ECE356: Database Systems Winter 2018

Lecture 4 — Relational Query Languages & SQL, Continued

Jeff Zarnett

Relational Query Languages, Continued

When we left off we had covered the operations of selection, projection and union. Not we will continue with set difference, cartesian product, and rename, followed by the shortcut operations.

Difference. The set difference operation works more or less as you would expect: it takes a relation and removes from it any tuples that are in the second relation. After taking away things we do not want, then we are left with the things we do want. Its input is therefore two relations and it produces a new relation as its output.

The mathematical symbol is $-$ (the minus or subtraction operator). The notation is then $r_1 - r_2$ where r_1 and r_2 are relations. This produces a relation r_3 that contains all tuples in r_1 that are not in r_2. It is possible to chain the subtraction operators, but like regular mathematical subtraction, it does not commute: the order matters a lot.

The query $\sigma_{make=''Honda''}(vehicle) - \sigma_{year<''2010''}(vehicle)$ produces the results:

VIN	year	make	model	license_plate_number
1GYS3BKJ5FR338462	2016	Honda	Civic	YYYY 585

Just like the union operation, the sets must be compatible. The same rules from union apply:

1. The relations must have the same number of attributes.
2. The domain of attribute i in the first relation must be the same as the domain of attribute i in the second relation, for all i.

In SQL the keyword we need for this is `EXCEPT`[1]. Like the union operation, parenthesis prevent confusion and/or make your database server less sad.

```
(SELECT make, model FROM vehicle WHERE make = 'Volkswagen')
EXCEPT
(SELECT make, model, FROM vehicle WHERE year < 2016);
```

Produces as output:

make	model
Volkswagen	Jetta

Again, you say, wait, this is silly, you could cut this down with a select query with two clauses. Yes, that is also true. Those with experience in databases may also be asking about the use of `EXCEPT` vs another construct `NOT IN`. The `EXCEPT` keyword encompasses both the `DISTINCT` and `NOT IN` behaviour, so duplicates are eliminated. My typical use of the not-in behaviour usually refers to referencing a unique key column in the database, such as:

```
SELECT make, model FROM vehicle WHERE make = 'Volkswagen' AND VIN NOT IN
(SELECT VIN FROM vehicle WHERE year < 2016);
```

This demonstrates a subquery: the first query selects a relation with one attribute.

[1] Although Oracle uses `MINUS` instead.

Cartesian Product. The cartesian product combines information from two relations. It is often the case that the data we need is in more than one relation. That might be a design problem, but good design means data is in there only once and sometimes we do need to combine relations. Suppose we wanted to send reminder notices to people whose license plates will expire next month. We need to combine license plate data with address data.

The mathematical symbol for cartesian product is $\times$, the multiplication symbol. The cartesian product $r_1 \times r_2$ forms a third relation r_3. The new relation has all the attributes of both the relations that went into it (in the order specified). An example would clarify.

Let us pretend our license plate relation contains exactly two tuples:

number	expiry	owner_address_id
ZZZZ 123	2018-09-30	24601
AAAA 855	2019-04-01	12949

And our owner address set is also two tuples:

id	name	street	city	province	postal_code
24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

Then the query $license\_plate \times owner\_address$ produces:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
ZZZZ 123	2018-09-30	24601	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
AAAA 855	2019-04-01	12949	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

This is all possibilities (and that's why we restricted these relations to 2 entries, just so we don't waste too much space). Each tuple from the first relation is paired with each tuple from the second. So if r_1 contains x tuples and r_2 contains y tuples, there will be $x * y$ tuples in the resulting relation... the vast majority of which are garbage!

Garbage, you say? The second and third tuples in the relation shown immediately above don't make any sense: plate ZZZZ 123 is associated with the owner address with ID 24601, but it looks like it's also associated with address with ID 12949. Whom does the plate belong to, Jean Valjean or Alice Jones? Well, we know it's Valjean, but the cartesian product can suggest wrong things (and produces a very big relation). Even more illogical results arise if we try to take the cartesian product of one relation with itself.

This problem did not occur with our particular example because all attribute names are distinct, but if we had two names that were the same, we would need a way to differentiate them. Suppose that two relations, one called `book` and one called `author`, each having an attribute `id`. The convention is to prefix the attribute with the name of the relation from which it came. Thus, in the cartesian product the two columns would be `book.id` and `reader.id`.

In SQL it is very simple to get the cartesian product: `SELECT * FROM license_plate, owner_address;` – that is to say we just separate the relations we wish to appear in the cartesian product with a comma in between.

We can always combine more if we needed: imagine we need to notify all current owners of Volkswagen vehicles that there is a recall: this requires us to look at both the vehicle relation and the owner address relation. To connect those we also need the license plate relation (this escalated quickly).

Typically get our data to make sense we need to restrict our query with a selection predicate. If the two relations are connected by some ID in in common, we need that. In this case, the `id` attribute in the address matches `owner_address_id` in the license plate relation. So $\sigma_{owner\_address\_id=id}(license\_plate \times owner\_address)$. Or

in SQL: `SELECT * FROM license_plate, owner_address where owner_address_id = id;` This may seem unclear so we might prefer to prefix these with their names: `SELECT * FROM license_plate, owner_address where license_plate.owner_address_id = owner_address.id;`

If we add that restriction:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

Now our data makes sense! In the future we will get to the same place through an easier operation, the join.

Rename. The rename operation is used to both change the name of existing attributes/relations and assign names to ones that have no name.

The mathematical notation is ρ (rho). If applied to a relation, it is $\rho_x(E)$ where it renames the relation to x. Or it can be applied to the name of the attributes: $\rho_{x(a_1,a_2,a_3,a_4...)}(E)$ which renames the relation to x and then the attributes a_1...

In SQL we can use the `AS` keyword. That is not super interesting on a simple query to rename a relation: `SELECT * FROM vehicle AS autos;` renames the relation result to "autos" but that does not do very much for us. It makes some more sense if we are going to use it in a subquery it will change the prefix for any duplicate attribute names (e.g., change `book.id` to `textbook.id`).

A more likely use is to rename the attributes of a relation. We can apply it to as few or as many attributes as we like: `SELECT make, model, year AS modelYear FROM vehicle;` will produce a relation with the make, model, and year, but the year attribute will be called "modelYear" as we have asked.

The motivation for the rename operation right now probably seems weak. There are some situations where we will need to use the rename operation. And I do mean need, when we do not have an AS clause on certain operations the SQL server will decline to carry out the query.

Additional Operations

Having reached the end of the basic operations, there are a few operations that we have discussed that are not fundamental: they can be derived from the six operations we have already introduced. They are, however, notationally convenient. These are from [SKS11]:

Set Intersection. Set intersection is exactly what it sounds like based on our understanding of mathematical sets. The symbol for it is $\cap$ and it is equivalent to $r_1 - (r_1 - r_2)$.

In SQL they keyword is `INTERSECT`. Its use is limited, however, in a way similar to union. If we want to use intersection on two queries of the same relation, we could just as easily set it up as a selection with a compound predicate. But a short example:

```
(SELECT name, street, city, province, postal_code FROM owner_address)
INTERSECT
(SELECT name, street, city, province, postal_code FROM employee);
```

This would produce the set of addresses that are both in the owner address relation and in the employee relation.

Assignment. The assignment operation allows us to take the result of an expression and put it in a temporary variable. The mathematical symbol for this is $\leftarrow$. This is just notational convenience for the benefit of the reader. Instead of a complicated expression that requires several sets of parenthesis and is hard to follow, we could break up the query into parts and then use those parts. So instead of something $\sigma_{owner\_address\_id=id}(license\_plate \times owner\_address)$. we could write:

$temp \leftarrow license\_plate \times owner\_address$
$\sigma_{owner\_address\_id=id}(temp)$.

This may improve (or decrease) readability if used well (or badly).

SQL does have an assignment operator: `:=`. I will, however, discourage you from using it. There are advanced situations where the use of this is appropriate, such as manipulating a counter. Eventually we will learn how to modify the data in our database and not just look at it and when we do, an assignment will take place, but we are unlikely to write it explicitly. The reason I discourage the use of assignment is the need to get away from the C-like thinking with counters and iteration; operations take place on a set of operations and we should think in a set based mindset.

Join. We have already seen the cartesian product operation tends to produce a lot of rows, some of which are illogical. By combining that with a restriction, i.e., a selection, we throw away the tuples that don't make sense and are left with only those where there is some meaning. One is hard pressed to think of a situation where this is not needed. Thus, a little notational shorthand is in order.

The mathematical symbol for the *natural join* is $\bowtie$ (cute little bowtie). It combines the cartesian product with a selection, forcing equality. A natural join combined with a selection is a theta join $\bowtie_\theta$ where θ is the selection predicate. Note that it is associative: so we can chain it as we need: $vehicle \bowtie license\_plate \bowtie owner\_address$ is equivalent to $(vehicle \bowtie license\_plate) \bowtie owner\_address$ and $vehicle \bowtie (license\_plate \bowtie owner\_address)$ [SKS11].

Except, this probably doesn't work the way we hope on our sample data because the fields are not named the same thing. If both fields are called `id`, the database server will try to match on those. But this is not going to work for our data since they all have different names.

In SQL the keyword we need for this is `NATURAL JOIN`. So we would do: `SELECT * FROM vehicle NATURAL JOIN license_plate;` Again, though, this is not going to cooperate for us because our names don't match.

What do we do then? We need to expand our concept of the join operation. The default type is the `INNER JOIN` and it's what you get if you just write `JOIN` instead of being specific. It requires us to use the `ON` clause to specify how we relate one side to the other.

`SELECT * FROM license_plate JOIN owner_address ON owner_address_id = id;` produces:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

My general recommendation is always to use the `ON` clause rather than `NATURAL JOIN`. Oftentimes you need it and when you don't you sometimes get the wrong behaviour because a natural join takes the unique ID field from each relation and not the field that actually links them. If the license plate relation called the plate number "id" instead of "number", the natural join operation would run but produce no results. Be explicit about how the join should work.

If there is an inner join, there is obviously an outer join. Actually, there are three of them! They are, however, all very similar.

Suppose we have a license plate that does not correspond to any address:

number	expiry	owner_address_id
ZZZZ 123	2018-09-30	24601
AAAA 855	2019-04-01	12949
BBCC 394	2019-02-12	null

and an address that does not correspond to any license plate:

id	name	street	city	province	postal_code
24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
86753	Sherlock Holmes	221B Baker St	London	ON	D4D 4D4

The inner join of license plate with owner address would look like this:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

... that is to say, no different from how it was before. Because there is no match for license plate "BBCC 394" it does not appear in the output relation when the natural join is used. The same applies to address with ID "86753"; it has no match in the license plate relation.

If we use the left outer join, which has the symbol ⟕, we get a tuple in the result for each entry in the left (first) table, even if it has no match in the second. For such a tuple, all attributes from the right (second) table will simply be "null". If a tuple in the right table has no match in the left table, it does not appear at all. The SQL for this is, of course, `LEFT OUTER JOIN`. So `SELECT * FROM license_plate LEFT OUTER JOIN owner_address ON owner_address_id = id;` returns:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
BBCC 394	2019-02-12	null	null	null	null	null	null	null

The right outer join, symbol, ⟖ (`RIGHT OUTER JOIN`) is the mirror image operation of the left outer join. We get a tuple in the result for each entry in the right (second) table, even if it has no match in the first. For such a tuple, all attributes from the left (first) table will simply be "null". If a tuple in the left table has no match in the right table, it does not appear at all.`SELECT * FROM license_plate RIGHT OUTER JOIN owner_address ON owner_address_id = id;` returns:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
null	null	null	86753	Sherlock Holmes	221B Baker St	London	ON	D4D 4D4

In both the left and right outer join, it is important to note the order in which the relations appear. The operation thinks of the left and right side from the perspective of the placement of the operator (as one would expect).

Finally, there is the full outer join, with the symbol ⟗ does both the left and right outer join operations, including tuples from each relation that do not match the other:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
BBCC 394	2019-02-12	null	null	null	null	null	null	null
null	null	null	86753	Sherlock Holmes	221B Baker St	London	ON	D4D 4D4

It might be tempting to combine tuples from one relation with those from another where there are suitable null attributes, but this is misleading: the license plate BBCC 394 is NOT associated with the address of Sherlock Holmes and it does not make sense to combine these.

References

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.