshold values 100,000,000, 10,000,000 and 1,000,000 resp. We use temporal load balancing primarily as a safety check, although it also a practical optimization for corner-case topology graphs. We believe that our design is flexible enough to be tuned to different scales and different degrees in imbalance in the input graph.

4.3 Transitive closure at scale

We also performed experiments to study the impact of load balancing on real-world and random graphs. We compute the transitive closure of six real world graphs [7] and two random graphs generated via RMAT [13]. All our experiments were performed at $8,192$ processes with both temporal and spatial load-balancing enabled. In these experiments we check for spacial imbalance every tenth iteration and temporal imbalance at every iteration—the roll-over threshold is set at 8,000,000 tuples. Our results are shown in Table 1. All graphs except `TSC_OPF_300` make use of spatial load balancing. We also note that graphs

`sgpf5y6`, `RMAT_1`, and `RMAT_2` make use of temporal load balancing, as the number of edges generated for these graphs grow at a rapidly increasing rate (resp., 76, 2, and 9 billion edges in the first 20 iterations).

Name	Edges	Time (seconds)	Spatial balancing	Temporal balancing	Iterations	TC Edges
lhr34	764,014	64.3391	✓		30	1,233,554,044
nemeth13	241,989	28.8445	✓		310	45,186,771
sgpf5y6	831,976	578.641	✓	✓	20	76,382,533,943
rim	1,014,951	46.7834	✓		30	508,931,041
TSC_OPF_300	415,288	2.11			30	1,876,367
RMAT_1	200000	68.8143	✓	✓	20	2,502,341,599
RMAT_2	400000	220.993	✓	✓	20	9,481,998,719

Table 1: List of eight (6 real world + 2 random) graphs used in our evaluation.

We also performed strong scaling studies for the graphs in table 1, we report the performance numbers for four graphs `lhr34`, `sgpf5y6`, `TSC_OPF_300`, and `rim` in Figure 13. For graphs `lhr34` and `rim` we observe 7× improvement in performance while going from 512 processes to 8192 processes (16×). For graph `TSC_OPF_300` the trend is reversed, this is because the graph under consideration is sparsely connected (as seen from the small TC size), requires very few iterations to converge and hence, is not suitable for a large process run.

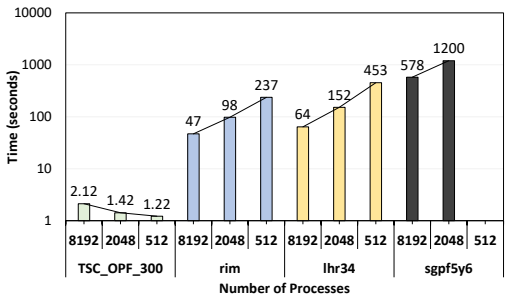


Fig. 13: Strong-scaling plots for `lhr34`, `sgpf5y6`, `TSC_OPF_300`, and `rim` graphs

5 Conclusion

In this paper, we have explored the issue of inherent imbalance in relations, and across iterations of fixpoint computations. We have described three techniques for mitigating these issues in parallel relational algebra, distributed in a data-parallel manner across many cores, and have evaluated our approach by computing the transitive closures of real world, random, and corner-case graphs.

References

1. Theta alcf home page. https://www.alcf.anl.gov/theta
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases: the logical level. Addison-Wesley Longman Publishing Co., Inc. (1995)
3. Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: Sort vs. hash revisited. Proceedings of the VLDB Endowment **7**(1), 85–96 (2013)

4. Barthels, C., Müller, I., Schneider, T., Alonso, G., Hoefler, T.: Distributed join algorithms on thousands of cores. Proc. VLDB Endow. **10**(5), 517–528 (Jan 2017)
5. Cacace, F., Ceri, S., Houstma, M.A.W.: An overview of parallel strategies for transitive closure on algebraic machines. In: Proceedings of the PRISMA Workshop on Parallel Database Systems. pp. 44–62. Springer-Verlag New York, Inc., New York, NY, USA (1991)
6. Cheiney, J.P., de Maindreville, C.: A parallel strategy for transitive closure using double hash-based clustering. In: Proceedings of the Sixteenth International Conference on Very Large Databases. pp. 347–358. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1990)
7. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (Dec 2011)
8. Jordan, H., Scholz, B., Subotić, P.: Soufflé: On synthesis of program analyzers. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 422–430. Springer International Publishing, Cham (2016)
9. Jordan, H., Subotić, P., Zhao, D., Scholz, B.: A specialized b-tree for concurrent datalog evaluation. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. pp. 327–339. PPoPP '19, ACM, New York, NY, USA
10. Jordan, H., Subotić, P., Zhao, D., Scholz, B.: Brie: A specialized trie for concurrent datalog. In: Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores. pp. 31–40. PMAM'19, ACM, New York, NY, USA (2019)
11. Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core cpus. Proceedings of the VLDB Endowment **2**(2), 1378–1389 (2009)
12. Kumar, S., Gilray, T.: Distributed relational algebra at scale. In: International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE (2019)
13. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C.: Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In: European conference on principles of data mining and knowledge discovery. pp. 133–145. Springer (2005)
14. Liu, M., Dobbie, G., Ling, T.W.: A logical foundation for deductive object-oriented databases. ACM Transactions on Database Systems (TODS) **27**(1), 117–151 (2002)
15. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in datalog. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 196–206. CC 2016, ACM, New York, NY, USA (2016)
16. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 632–647. Springer (2007)
17. Van Gael, J.A.F.M., Herbrich, R., Graepel, T.: Machine learning using relational databases (Jan 29 2013), uS Patent 8,364,612
18. Wang, K., Zuo, Z., Thorpe, J., Nguyen, T.Q., Xu, G.H.: Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). pp. 763–782 (2018)
19. Zinn, D., Wu, H., Wang, J., Aref, M., Yalamanchili, S.: General-purpose join algorithms for large graph triangle listing on heterogeneous systems. In: Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit. pp. 12–21. GPGPU '16, ACM, New York, NY, USA (2016)