

# The Relational Data Model: Additional Operations

## 1 Overview

- While the fundamental relational algebra operations — select, project, union, set-difference, cartesian-product, rename — form the basis for a broad spectrum of relational database queries and manipulations, it remains possible to define other operations over relations.
- Some operations are just notational shortcuts — they are completely rewritable in terms of the 6 fundamentals, but for convenience we abstract them into operations of their own. These are the “additional operations” in Section 2.
- On the other hand, the “classical” relational algebra itself has been extended with new operations, with new meanings and manipulations. These are the “extended operations” in Section 3.

## 2 Additional Operations

- “Additional operations” refer to relational algebra operations that can be expressed in terms of the fundamentals — select, project, union, set-difference, cartesian-product, and rename.
- The compositions of these operations are so lengthy, yet so common, that we define new operations for them, based on the fundamentals. Kind of a mathematical “syntactic sugar.”

### 2.1 Set-Intersection

- The *set-intersection* operation is a binary operation on relations  $r$  and  $s$  that is denoted by the traditional intersection symbol,  $\cap$ .  $r \cap s$  results in all tuples  $t$  such that  $(t \in r) \wedge (t \in s)$ .

- Set-intersection is defined in terms of set-difference:

$$r \cap s = r - (r - s)$$

- Thus, set-intersection must follow the same *compatibility* rules as set-difference: same arity, corresponding domains.

## 2.2 Natural-Join

- The *natural-join* operation is a binary operation on relations  $r(R)$  and  $s(S)$  that is denoted by the symbol  $\bowtie$ . Intuitively, a natural-join “matches” the tuples of  $r$  with the tuples of  $s$  based on attributes that are both in  $r$  and  $s$ .
- If we take the relational schemas  $R$  and  $S$  as sets of attributes, then we can define “attributes that are in both  $r$  and  $s$ ” as  $R \cap S = \{A_1, A_2, \dots, A_n\}$ . With that, we can formally define  $r \bowtie s$  as:

$$r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \dots \wedge r.A_n = s.A_n}(r \times s))$$

- Note that  $R \cup S$  removes duplicate attribute names, so  $r \bowtie s$  will only have one attribute  $A_k \forall A_k \in R \cap S$ .
- Natural join is *associative* — that is,  $(a \bowtie b) \bowtie c = a \bowtie (b \bowtie c)$ .
- When  $r$  and  $s$  do not have any common attributes — i.e.,  $R \cap S = \emptyset$  — then  $r \bowtie s = r \times s$ .

### 2.2.1 Theta Join

- One can generalize the natural-join operation into a *theta join*, so named because instead of the specific “attribute-matching” condition involved in natural-join, we allow  $\theta$  to be any predicate on the attributes in  $R \cup S$  for  $r(R)$  and  $s(S)$ .
- Thus, we have  $r \bowtie_{\theta} s$ , defined as:

$$r \bowtie_{\theta} s = \sigma_{\theta}(r \times s)$$

## 2.3 Division

- The *division* operation is a binary operation, notated as  $\div$ , on relations  $r(R)$  and  $s(S)$  such that  $S \subseteq R$ . Intuitively, division is a “for all” query — it returns the tuples in  $r$  that “match” *all* of the tuples in  $s$ .

- Formally,  $r \div s$  is a relation with schema  $R - S$  whose tuples  $t$  satisfy both of these conditions:

1.  $t \in \Pi_{R-S}(r)$
2.  $\forall t_s \in s, \exists t_r \in r$  that satisfies both of:
  - (a)  $t_r[S] = t_s[S]$
  - (b)  $t_r[R - S] = t$

- Division can also be defined in terms of the relational algebra. Again, we start with  $r(R)$  and  $s(S)$  with  $S \subseteq R$ :

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)) \quad (1)$$

- This is quite an eye-ful, so we break it down like this:
  - First, we take  $\Pi_{R-S}(r)$ . This defines the relation for the first condition of the formal definition:  $t \in \Pi_{R-S}(r)$ , or the tuples in  $r$  with their  $s$ -shared attributes removed.
  - Now, from this, we perform set-difference using:

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

- So these tuples are removed from  $\Pi_{R-S}(r)$ . What are these tuples exactly? We start with  $\Pi_{R-S}(r) \times s$ : it takes all of the tuples in  $r$  and removes any attributes that  $r$  shares with  $s$ , then pairs those tuples with *every* tuple in  $s$ .
  - $\Pi_{R-S,S}(r)$  just rearranges the attributes in  $r$  so that the attributes that are in  $r$  alone are listed first, followed by the attributes that  $r$  shares with  $s$ . This ensures that the relation of the first difference term,  $\Pi_{R-S}(r) \times s$ , is compatible with  $\Pi_{R-S,S}(r)$  (same arity, corresponding domains).
  - So, the set-difference operation can take place, and the result would be the  $r$ -to- $s$  tuple matchups that are *not* in  $r$ .
  - If we then remove the  $s$  attributes from that relation, resulting in  $\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$ , we now have the tuples in  $r$  for which at least one tuple in  $s$  does *not* match. Since  $r \div s$  is about tuples in  $r$  that have a corresponding match for *all* tuples in  $s$ , then these are precisely the tuples that we *don't* want, and so we subtract them from  $\Pi_{R-S}(r)$ .
- Phew! When all is said and done, just remember that division,  $r \div s$ , is all about “for all” — tuples in  $r$  that match *all* tuples in  $s$ .

## 2.4 Assignment

- You’ve probably gotten a sense, particularly with division, that relational algebra feels a lot like programming: there are many steps to some expressions, with intermediate or temporary relations along the way.

- For this very reason, we have the *assignment* operation, which works a lot like assignments in a programming language. It is notated with the left-pointing arrow  $\leftarrow$ :

$$variable \leftarrow E$$

where  $E$  is any relational algebra expression.

- The assignment operation is more of a notational convenience rather than a real relational operation — it helps human beings with writing out complex relational expressions in steps so that they can be more easily understood.
- Revisiting the breakdown of the division operation, we can use assignment to rewrite (1) this way:

$$\begin{aligned} temp1 &\leftarrow \Pi_{R-S}(r) \\ temp2 &\leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r)) \\ r \div s &= temp1 - temp2 \end{aligned}$$

### 3 Extended Operations

- In this section, we discuss “extended” operations — new operations that introduce new concepts beyond what can be built from the fundamental relational operations.
- While they are technically “extensions” to the algebra, they still follow the mathematical rigor and precision that allow us to draw sweeping and powerful conclusions from simpler concepts.

#### 3.1 Generalized Projection

- The *generalized-projection* operation extends the fundamental projection operation by allowing arithmetic (or, in the most general case, overall transformative) functions in the projection’s attribute list. It is still denoted with  $\Pi$ , but now the straight-up attribute list  $A$  has changed into an expression list  $F_1, F_2, \dots, F_n$ :

$$\Pi_{F_1, F_2, \dots, F_n}(E)$$

$E$  is any relational algebra expression, which is of course a relation.  $F_k$  may be any expression involving constant values and the attributes of  $E$ ’s resultant relation schema.

- By default, new attributes resulting from evaluating  $F_k$  have no name. We can use the fundamental rename operation to give it a name:

$$\rho_{x(a_1, a_2, \dots, a_n)}(\Pi_{F_1, F_2, \dots, F_n}(E))$$

- ... but that can be too verbose or redundant, so as a notational convenience, we can prepend an *as new\_name* suffix to  $F_k$ :

$$\Pi_{F_1 \text{ as } a_1, F_2 \text{ as } a_2, \dots, F_n \text{ as } a_n}(E)$$

- Note how generalized projection doesn't add a completely new operation per se — it just adds new functionality to the fundamental projection operation.

## 3.2 Aggregate Functions

- An *aggregate function* takes a collection of values and returns a single value as its result. Examples of aggregate functions include:

**sum** takes a collection of numerical values and adds them up.

**avg** also takes a collection of numerical values and returns their mean.

**count** returns how many items are in a given collection.

**min** returns the minimum value in a collection.

**max** returns the maximum value in a collection.

- Of course one can define all kinds of other possible aggregate functions — median and standard deviation come to mind.
- Note how we say *collection* of values, not a *set*; this is a deliberate choice, because an individual value may occur multiple times within that collection. However, as with sets, the order in which values appear does not matter. This type of collection is formally known as a *multiset*.
  - Sometimes, you *do* want the aggregate function to ignore duplicates — for instance if you want to count the number of individual values in a relation. In this case, the suffix *-distinct* may be used to indicate that the aggregate function removes duplicates before performing its calculation. This is primarily a notational convenience; for all intents and purposes, *count* and *count-distinct* are just different aggregate functions, the way *sum* is different from *avg*.
- An aggregate function is notated using a “calligraphic G:”  $\mathcal{G}_{F(A)}$ . The subscript  $F(A)$  represents the specific aggregate function  $F$  to be applied, along with the attribute  $A$  on which to perform the function:

$$\mathcal{G}_{F(A)}(E) \tag{2}$$

As before,  $E$  represents any relational algebra expression, and of course,  $A$  is part of  $E$ 's list of attributes.

- A frequent aggregate function parameter is the attribute in  $E$  over which the aggregate function is to be applied. For instance, in a relation containing cities and their populations, the total population would be written as:

$$\mathcal{G}_{sum(population)}(city)$$

- The result of  $\mathcal{G}_{F(A)}(E)$  is a relation with a single attribute, containing a single tuple that holds the result of the aggregate function.

### 3.2.1 Groups

- The expression in (2) performs the aggregate function over the entire relation represented by  $E$ . Sometimes, we want to perform the aggregation multiple times over separate sets of tuples in  $E$ . These separate sets are called *groups*.
- Typically, groups are specified as a list of attributes  $G_1, G_2, \dots, G_n$ , meaning that, for a given relation, tuples should first be “clustered” into partitions such that these tuples share the same values for  $G_1, G_2, \dots, G_n$ . The aggregate function should then be applied to each cluster:

$$_{G_1, G_2, \dots, G_n} \mathcal{G}_{F(A)}(E)$$

- With groups, the resulting aggregate relation would have as many tuples as there were groups in the original, partitioned relation.

### 3.2.2 General Form of the Aggregation Operation

- As a final extension, we note that we can perform more than one aggregate function  $F_i(A_i)$  over the groups in a relational algebra expression. Thus, the general form of the *aggregation* operation is:

$$_{G_1, G_2, \dots, G_n} \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_m(A_m)}(E)$$

- The resulting relation in an aggregation operation has a schema consisting of the attributes  $G_1, G_2, \dots, G_n$  plus one attribute for each aggregate function  $F_i$ . A tuple in this relation has the form  $(g_1, g_2, \dots, g_n, a_1, a_2, \dots, a_m)$  where each  $a_i, 1 \leq i \leq m$ , is the result of  $F_i(A_i)$  applied to the multiset defined by the group  $(g_1, g_2, \dots, g_n)$ .

## 3.3 Outer Join

- In a perfect world, we would have data for everyone and everything, and all relations would be complete in terms of their data and data that refers to other relations. In that perfect world, natural join ( $\bowtie$ ) pretty much takes care of our relation-connection needs.

- But it isn't a perfect world. One thing that natural join does, by its own definition, is drop any tuples from either relation that does not match a tuple from the other relation (i.e., no shared attributes with the same values). In some cases, this is appropriate, but in others, we don't want to lose data.
- Enter the *outer join*. An outer-join operation does the same thing as natural join, except that it *preserves* tuples without a match in the other relation. The tuple that does not have a match uses *null* values to populate the attributes from the second relation.
- There are three types of outer join, based on which non-matching tuples “stick around” after the operation:
  - *Left outer join*, written as  $r \bowtie\! \! \! \bowtie s$ , keeps the non-matching tuples in  $r$  and uses *null* values for the attributes that come from  $s$ .
  - *Right outer join*, written as  $r \bowtie\! \! \! \sqsubset s$ , keeps the non-matching tuples in  $s$  and uses *null* values for the attributes that come from  $r$ .
  - *Full outer join*, written as  $r \bowtie\! \! \! \sqcup s$ , keeps the non-matching tuples from both  $r$  and  $s$ , and uses *null* values for the attributes that come from the “other” relation.
- Fun fact: outer join is actually expressible in terms of the fundamental relational operations. For example, left outer join can be rewritten as:

$$r \bowtie\! \! \! \bowtie s = (r \bowtie s) \cup ((r - \Pi_R(r \bowtie s)) \times \{(null, \dots, null)\})$$

Here,  $\{(null, \dots, null)\}$  is a constant (literal), single-tuple relation with schema  $S - R$ .

- Can you see how the other outer joins can be rewritten in terms of the more fundamental operations (hint hint)?

### 3.4 Null Values

- The outer-join operations bring the notion of *null* values into the relational algebra. How does one deal with such values formally?
- In reality, there are a lot of plausible approaches for handling *null*. If we stick to the definition of *null* as an *unknown or non-existent value*, we can establish the following:
  - Any arithmetic operations involving *null* must return *null*.
  - Comparing anything to *null* really doesn't have much meaning, so we create a new “boolean value” in this case — *unknown*, meaning that we really can't say whether a comparison to *null* is true or false.
  - By defining the new boolean value *unknown*, we need to determine how *unknown* interacts with the other boolean values, *true* and *false*, in terms of the boolean

operations  $\wedge$ ,  $\vee$ , and  $\neg$ . These definitions are actually fairly intuitive, if you think about it:

$$true \wedge unknown = unknown$$

$$false \wedge unknown = false$$

$$unknown \wedge unknown = unknown$$

$$true \vee unknown = true$$

$$false \vee unknown = unknown$$

$$unknown \vee unknown = unknown$$

$$\neg unknown = unknown$$

- With the effect of *null* defined at the value level, we then consider how *null* (and its sister value *unknown*) should affect relational operations:

**select** When  $P$  in  $\sigma_P(E)$  is *unknown* for a particular tuple, then the tuple does not get included in the result.

**join** Because joins (natural, outer) come down to a selection on a Cartesian product, then the behavior of the select operation determines the behavior of the joins. Essentially, since all comparisons against *null* are *unknown*, then tuples with *null* values in the common attributes won't be included in the join result.

**project** The main issue with *null* and projection is what to do with potentially duplicate tuples — that is, are tuples with the same non-null attributes, and with *null* everywhere else, to be considered as duplicates and therefore “unified” in the projection result? In this case, we say yes: tuples with the same non-null attributes and nulls everywhere else *are* duplicates, and so only one of these tuples gets included in the projection.

**union, intersection, difference** We apply the same *null* handling with these operations as we did with projection, since this was based on the premise of not having duplicates in a set.

**generalized projection** Calculated or derived attributes follow the above guidelines for calculations when *null* is involved; otherwise this is just like “standard” projection.

**aggregation** Same logic as with projection: tuples with matching *null* and non-null values in the same grouping attributes will end up in their own group.

However, instead of propagating *null* in the actual aggregate function calculations, we choose to remove them from the multiset before applying the aggregate function. After all, it wouldn't make sense to say that the count of a particular attribute in a relation is *null* just because one of its tuples has a *null* in that attribute.

## 4 Modification of the Database

- So far, all of the operations that we have discussed “derive” new relations from others, but don’t actually modify or alter the original relations.
- How does one specify operations that *change* relations “in place?”
- The general guideline is to use the assignment operator, but to make assignments to *existing* relations instead of designating new temporary variables.

### 4.1 Deletion

- Deletion of tuples is effectively a set-difference operation that is “permanent.” Thus, we can write deletion as:

$$r \leftarrow r - E$$

$E$  in this case is any relational algebra expression that determines the tuples that are to be removed from  $r$ .

### 4.2 Insertion

- Similarly to deletion, insertion of tuples can be viewed as a union operation that is made permanent. Thus, insertion is:

$$r \leftarrow r \cup E$$

$E$  once more is a relational algebra expression that determines the tuples to be set-unioned with (thus “inserted into”)  $r$ .

- Note how this definition includes plain old data loading without any problems — in that case,  $E$  is simply a constant or literal relation.

### 4.3 Updating

- An *update* modification changes one or more values *within* a tuple. Once more, we find that updating is just a persistent version of another relational operation, this time generalized projection:

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_n}(r)$$

- Specifically, the function  $F_i$  may be just the  $i$ th attribute of  $r$  if it doesn’t need to be updated, or otherwise an expression involving constants and the attributes of  $r$  if the  $i$ th attribute is involved in the update.

- The schema of the generalized-projection operation must also match the current schema of  $r$ , so in all likelihood we will perform attribute-level renames to ensure this match.
- Of course, performing the generalized projection on  $r$  will modify all tuples in  $r$ , which isn't always what we want. To update only the tuples in  $r$  that satisfy a predicate  $P$ , we say:

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_n}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

This restricts the generalized-projection operation only to the tuples for which  $P$  is true, then includes all other untouched tuples.