Relational Algebra: a Brief Introduction

Kyle R. Wenholz

University of Puget Sound

April 17, 2012

Contents

1 Introduction

To the data-hungry, the world is composed of infinitely many data points waiting to be observed. In order to make sense and use of this data computer scientists find it helpful to devise mathematical structures for describing its storage and manipulation. This paper will examine the most popular of these: the relational algebra first proposed by E.F. Codd in 1970 [2]. While it is not necessary to have a background in Abstract Algebra, familiarity and exposure to various algebras and their structures only makes relational algebra more sensible. Relational algebra is regarded as an algebra in large part because of its roots in first-order logic and set theory.

We will begin our discussion with a motivating example in Section 1.1 to appreciate the importance of approaching data rigorously. Section 2 will discuss the objects, operations, general terms, and basic results used in relational algebra. Our discussion concludes in Section 3 with a discussion of the importance this algebra plays in query optimization.

1.1 Motivation

Suppose we are presented with the task of organizing a vast amount of data. There are many ways to accomplish this task and many reasons for choosing each. Our solution may be hardware dependent to improve speed and efficiency in the machine, or we may opt for a model easily understood by humans but complex and difficult to implement in a computer. Before E.F. Codd's relational algebra, many solutions were proposed (and some are still in use for various reasons) including tree structures much like a file system and network based models behaving as relationships between entities [1]. The major advancements of Codd's day came with the description of fast set-theoretic-like data structures.

Consider the task of organizing data about honey bees. We know the names, date of birth, and hive "address" for each bee. Each hive has an address and queen's name. This information is organized in Figure 1 for future reference. We need to store potentially massive amounts of this data, be able to ask questions about the data and compute answers, and have a way of representing our system to experts and non-experts alike. With programs like *Microsoft Excel*, it may be apparent how we might organize all of the data. We now examine how relational algebra deals with these tasks for general data, not just bees.

2 The Relational Algebra

2.1 Elements

The element of relational algebra is, unsurprisingly, a relation. A **relation** is a subset of the Cartesian product of some number of sets [3](Definition 1). Note this implies each element is uniquely identified by its specific contents.

Definition 1. ***Cartesian Product****: Given sets $D_1, D_2, \ldots, D_n$, we denote*

$$D_1 \times D_2 \times \cdots \times D_n = \{(d_1, d_2, \ldots, d_n) | d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n\}$$

as the Cartesian product ($\times$) of the sets $D_1, D_2, \ldots, D_n$. The elements of this set are referred to as $n-tuples$, or just tuples. As in the literature, we will use the terms Cartesian product and cross product synonymously.

It is convenient to think of relations as spreadsheets, where each column represents values from a set and each row itself a single element of the cross product. This makes the information easy to visualize and explain. Columns are called **attributes** and rows are called **tuples**. We denote a relation R with attributes $\{A, B, C\}$ as $R(A, B, C)$, where A, B, C are sets. The list of attributes is called a **schema** and generally order does matter. Relation 1 uses our bee example to make a relation for bees. We may think of these relations recursively, with tuples as a "single entry" relation. This concept will help as we begin to explore the various operations of the relational algebra.

Name	DateOfBirth	HiveAddress
Joe	10/20/1901	222 Smokey Street
Sally	3/4/2001	321 Pooh Circle
Frank	10/20/1901	321 Pooh Circle

Relation 1: : An example of $Bee(Name, DateOfBirth, HiveAddress)$.

Bees	Hive
Name	Address
Date of Birth	Queen Name
Hive Address	

Figure 1: The basic bee information

2.2 Operations

Relational algebra boils down to six basic operations (depending on who you ask)[3][4]. Several of these are closely related to operations in set theory, but relations are not sets and must be treated with special care. Still, we will see a close relationship with set theory throughout our work. All of the following operations were developed in Codd's original paper [3], but we use more modern notation [5] to benefit from years of refinement.

The Cartesian product, defined in Definition 1, is our first operation. This operation joins all possible pairs of tuples between two relations. The **degree** (number of elements in a tuple) of the resultant relation is different from that of strict set theory, as the next example shows. For relations $R(A, B, C)$ and $S(A, E, F)$ we have $R \times S = T(R.A, B, C, S.A, E, F)$. Note we used two identically named attributes in our parent relations and these were denoted with $Parent.Name$ in the new relation. Sometimes this is considered part of a separate operation done before the Cartesian product called **rename**. Rename is necessary for many operations in databases, but we will use it only implicitly throughout this paper.

Set union ($\cup$), intersection ($\cap$), and difference ($-$) all behave in the usual set theoretic manner with the qualification that participating relations have identical schemas.

Definition 2. ***Selection****: Given $R(i_1, i_2, \ldots, i_n)$, we write a selection as*

$$\sigma_{i_k \theta v}(R)$$

where i_k is an attribute name, v is a value constant, and θ is a binary operation in the set $\{<, \leq, =, \geq, >\}$. The result is a relation, whose elements are a subset of R, satisfying the predicate formed by $i_k \theta v$. We may extend this to any number of predicates simply listed as $\sigma_{a_1 \theta v_1, a_2 \theta v_2, \ldots, a_l \theta v_l}$.

We may think of selection as a nice way to pick and choose specific tuples from any relation, whether this be a single tuple or several. Relation 2 shows the result of a selection performed on Figure 1.

Definition 3. ***Projection****: Given a relation $R(i_1, i_2 \ldots, i_n)$ and a set $\{j_1, j_2, \ldots, j_m\}$, where each j_k corresponds to an attribute i_l in the relation and $m \leq n$, we write the projection as*

$$\Pi_{j_1, j_2, \ldots, j_m}(R) = S(j_i, j_2, \ldots, j_m)$$

S is a new relation containing all tuples of R restricted to the attributes $j_1, j_2, \ldots, j_m$.

Name	DateOfBirth	HiveAddress
Joe	10/20/1901	222 Smokey Street
Sally	3/4/2001	321 Pooh Circle
Frank	10/20/1901	321 Pooh Circle

Relation 1=A

Name	DateOfBirth	HiveAddress
Sally	3/4/2001	321 Pooh Circle
Frank	10/20/1901	321 Pooh Circle

Relation 2: $\sigma_{HiveAddress='321PoohCircle'}(A)$.

It is easiest to think of a projection as simply column selection in the relation. In fact, this really is an equivalent definition, albeit less rigorous. Look to Relation 3 for the result of a projection.

Name	DateOfBirth	HiveAddress
Joe	10/20/1901	222 Smokey Street
Sally	3/4/2001	321 Pooh Circle
Frank	10/20/1901	321 Pooh Circle

Relation 1: A

Name	DateOfBirth
Joe	10/20/1901
Sally	3/4/2001
Frank	10/20/1901

Relation 3: $\Pi_{Name,DateOfBirth}(A)$.

We might now wonder how to combine these operations. The answer is quite straightforward: operations in the relational algebra are composed just like functions (even with the binary operators union, intersection, and difference). For example, we might take

$$\sigma_{DateOfBirth=10/20/1901}(\Pi_{Name,DateOfBirth}(Relation1))$$

This is equivalent to asking for the name and date of birth of any bee born on 10/20/1901. See Relation 4 for the result. These and even more complicated questions are now askable and answerable through the relational algebra.

Name	DateOfBirth
Joe	10/20/1901
Frank	10/20/1901

Relation 4: : An example of $\sigma_{DateOfBirth=10/20/1901}(\Pi_{Name,DateOfBirth}(Relation1))$.

2.3 Derived Operations

Now that we have a grasp on the fundamental operations and elements of relational algebra, let us examine more powerful questions. The goal of organizing data as we have done so far is to create an easy and understandable means for answering questions about this data. We call these questions **queries**. The following are several common types of queries which are sometimes treated as their own operations, but they are all derived from the basic operations stated earlier.

2.3.1 Join

Suppose we are given the relation $Bee(Name, DateOfBirth)$. How might we go about determining which bees were born on the same day? Assume we created a copy of our table as $Bee2$. Our query might look like

$$\sigma_{Bee.DateOfBirth=Bee2.DateOfBirth, Bee.Name \neq Bee2.Name}(Bee \times Bee2)$$

Our resulting relation would contain pairs of bees born on the same day, with their names and each one's date of birth. We could, of course, remove the date of birth with a projection, but this is enough for now. We have just made a **join** of two tables (denoted $R \bowtie S$ for relations R and S). A join is an operation in which we take the cross product of two tables and then select an attribute on which to "match" tuples. This is a very natural and common operation. It is so common that at least four other variants of join have been defined and are commonly used [5].

2.3.2 Division

Some queries may involve statements like the phrase "for all". Suppose our bees from earlier have many hives and we wish to know the bees registered with every residence (i.e. hive). This sort of operation is normally referred to as **division** [5].

Definition 4. ***Division****:let r and s be two schemas such that $s \subset r$. Let $R(r)$ and $S(s)$ be two relations, R with schema r and and S with s. The relation $R \div S$ is a relation with schema $r - s$ (all attributes of R not in S). A tuple t is in $R \div S$ if and only if both of two conditions hold:*

1. $t \in \Pi_{r-s}(R)$
2. *For every tuple $t_S \in S$, there is a tuple $t_R \in R$ satisfying both of the following:*

(a) $t_R[s] = t_S[s]$ *("matching" attributes of* t_R *and* t_S *hold equivalent values)*

(b) $t_R[r-s] = t$ *(the attributes* $r-s$ *of* t_R *are equal to an element* t *of the division)*

Figure 2 provides an example of our previous question involving bees and their places of residence.

Name	DateOfBirth	HiveAddress
Joe	10/20/1901	222 Smokey Street
Sally	3/4/2001	321 Pooh Circle
Frank	10/20/1901	321 Pooh Circle
Frank	10/20/1901	222 Smokey Street

A relation **R**.

Name
Frank

Figure 2: : $\Pi_{Name,HiveAddress}(R) \div \Pi_{HiveAddress}(R)$.

It might not be obvious how we could write division with our basic operations, but Codd had a colleague point this out to him back when division was originally proposed as a basic operation [3]:

$$R \div S = \Pi_{r-s}(r) - \Pi_{r-s}((\Pi_{r-s}(R) \times S) - \Pi_{r-s,s}(R))$$

To understand how this expression is true, observe that $\Pi_{r-s}(R)$ gives us all tuples satisfying the first condition of Definition 4. The expression

$$(\Pi_{r-s}(R)) \times S) - \Pi_{r-s,s}(R))$$

removes all tuples not satisfying the second condition. $\Pi_{r-s}(R) \times S$ is a relation on schema r and pairs every tuple of $\Pi_{r-s}(R)$ with every tuple in S. $\Pi_{r-s,s}(R)$ simply reorders the attributes of R. The difference of these two parts is the pairs of tuples from $\Pi_{r-s}(R)$ and S not in R. Then subtracting $(\Pi_{r-s}(R) \times S) - \Pi_{r-s,s}(R))$ from $\Pi_{r-s}(R)$ gives us the tuples in $R \div S$.

2.3.3 Extending Relational Algebra

From the operations we have just defined, there are many ways to extend the algebra with new operations. We mentioned new join operations as one way. Others include aggregate operations for sums over attributes, arithmetic operations as part of the projection, and some notational schemes for assigning relations to variables.

3 Query Optimization

Now we turn our attention to the benefits of having a relational algebra. First and foremost is simply having a means for expressing queries without a reliance on hardware or other implementation details. Of primary importance to computer scientists, however, is the use of relational algebra for query optimization and query language categorization. We will not go into the details of the latter, but it is much like Turing completeness.

Query languages are simply programming languages meant to behave like relational algebra. Due to constraints within the machine, however, these languages are not always perfect implementations of the pure math we have worked with so far. We will consider the language SQL (pronounced "sequel") and how queries from this language may be optimized. A query is executed optimally if it uses the fewest resources and takes the least amount of time compared to all other execution strategies. Throughout this section we will work with the list of relations given in Figure 3. This is the canonical example of a company database. The domain of each attribute is not necessary to know, but most should be self-evident.

Ioannidis [4] provides the best discussion of optimizing query languages using relational algebra. All diagrams following are redrawn from that paper. A brief discussion is also found in Silberschatz et al. [5].

emp(name,age,sal,dno)
dept(dno,dname, oor,budget,mgr,ano)
acnt(ano,type,balance,bno)
bank(bno,bname,address)

Figure 3: : The relations we work with in Section 3

A basic SQL query has three components:

select	<attribute names>
from	<relations>
where	<predicate statements>

The *select* clause is a list of attribute names we want in the result query (much like a *projection*); $from$ is a Cartesian product of the relations involved in this query; and *where* is a list of predicates to select with, applied to the cross product of $from$ (just like *selection*). The "list" in the *where* clause is a series joined by any combination of $\{and, or, not\}$, where these are equivalent to the eponymous boolean operations. An example query can be found in Figure 4.

```
select   name, floor
from     emp, dept
where    emp.dno=dept.dno and sal>100K
```

Figure 4: : Note in this example how we use *relation.attribute* for like named attributes and keywords like *and* replace some boolean operators.

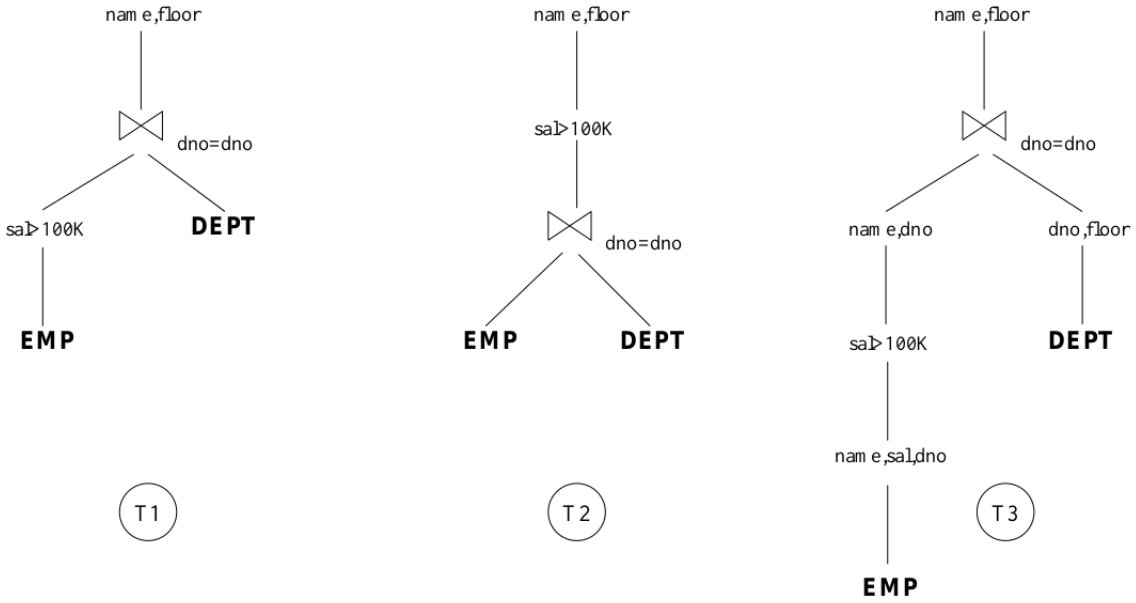

Figure 5: : These trees correspond to the SQL query in Figure 4. Select and project symbols are removed for brevity, but understand boolean statements to be select statements and attribute listings to be projections. Note *join* ($\bowtie$) uses leaves as inputs.

Any SQL statement may be broken down into a select-project-join query in relational algebra. Furthermore, this query may be represented in a *query tree* whose leaves are database relations and non-leaf nodes are algebraic operators select, project, or join. Any non-leaf node represents the relation created by applying the given operation on its children. Examples of query trees for Figure 4 are in Figure 5

We can see the number of possible trees for complicated queries may be immense, and in order to truly optimize, we must consider the execution costs of every tree. We can not store this many trees in memory and we certainly do not wish to consider all of them. Thus, database management systems (DBMSs) usually form several restrictions on the space of all trees considered.

3.0.4 Restriction 1

The first restriction is for selections and projections to be processed on the fly. That is, we never consider them to generate intermediate relations. This restriction is somewhat implementation specific, but in Figure 5 we would consider the leaves generated by select and project statements to be part of join operations above them and never creating "temporary variables". For queries with no joins, this restriction is not applicable.

This restriction eliminates only sub-optimal query trees. Separate processing of selections and projections would only incur additional costs. Hence only different formations of joins are considered for alternative query trees, rather than including variations on the projections and selections.

3.0.5 Restriction 2

Note that joins are both commutative

$$R_1 \bowtie R_2 = R_2 \bowtie R_1$$

and associative

$$(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)$$

From this fact we could produce all alternative join trees. The number of these is huge, on the order of $\Omega(N!)$ for N relations. So to further restrict the number of trees we must examine, the second restriction helps deal with some of these joins.

The second restriction states that cross products are never formed unless the query itself asks for them. Relations are always combined through a join. Consider Figure 6 with a query and its trees. Because *emp* and *acnt* are not joined in the query (note no comparison in the *where* clause), they may be joined as shown in Figure 6 or combined via cross product to answer the query. Restriction 2 would remove tree $T3$ in the example.

Because Cartesian products are expensive computationally (multiple loops), this restriction nearly always eliminates sub-optimal queries. That this is false in certain fringe cases is an acceptable cost for nearly all general purpose systems.

3.0.6 Restriction 3

This last restriction is made by a select few DBMSs, mostly because it is of a more heuristic nature than our previous two. The restriction requires at least one operand of

```
select   name, floor, balance
from     emp, dept, acnt
where    emp.dno=dept.dno and dept.ano=acnt.ano
```

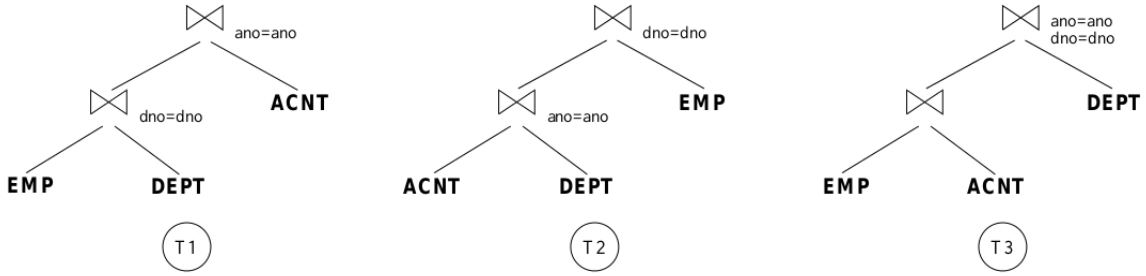

Figure 6: : Examples of join trees for the query; $T3$ has a cross product.

any join operation to be a database relation and never an intermediate result. For the query in Figure 7 we provide a tree satisfying this restriction.

```
select   name, floor, balance, address
from     emp, dept, acnt, bank
where    emp.dno=dept.dno and
         dept.ano=acnt.ano
```

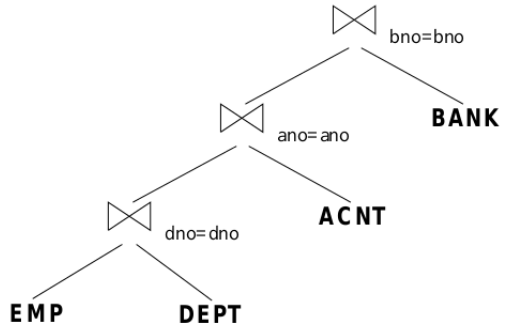

Figure 7: : A query and its corresponding left-deep tree, satisfying Restriction 3.

We call the tree in Figure 7 a **left-deep** tree. Right-deep trees will similarly fulfill restriction three. The advantage of these structures lies in the ability to reuse preexisting indices and exploit implementation details regarding nested joins. The details of these advantages are beyond the scope of this paper, but rest assured they are significant.

The third restriction will sometimes remove optimal trees, but it is genuinely agreed these cases are few and the optimal left-deep tree is barely less optimal than the true optimal tree.

3.0.7 Combining the Restrictions

The purpose of all the restrictions was to reduce the number of trees needing to be examined. Ioannidis claims the number of trees to be explored can be reduced to $O(2^N)$ for many queries with N relations. This is a massive improvement from where we started with at least $N!$ trees. Further enhancements may be made if we continue exploiting properties of the relational algebra with low-level implementation details in mind [4].

4 Conclusion

Throughout this paper we attempted to provide an introduction to relational algebra and an example of the benefits we reap from a rigorous mathematical approach to data. In Section 2 we examined the basic operations and several more interesting derived operations from E.F. Codd's relational algebra. This provided us with a hardware independent and mathematical structure for answering questions about data. Section 3 explored the use of relational algebra for structuring a database query and optimizing that structure. Gains in speed and efficiency are essential for some industries. The use of an algebra to describe something as ambiguous as data is invaluable in general.

References

[1] David L. Childs, *Description of a set-theoretic data structure*, Proceedings of the December 9-11, 1968, fall joint computer conference, part I (New York, NY, USA), AFIPS '68 (Fall, part I), ACM, 1968, pp. 557–564.

[2] E. F. Codd, *A relational model of data for large shared data banks*, Commun. ACM **13** (1970), no. 6, 377–387.

[3] E. F. Codd, *Relational completeness of data base sublanguages*, Database Systems, Prentice-Hall, 1972, pp. 65–98.

[4] Yannis E. Ioannidis, *Query optimization*, 1996.

[5] A. Silberschatz, H.F. Korth, and S. Sudershan, *Database System Concepts*, 4th ed., McGraw-Hill, Inc. New York, NY, USA, 2002.