Parallel Relational Algebra for Logical Inference at Scale

Sidharth Kumar, Thomas Gilray
University of Alabama at Birmingham
https://sidharthkumar.io — sid14@uab.edu
https://thomas.gilray.org/ — gilray@uab.edu

Abstract: Relational algebra (RA) comprises an important basis of operations, conspicuously sparse in the HPC literature. It can be used to implement a variety of algorithms in satisfiability and constraint solving [4], graph analytics [6], program analysis and verification [3], deductive databases [2], and machine learning [5]. 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. Declarative logic programming offers the promise of unifying specification and implementation, permitting programmers to focus on writing correct and maintainable code, and permitting its operational evaluation to be synthesized automatically. Despite its expressive power, relational algebra has not received the same attention in high-performance-computing research as more common primitives like stencil computations, floating-point operations, numerical integration, and sparse linear algebra. Furthermore, specific challenges in permitting fixed-point iteration, in addressing representation and communication among distributed portions of a relation, and in balancing inherently unbalanced relations, have previously thwarted successful scaling of relational algebra applications to HPC platforms. We are developing a set of efficient algorithms to effectively parallelize and scale key relational algebra primitives. We aim to develop foundational theory, practical implementations, and rigorous evaluations of our approach on three important application domains with the ultimate goal of enabling massively parallel distributed reasoning on cluster computers directed by expressive, ergonomic, rule-based languages.

Applications: To evaluate the scalability of our parallel RA infrastructure, we focus on three applications: graph mining **(E1)**, static program analysis **(E2)**, and deductive databases for physical simulation data **(E3)**. Figure 1 shows the overall pipeline we propose; the top shows our three experimental applications, sitting atop a platform for logical inference, implemented with relational algebra that the techniques under investigation **(T1-T3)** supports.

Typical graph computational algorithms are not suited for mining tasks that aim to discover complex structural patterns of a graph. Extracting such features such as paths, cliques, frequent subgraphs, etc, is straightforward to implement using relational algebra. We focus on two fundamental graph-mining tasks **(E1)**, transitive closure and clique computation. These applications are the most immediate uses of relational algebra; transitive closure is simply an iterated sequence of relational *join*, *projection*, and *union*, until a fixed point is reached. Second, we focus on static program analysis **(E2)**, a vital logical inference problem that exemplifies the power of our approach. Static analysis brings in substantial task- and data-parallelism, as well as evolution in balancing requirements across time. Finally, we focus on deductive database applications and the use of relational algebra as a combined storage and-inference system **(E3)**. Of particular interest to us is the possibility that logical inference can benefit traditional HPC problems for feature extraction in an automated way.

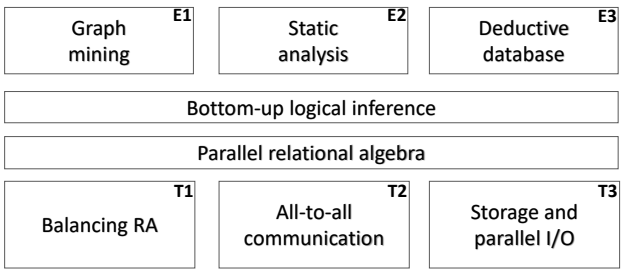

Figure 1: An overview of our pipeline: our application-foci are implemented as bottom-up logical inference, implemented as task- and data-parallel relational algebra, built using the specific techniques we investigate in our proposed work.

Techniques: One of the central challenges in parallel relational algebra is how to balance the various facets of the problem: balancing communication and computation, balancing across iterations of an inference task, balancing across relations, and balancing among task-level components. We have built on previous approaches by developing strategies that mitigate load-imbalance in a dynamic manner. Our own approach [1] uses 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. Each tuple is assigned to a bucket based on a hash of its key-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. The first step in a join operation is an *intra-bucket communication* (Figure 2a) 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) is performed in every subbucket (Figure 2b), and, as output tuples may each belong to an arbitrary bucket in the output relation, an MPI *all-to-all* communication phase (Figure 2c) shuffles the output of all joins to their managing processes (preparing them for any subsequent iteration).

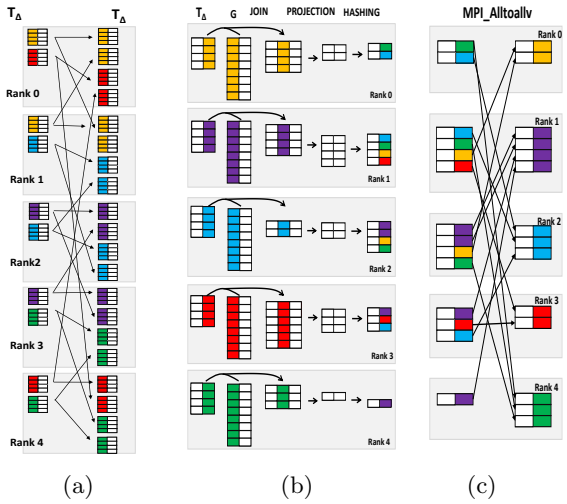

Figure 2: (a) Intra-bucket communication; each subbucket of T_Δ sends its data to all subbuckets of G. (b) Local, per-subbucket joins (including projection and re-hashing). (c) All to all communication.

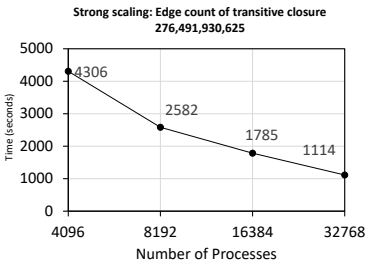

Figure 3: Strong scaling for TC computation of SuiteSparse graph `mc2depi`.

Results: The transitive closure (T) of an input graph (G) is iteratively extended by adding new paths discovered by a join operation until a fixed point is reached, and no new paths can be added to T. We performed strong scaling analysis for a graph with edge count 2,100,225 (mc2depi), varying the number of processes from 4,096 to 32,768. Figure 3 shows a preliminary strong-scaling study. We observed decent scaling to about 16k cores, producing a graph of over 276 billion paths. To the best of our knowledge, this is the largest such feature extraction now described in the literature. Given that this approach does not use any load balancing and does nothing to optimize a synchronous use of MPI's `All_to_allv` primitive, this scaling plot is a promising proof-of-concept that iterated joins on graphs derived from real applications will be scalable to 10k+ processes.

Conclusion and research direction: Effective declarative programming represents a long-standing dream of computing—exchanging code describing *how* to compute for code simply describing *what* to compute. Instead of requiring programmers to themselves balance the vying concerns of correctness, maintainability, and scalability in each task, declarative programming languages permit users to focus on the first two concerns, writing high-level specifications of *what* should be computed, while allowing the underlying implementation (i.e.,*how* the operational mechanics of the program work) to be extracted automatically. With this research we are making inroads by developing techniques for parallelizing relational algebra as a platform for declarative logical inference tasks. Our research is motivated by applications from the domain of graph analysis (feature extraction), static program analysis (reverse engineering, component verification, exploit generation, etc), and deductive databases (statistical and topological features).

References

[1] Sidharth Kumar and Thomas Gilray. Distributed relational algebra. In *International Conference on High Performance Computing, Data, and Analytics (In Submission)*, 2019.

[2] Mengchi Liu, Gillian Dobbie, and Tok Wang Ling. A logical foundation for deductive object-oriented databases. *ACM Transactions on Database Systems (TODS)*, 27(1):117–151, 2002.

[3] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 196–206, New York, NY, USA, 2016. ACM.

[4] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[5] Jurgen Anne Francios Marie Van Gael, Ralf Herbrich, and Thore Graepel. Machine learning using relational databases, January 29 2013. US Patent 8,364,612.

[6] Daniel Zinn, Haicheng Wu, Jin Wang, Molham Aref, and Sudhakar Yalamanchili. 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*, GPGPU '16, pages 12–21, New York, NY, USA, 2016. ACM.