Scalable Relational Algebra

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. Scalable Relational Algebra. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 2 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 datalog solvers.

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

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

... talk about names as indices?

2.1 Standard RA operations

...

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

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

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

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

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

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

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

selection on multiple columns can be accomplished by repeated selection on a single column at a time. Selecting just those tuples from relation R where column α_i matches value v is performed with operator $\sigma_{\alpha_i=v}(R)$ that is defined:
$\sigma_{\alpha_i=v}(R) = \{(r_{\alpha_0}, \ldots, r_{\alpha_k}) \mid (r_{\alpha_0}, \ldots, r_{\alpha_i}, \ldots, r_{\alpha_k}) \in R \wedge r_{\alpha_i} = v\}$.

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	address
samp	1554291414	162.103.150.12
karenk	1554181337	171.31.15.120
karenk	1554219962	155.28.11.102
karenk	1554133720	171.31.15.120

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

`users` ⋈ `logins`

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

2.2 Transitive closure

One of the simplest common algorithms that may be implemented as a loop over fast 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.

2.3 Datalog

Transitive closure is a simple example of deduction. At each...

3 IMPLEMENTATION

Lorem ipsum dolar sit amat

4 EVALUATION

Lorem ipsum dolar sit amat

5 RELATED WORK

Lorem ipsum dolar sit amat

6 CONCLUSION

Lorem ipsum dolar sit amat