CS 186
Fall 2021

Relational Algebra

1 Motivation

In the previous notes we talked about how SQL is a declarative programming language. This means that you specify what you want, but you don't have to specify how to do it. This is great from a user's perspective as it makes the queries much easier to write. As database engineers, however, we often want a language that is more expressive. When we study query optimization in a few weeks we're going to want a way to express the many different valid plans a database can use to execute a query. For this we will use **Relational Algebra**, a procedural programming language (meaning that the query specifies exactly what operators to use and in what order).

2 Relational Algebra Introduction

All of the operators in relational algebra take in a relation and output a relation. A basic query looks like this:

$$\pi_{name}(dogs)$$

The π operator picks only the columns that it wants to advance to the next operator (just like SQL SELECT). In this case, the operator takes the `dogs` relation in as a parameter and returns a relation that only has the `name` column of the `dogs` relation. An important fact about relational algebra is that the relations are sets of tuples, meaning that they cannot have duplicates in them. If the dogs relation is initially:

name	**age**
Scooby	10
Buster	15
Buster	20

The query above would return:

name
Scooby
Buster

Initially the two Busters are different because they have different ages, but once you get rid of the age column, they become duplicates, so only one remains in the output relation.

Let's formally introduce the relational algebra operators.

3 Projection (π)

We have already been introduced to the **projection operator** which takes in a single relation as input and selects only the columns specified. The columns are specified in the subscript of the

operator like almost all parameters to operators. The output schema of projection is determined by the schema of the column list. The projection operator is relational algebra's version of the SQL SELECT clause.

We now can express SQL queries involving just the SELECT and FROM clauses with relational algebra. For example the SQL query:

```
SELECT name FROM dogs;
```

Can be represented with the expression we introduced in section 2:

$$\pi_{name}(dogs)$$

Note that there is no operator equivalent to the FROM operator in relational algebra because the parameters of these operators specify which tables we pull from.

4 Selection (σ)

The **selection operator** takes in a single relation and filters rows based on a certain condition. The output schema will be the same as the input schema, and duplicate elimination is not needed for selection. Don't let the name confuse you - this operator is equivalent to SQL's WHERE clause, **not** its SELECT clause. Let's try to express the following query in terms of relational algebra:

```
SELECT name, age FROM dogs WHERE age = 12;
```

The equivalent relational algebra expression is:

$$\sigma_{age=12}(\pi_{name,age}(dogs))$$

Another correct expression for that query is:

$$\pi_{name,age}(\sigma_{age=12}(dogs))$$

This illustrates the beauty of relational algebra. There is only one (reasonable) way to write SQL for what the query is trying to accomplish, but we can come up with multiple different expressions in relational algebra that get the same result. In the first expression we select only the columns we want first, and then we filter out the rows we don't want. In the second we filter the rows first and then select the columns. We will soon learn ways to evaluate which of these plans is better!

The selection operator also supports compound predicates. The $\wedge$ symbol corresponds to the AND keyword in SQL and the $\vee$ symbol corresponds to the OR keyword. For example,

```
SELECT name, age FROM dogs WHERE age = 12 AND name = 'Timmy';
```

is equivalent to

$$\pi_{name,age}(\sigma_{age=12 \wedge name='Timmy'}(dogs))$$

5 Union ($\cup$)

The first way we will learn how to combine data from different relations is with the **union operator**. Just like the UNION clause in SQL, we take all the rows from each tuple and combine them removing duplicates along the way. As an example, say we have a dogs table:

name	**age**
Scooby	10
Buster	15
Garfield	20

and a cats table that looks like this:

name	**age**
Tom	8
Garfield	10

The expression:

$$\pi_{name}(dogs) \cup \pi_{name}(cats)$$

would return:

name
Scooby
Buster
Tom
Garfield

Note that Garfield only shows up once because relations are sets of tuples and remove all duplicates as a result. Additionally, note that one rule for all of these set operators is that they must operate on relations that have the same number of attributes (columns), and the attributes in corresponding positions must have the same type. It would not be legal to union a relation with two columns with a relation that only has one column nor would it be legal to union a relation with a column of strings with another relation with one column of ints.

6 Set Difference (-)

Another set operator is the **set difference** operator. Same as with union, both input relations must be compatible (the columns must be in the same order with the same type). Set difference is equivalent to the SQL clause EXCEPT. It returns every row in the first table except the rows that also show up in the second table. Similar to selection, no duplicate elimination is needed for set difference. If you run:

$$\pi_{name}(dogs) - \pi_{name}(cats)$$

expression on the dogs and cats table introduced in the previous section you would get:

name
Scooby
Buster

Garfield does not show up because he is in the cats table, and none of the cats' names will show up because it is only possible for rows from the first relation to show up in the output.

7 Intersection ($\cap$)

Intersection is just like the INTERSECT SQL operator in that it only keeps rows that occur in both tables in the intersection. If you run:

$$\pi_{name}(dogs) \cap \pi_{name}(cats)$$

on the tables introduced in section 5 you would get:

name
Garfield

because Garfield is the only name to occur in both tables.

8 Cross Product ($\times$)

The cross product operator is just like performing a Cartesian product in SQL. The output is one tuple for every possible pair of tuples from both relations. The schemas of the two input relations do not have to be compatible because cross product directly concatenates them. No duplicate elimination is needed because none will be generated. As an example, say we have a dogs table:

name	age
Scooby	10
Buster	15
Garfield	20

and a parks table:

park	city
Golden Gate Park	San Francisco
Central Park	New York City

The relational algebra equivalent of

```
SELECT * FROM dogs, parks;
```

is

$$dogs \times parks$$

and the output will be:

name	age	park	city
Scooby	10	Golden Gate Park	San Francisco
Scooby	10	Central Park	New York City
Buster	15	Golden Gate Park	San Francisco
Buster	15	Central Park	New York City
Garfield	20	Golden Gate Park	San Francisco
Garfield	20	Central Park	New York City

In fact, the cross product ($\times$) is the basis for the inner join, which we will go over next.

9 Joins ($\bowtie$)

We haven't yet discussed how to represent joins in relational algebra - let's fix that! To inner join two tables together, write the left table on the left of the $\bowtie$ operator, put the join condition in the subscript, and put the right operator on the right side. To join together the cats table with the dogs table on the name column, you would write:

$$cats \bowtie_{cats.name=dogs.name} dogs$$

If you don't specify the join condition, it becomes a natural join. Recall from the SQL notes that a natural join joins together all columns from each table with the same name. Therefore, you could also write the same query as above like:

$$cats \bowtie dogs$$

Formally, we refer to the inner join operator as a **Theta Join** ($\bowtie_\theta$). The θ refers to the join condition, so for the expression from above, the θ join condition is $cats.name = dogs.name$.

The $\bowtie$ operator performs an inner join, which is the only join we will cover for relational algebra expressions in this class. There are ways to derive right, left, and full outer joins from the operators we have already introduced, but that is beyond the scope of this class.

Just like the selection operator σ, the join operator $\bowtie$ also supports the compound predicate operators $\wedge$ (AND) and $\vee$ (OR).

Theta joins and natural joins can actually be derived from just a cross product ($\times$) and a conjunction of selections (σ). For example,

$$cats \bowtie_\theta dogs$$

can be rewritten as

$$\sigma_\theta(cats \times dogs)$$

and the natural join

$$cats \bowtie dogs$$

can be rewritten as

$$\sigma_{cats.col1=dogs.col1 \wedge ... \wedge cats.colN=dogs.colN}(cats \times dogs)$$

10 Rename (ρ)

The **rename operator** essentially accomplishes the same thing as aliasing in SQL. It is used to change the schema by renaming the relations and/or their attributes. For example, if you wanted to avoid having to include the table name for the rest of the expression like you would for the expressions in the join section, you could instead write:

$$cats \bowtie_{name=dname} \rho_{name->dname}(dogs)$$

This expression renames the dogs relation's name column to dname first, so there is no conflict in column names. You can no longer use a natural join anymore because the columns do not have the same name, but you no longer need to specify which relation the column is coming from if you want to include other operators.

11 Group By / Aggregation (γ)

The final relational algebra operator we will cover is the **groupby / aggregation operator**, which is essentially equivalent to using the GROUP BY and HAVING clauses in SQL. For example, the SQL query

```
SELECT age FROM dogs GROUP BY age HAVING COUNT(*) > 5;
```

can be expressed in relational algebra as

$$\gamma_{age,COUNT(*)>5}(dogs)$$

Furthermore, the γ operator can be used to select aggregate columns, such as MAX, MIN, SUM, COUNT, etc. from SQL. This modified query from earlier

```
SELECT age, SUM(weight) FROM dogs GROUP BY age HAVING COUNT(*) > 5;
```

can be expressed in relational algebra as

$$\gamma_{age,SUM(weight),COUNT(*)>5}(dogs)$$

12 Practice Questions

Given the following two relations:
`teams(teamid, name)`
`players(playerid, name, teamid, position)`

Answer the following questions:

1. Write an expression that finds the name and playerid of every player that plays the "center" position.
2. Write an expression that finds the name of every player that plays on the "Warriors". How would this expression change if we renamed players' teamid column to pteamid?
3. Write an expression that finds the teamid of all teams that do not have any players.
4. Write an expression that is equivalent to the following SQL query:

```
SELECT teamid AS tid
FROM players
WHERE players.teamid NOT IN
    (SELECT teamid FROM teams)
AND position='shooting guard';
```

13 Solutions

1. $\pi_{name,playerid}(\sigma_{position='center'}(players))$. We first filter out the rows for players who aren't centers, then we project only the columns that we need.
2. $\pi_{players.name}(\sigma_{teams.name='Warriors'}(teams \bowtie_{teams.teamid=players.teamid} players))$. We first join together the teams and players table to get all the information that we need, then we filter out the rows that aren't for players who play for the Warriors, then we finally project the only column that we're looking for.
3. $\pi_{teamid}(teams) - \pi_{teamid}(players)$. All teams must be in the teams table so we first get all their teamids. Then we subtract any teamid that appears in the players table, because if that teamid appears in the players table it implies that the team has a player on it. We are then left with only teamids of teams that don't have any players.
4. $\rho_{teamid \to tid}(\pi_{teamid}(\sigma_{position='shooting\ guard'}(players)) - \pi_{players.teamid}(players \bowtie_{players.teamid=teams.teamid} teams))$
   We first filter out rows for players who aren't shooting guards, then we only project the column we need, teamid. We then use set difference to only keep players who play for a team not in the teams table. Finally, we use the renaming operator to rename teamid to tid.