RELATIONAL ALGEBRA: LOGIC, NOT MAGIC!

In this lesson, we'll be answering the following questions:

1. What is **relational algebra,** and why is it so important to the relational model? What does it mean to say it is **complete?**
2. How do the **select, project,** and **rename** operators work?
3. How do the set **union** and set **difference** operators work?
4. How does the **cross-product** operator work, and how does this relate to **full joins, natural joins,** and **outer joins?**

If the relational model is based on the idea that users of data can interact with from the "logical" viewpoint of tables and rows, **relational algebra** (and its sister language, **relational calculus)** provide the language to express this logic. Until Codd's paper (which launched the relational model), relational algebra was mostly an obscure language, of interest mostly to set theorists and logicians. Codd, however, showed that it could be put to eminently practical use: so long as users understood just a few (five!) basic operations, they could combine these questions to produce ANY database query they liked! Here, we'll be looking at the basics of Codd's insight, without *too* much concern about becoming experts on it (relational algebra is only rarely used in "practical" applications). Later, when we turn our attention to Structured Query Language (SQL) we'll see many of these ideas again.

The basic idea behind relational algebra is very simple. We take one or more relations as inputs, do something to them, and then produce a *different* relation as an output (so, the initial relations are untouched!). Relations are **closed** under the operators of relational algebra, which means that we will ALWAYS get a relation back (which can then be used as in the input for more relational algebra!). In relational calculus (not covered here, but which provides equivalent functionality), we will always get a back a TRUE or FALSE to a question posed in the correct manner.

THE BASICS: SELECT, PROJECT, RENAME, UNION, DIFFERENCE

Let's suppose that Tony Stark (the "Iron Man" character, apparently modeled after Oracle founder and database superstar Larry Ellison) has the following two relations R1 and R2, containing information on some of his friends: The second table simply contains information on various superheroes' favorite colors, and whether or not they scare Tony Stark.

Relation R1

first_name	last_name	secret_identity	age
Peter	Parker	Spiderman	16
Wanda	Maximoff	Scarlett Witch	35
Thor	Odinson	Thor	1500

Relation R2

secret_indentity	favorite_color	scare_me
Hulk	Green	True
Spiderman	Red	False
Scarlett Witch	Red	True
Black Widow	Black	False

We are now in a place to introduce the basic operators of relational algebra.

Selection $\sigma_{condition}Relation$ is used to select tuples (rows) from a relation (table) that satisfy some condition. So, $\sigma_{\text{age} > 30}R1$ will return just those rows from relation R1 where the age attribute has a value of greater than 30. You can also write this as SELECT age > 30 FROM R1 in **syntactically sugared relational algebra** (basically, relational algebra with less math-y symbols).

SELECT age > 30 FROM R1

first_name	last_name	secret_identity	age
Wanda	Maximoff	Scarlett Witch	35
Thor	Odinson	Thor	1500

Rename ($\rho_{"new\ table\ name"}Relation$) renames a table as "X". This can be super convenient, since it allows us an easy way to reference the results of *pervious* relational algebra operations. So, for example, I could rename the result of the previous SELECT operation (which found "old" superheroes) to be the OldHeroes. There are several different notations for this. These all mean the same thing!

- $\rho_{OldHero}(\sigma_{\text{age} > 30}R1)$
- $OldHero \leftarrow (\sigma_{\text{age} > 30}R1)$

- SELECT age>30 FROM R1 AS OldHero

Projection ($\Pi_{\text{list of attributes}}\boldsymbol{Relation}$) is used to "project" selected attributes from a table, and ignore the others. So, $\Pi_{secret\ identity,scare\ me}R2$ or PROJECT secret_identity, scare_me FROM R2 returns a new table with just these attributes (and excluded favorite_color).

PROJECT secret_identity, scare_me FROM R2	
secret_identity	scare_me
Hulk	True
Spiderman	False
Scarlett Witch	True
Black Widow	False

Union. ($\boldsymbol{FirstRelation} \cup \boldsymbol{SecondRelation}$) "combines" the tuples/rows from two relations so long as they have (1) the same number of attributes and (2) these attributes have the same type (for example, we can combine strings with strings, but not strings with numbers). Trying to do something like $R1 \cup R2$ would fail because R1 has four attributes, while R2 has three. We could, however, create a union of just the "secrect_identity" attribute from both tables like this:

$$(PROJECT\ secret\_identity\ FROM\ R1) \cup (PROJECT\ \text{secret\_identity}\ FROM\ R2)$$

secret_identity
Hulk
Spiderman
Thor
Scarlett Witch
-Black Widow

Notice that even if some identities appeared in both tables (such as Spiderman) these would appear only once in the resulting table.

Difference ($\boldsymbol{FirstRelation} - \boldsymbol{SecondRelation}$) returns the tuples/row that appear in the first relation but do not appear in the second. For example,

$$(PROJECT\text{secret\_identity}\ FROM\ R1) - (PROJECT\ \text{secret\_identity}\ FROM\ R2)$$

secret_identity
Scarlett Witch
Thor

PRODUCTS AND JOINS

Cartesian Product (or "Cross Join"). The real power of relational algebra comes from the ability to combine (or **join)** different tables. In relational algebra, the basic operation underlying this ability is called the **Cartesian product** (or **cross-product).** This operation returns *every possible combination of rows from R1 and R2.* So, if we cross R1 (which has three rows) with R2 (which has four rows), this yields a new relation with 12 total rows. Many of these rows make no intuitive "sense". For example, there first rows contains two secret identities (Hulk and Spiderman) and doesn't seem to represent anything "real" in the superhero world (that is, there in no Hulk-Spiderman combination). For this reason, you only rarely want to use this sort of operation in real-life operations. However, its nevertheless crucially important concept to understand, because all of the other (more useful!) joins are ways of taking the results of this sort of join and "filtering" these results in some way. You want to make really, really sure that you don't accidentally make the database produce unfiltered cartesian products unless you really need them: they are big, slow to compute, and often yield "result" which might well confuse users.

$\boldsymbol{R1 \times R2}$ **(R1 CROSS JOIN R2)**						
first_name	**last_name**	**R1.secret_identity**	**age**	**R2.secret_indentity**	**favorite_color**	**scare_me**
Peter	Parker	Spiderman	16	Hulk	Green	True
Peter	Parker	Spiderman	16	Spiderman	Red	False
Peter	Parker	Spiderman	16	Scarlett Witch	Red	True
Peter	Parker	Spiderman	16	Black Widow	Black	False
Wanda	Maximoff	Scarlett Witch	35	Hulk	Green	True
Wanda	Maximoff	Scarlett Witch	35	Spiderman	Red	False
Wanda	Maximoff	Scarlett Witch	35	Scarlett Witch	Red	True
Wanda	Maximoff	Scarlett Witch	35	Black Widow	Black	False
Thor	Odinson	Thor	1500	Hulk	Green	True
Thor	Odinson	Thor	1500	Spiderman	Red	False
Thor	Odinson	Thor	1500	Scarlett Witch	Red	True
Thor	Odinson	Thor	1500	Black Widow	Black	False

A **natural join** takes the results of the above Cartesian join and throws away any tuples where (1) R1 and R2 have an attribute of the same name and domain, such as secret_identity and (2) these shared attributes have different values. So, for example, a natural join will throw away rows where R1.secret_idenity = "Spiderman" while R2.secret_identity = "Thor." Finally a natural join will keep only ONE copy of the shared attribute. Natural joins are the most common sorts of joins in database operations. A natural join of R1 and R2 looks like this.

R1 ⋈ R2 (R1 NATURAL JOIN R2)					
first_name	**last_name**	**secret_identity**	**age**	**favorite_color**	**scare_me**
Peter	Parker	Spiderman	16	Red	False
Wanda	Maximoff	Scarlett Witch	35	Red	True

Notice that a natural join includes *only* those rows for that have a secret_identity in *both* tables. An **equijoin** is very similar to a natural join. However, the "shared column" (in this case, secret_identity) appears twice (both times with the same values.

	R1 EQUIJOIN JOIN R2					
first_name	**last_name**	**R1.secret_identity**	**R2.secret_identity**	**age**	**favorite_color**	**scare_me**
Peter	Parker	Spiderman	Spiderman	16	Red	False
Wanda	Maximoff	Scarlett Witch	Scarlett Witch	35	Red	True
Thor	Odinson	Thor	Thor	1500	NULL	NULL

OUTER JOINS

Sometimes, though, you'll want to include rows even when they DON'T have matches in the other table. This is called an **outer join,** and it is often used when we are looking for issues re: "referential integrity" (that is, when an entity in one table doesn't "refer" to any entity in another table). There are several types of outer joins:

An **outer left join** between R1 and R2 returns every row that the natural join does plus "unmatched" rows from the "lefthand" relation (in this R1). The unmatched rows will contain "null" for any value that is missing:

R1 OUTER LEFT JOIN R2					
first_name	**last_name**	**secret_identity**	**age**	**favorite_color**	**scare_me**
Peter	Parker	Spiderman	16	Red	False
Wanda	Maximoff	Scarlett Witch	35	Red	True
Thor	Odinson	Thor	1500	NULL	NULL

A **right join** does just the opposite: it returns a natural join plus any unmatched rows from the "righthand" relation (in this case, R2):

R1 OUTER RIGHT JOIN R2					
first_name	**last_name**	**secret_identity**	**age**	**favorite_color**	**scare_me**
NULL	NULL	Hulk	NULL	Green	True
Peter	Parker	Spiderman	16	Red	False
Wanda	Maximoff	Scarlett Witch	35	Red	True
NULL	NULL	Black Widow	NULL	Black	False

PUTTING IT ALL TOGETHER: RELATIONAL ALGEBRA IN ACTION

Now that we've seen how this works in theory, let's try putting it together. Here, I've also used a few *logical operators* such as **OR, AND,** and **NOT** that will familiar to those of you've taken classes in programming, discrete mathematics, or formal logic (and here you thought you'd never get to use these skills!). I've also included some Here are some sample queries Iron Man might run. I've described what they *mean;* however, I'll leave it up to you to determine what they will return.

English Statement	Relational Algebra Statement	Notes
"Show me just the last names and ages from R1"	PROJECT(last_name, age) FROM R1	To project two attributes we use commas

"Which people from R1 are named 'Pater' or 'Wanda'?"	SELECT(first_name = Peter **OR** first_name = Wanda)	Here we have two "terms" and we want to return rows that satisfy *either* term.
"Which people from R2 do NOT have red as their favorite color?"	SELECT(**NOT**(favorite_color = red)) FROM R2	NOT(x = y) is often written as x !=y
"Give me a list of the people who appear on both tables, with everything I know about them."	R1 NATURAL JOIN R2	This is common sort of query! In fact, it is *so* common, people sometimes forget that the "default" join is often cross-join.
"List the secret identities of just those people that scare me AND whose favorite color is green from R2"	PROJECT secret_identity FROM (SELECT(scare_me='TRUE AND favorite_color=green) FROM R2)	Pay attention to the nesting here. We PROJECT from the result of the SELECT query.

REVIEW PROBLEMS: PRACTICE WITH RELATIONAL ALGEBRA[i]

Consider a database with the following schema:

Relations	**Keys**
Person (name, age, gender)	name is a key
Frequents (name, pizzeria)	(name, pizzeria) is a key
Eats (name, pizza)	(name, pizza) is a key
Serves (pizzeria, pizza, price)	(pizzeria, pizza) is a key

1. Write relational algebra expressions for the following nine queries. (Warning: some of the later queries are a bit challenging.)

a. Find all pizzerias frequented by at least one person under the age of 18.
b. Find the names of all females who eat either mushroom or pepperoni pizza (or both).
c. Find the names of all females who eat both mushroom and pepperoni pizza.
d. Find all pizzerias that serve at least one pizza that Amy eats for less than $10.00.
e. Find all pizzerias that are frequented by only females or only males.
f. For each person, find all pizzas the person eats that are not served by any pizzeria the person frequents. Return all such person (name) / pizza pairs.
g. Find the names of all people who frequent only pizzerias serving at least one pizza they eat.
h. Find the names of all people who frequent every pizzeria serving at least one pizza they eat.
i. Find the pizzeria serving the cheapest pepperoni pizza. In the case of ties, return all of the cheapest-pepperoni pizzerias.

If you already know SQL, you can try running SQL queries to match your relational algebra expressions. The file is available as "pizza.sql".

2. Consider a schema with two relations, R(A, B) and S(B, C), where all values are integers. Make no assumptions about keys. Consider the following three relational algebra expressions:

a. $\pi_{A,C}(R \bowtie \sigma_{B=1}S)$

b. $\pi_{A}(\sigma_{B=1}R) \times \pi_{C}(\sigma_{B=1}S)$

c. $\pi_{A,C}(\pi_{A}R \times \sigma_{B=1}S)$

Two of the three expressions are equivalent (i.e., produce the same answer on all databases), while one of them can produce a different answer. Which query can produce a different answer? Give the simplest database instance you can think of where a different answer is produced.

[i] These questions are adapted from Stanford's "Database" taught by Jennifer Wisdom in 2011.