ECE356: Database Systems Winter 2018

Lecture 3 — Relational Query Languages and SQL

Jeff Zarnett

Relational Query Languages

A *query language* is a language in which the user requests information from the database. Such languages can be procedural, which is to say that you tell the database exactly how to get the data you want, or non-procedural, in which case you just tell the database what you want and it figures out how to deliver that result[1].

The operations we want to discuss can be applied to a single relation, or a pair of relations; the result of is always a single relation [SKS11]. This means that the output of one operation can be used as the input to another operation, allowing us to chain operations as needed.

If it helps you to imagine this, every operation is a function with return type `Relation` and they take either one or two `Relations` as parameters. You will also need to keep in mind that because they all have input and output of relation(s), the operations operate on a set of data, not just a single element. In a C-like language you might write a for loop that sets all elements in an array to be `-1`. In SQL you would write a query that says set the value to be `-1` for all entries in the relation. This means that if your normal mode of thinking is C-like (procedural) then there is a mental transition that needs to be made.

The fundamental operations in relational algebra are [SKS11]:

- Select
- Project
- Union
- Set Difference
- Cartesian Product
- Rename

And there are three shortcuts we will use that can be created using the fundamental operations:

- Set Intersection
- Assignment
- Join

SQL – "Structured Query Language" – is not only for querying but also for defining and changing the database schema. It is a nonprocedural language: you ask for what you want and the database server returns a result that matches that format. What we will discuss in this class is SQL (and more specifically you will use a free implementation, MySQL, in the labs). As we go through the operations we will learn how to express the operation we want to do using SQL as a specific example. But other query languages do exist.

The course notes and the way the material is examined reflects the reality that the majority of students taking this course already have exposure to some form of database system, most likely a SQL relational database of some sort. That may mean the material is fully review for the reader. Still, the mathematical notation may be new.

[1] There is of course the possibility that the database server chooses a SUPER INEFFICIENT way of carrying out the query, as we will discuss in the future...

SQL consists of several parts [SKS11]:

- Data Definition Language (DDL) – Used to specify the schema, define domains, relations, integrity constraints.
- Data Manipulation Language (DML) – Used to query (read) as well as manipulate the data.
- Data Control Language (DCL) – Used to control access privileges in the database.
- Transaction Control Language (TCL) – Used to control the execution of transactions.

For now we will focus on the data manipulation language, although we will talk about the data definition language.

Let's use the following sample data to understand the operations.

VIN	**year**	**make**	**model**	**license_plate_number**
2B4FH25K1RR646348	2005	Honda	Civic	ZZZZ 249
4UZACLBW87CZ42980	2011	Ford	Focus	YYYY 995
WVWDM7AJ4BW227648	2015	Volkswagen	Golf	ZRHC 112
1J4FT38L7KL506678	2017	Audi	A4	VNTY PLT
2C8GF48415R447850	2016	Volkswagen	Jetta	VWJG 071
2T3RK4DV1AW089914	2014	Toyota	Camry	ZZZZ 385
1XPCDR9X2YN436206	2012	Toyota	Camry	ZZYY 251
1GYS3BKJ5FR338462	2016	Honda	Civic	YYYY 585
1GAGG25R6Y1243081	2015	Chevrolet	Corvette	GORLFAST
JH2SC59208M054959	2018	Genesis	Genesis	AAAA 123

Selection. The first operation we will discuss is selection. That is to say, find the tuples in the relation. As you might imagine, we need to specify what we want to find, and where we want to look for it. Also, we can specify that we want to find only the things that match certain criteria.

In mathematical notation, selection has the symbol σ (sigma). The predicate is put in a subscript and then the function parameter, the relation to select from, is follows in parenthesis. $\sigma_{make=''Volkswagen''}(vehicle)$ selects the tuples from the `vehicle` relation where the make is equal to "Volkswagen". That produces a new relation that looks like this:

VIN	**year**	**make**	**model**	**license_plate_number**
WVWDM7AJ4BW227648	2015	Volkswagen	Golf	ZRHC 112
2C8GF48415R447850	2016	Volkswagen	Jetta	VWJG 071

We specify the predicate p using propositional logic. A term is written as an attribute followed by a comparison operator then an attribute or constant. The comparison operators are $=, \neq, >, \geq, <, \leq$. So the attribute in the example is $make$ and the comparison operator is $=$ (equals) and then the right hand side of that equation is the constant "Volkswagen". Terms are connected by the mathematical AND $\wedge$, mathematical OR $\vee$, and mathematical NOT $\neg$ operators. Thus we could specify make = "Volkswagen" and year = "2015" ($\sigma_{make=''Volkswagen''\wedge year=''2015''}(vehicle)$) which would restrict the returned set of tuples to be those that match both parts of the predicate:

VIN	**year**	**make**	**model**	**license_plate_number**
WVWDM7AJ4BW227648	2015	Volkswagen	Golf	ZRHC 112

We can chain as many of these qualifiers as desired. Use of parenthesis is recommended to make it clear how the boolean logic is to be evaluated. If you wish the year to be either 2015 or 2016 and the make to be Volkswagen, then it is advisable to write that as $make ='' Volkswagen'' \wedge (year =''\ 2015'' \vee year =''\ 2016'')$. My general rule

is that it doesn't cost extra money to write the parenthesis so it is advisable to use them anywhere there is the possibility of error or confusion.

It is possible, of course, that no tuples in the relation match all of the predicates. If the predicate is make = "Chrysler" then no matches will be found in our sample data. Or, if we specify make = "Honda", and model = "Camry", we will still find no tuples that match that either. Honda doesn't manufacture any model named "Camry" so we expect not to find any such rows in the database, but Chrysler does make cars (still), so that data is plausible, it just so happens that we didn't find any in our sample data set. Either way we get back an empty relation.

In SQL, the `SELECT` clause is used to, well, select tuples from the database. The SQL command that corresponds to $\sigma(vehicle)$ is `SELECT * FROM VEHICLE;` Some observations spring from this immediately. For one thing, it is traditional in SQL that we write keywords in all capitals. Many databases don't really seem to mind if you use lowercase or mixed case, but some systems are case sensitive. Another item of note is that a statement is terminated by a semicolon (which is nothing new for those of us who write C-like languages). We also note the use of `FROM` to specify the relation. The asterisk (`*`) is needed to specify what we are going to select, as subject we will return to soon when we talk about projection.

The select statement as is contains no predicates so it would return all tuples in the relation. If we wished to add the predicate that make equals Volkswagen, the statement becomes `SELECT * FROM VEHICLE WHERE make = 'Volkswagen';` Some more observations then: the `WHERE` keyword indicates the start of the predicate. The string literal "Volkswagen" is enclosed in single quotation marks but it can also be in double quotation marks (although some database systems may object to this and insist on one or the other). To form a compound predicate: `SELECT * FROM VEHICLE WHERE make = 'Volkswagen' AND year = '2015';`.

The comparison operators are what you might expect for a programming language: `<`, `>`, `>=`, `<=` are all as expected in a C-like language. Equals is just a single equals sign (`=`) and the not equals operator is written `<>`, which although it looks really strange if you are a C programmer, is the correct not equals operator in BASIC. These operations can work on strings, mathematical types, dates, et cetera. You may, however, get unexpected results if you attempt to compare two types that don't compare well.

Although it is possible to chain the where clause to include a range, such as `WHERE year >= 2010 AND year <= 2015` there is a bit of notational convenience in SQL: the `BETWEEN ... AND` syntax. Thus one can write `WHERE year BETWEEN 2010 AND 2015`. This is inclusive on both ends, so be careful about whether this is the desired behaviour.

The predicate may also contain the name of other attributes in the relation, such as `SELECT * FROM VEHICLE WHERE make = model;`. This will return the one tuple in the sample data where make and model are the same. There are situations in real life where we want use an attribute rather than a constant, although in this particular example it may look a little strange.

Projection. The second operation is projection. This takes a subset of a relation: take a relation and extract from it only the things that we asked for. Projection cuts down the relation to the specific attributes. Projection takes a single relation and returns another relation.

As you can imagine, this may help the user or application program: if we want to know what the VIN is for the car with the license plate "GORLFAST" we can ask for just that and need not get back the entire tuple. Similarly, if you want to get a list of the makes and models in the database, again, it's nice to be able to get this data without any of the parts you do not need. Projection also makes certain things a lot faster: it makes no sense to load all this extra data to your program if you are going to throw away almost all the columns[2].

In mathematical notation, projection has the symbol Π (capital pi). Like selection, it takes a subscript and the input relation follows in parenthesis. So, $\Pi_{make,model}(vehicle)$ will reduce the relation down to just the attributes make and model, leaving out VIN, year, and license plate number information. All tuples in the relation will appear in the output of the projection operation, although it is entirely possible that some of them look very much alike. See the example data below for when we project the data:

[2]At some point I will end up telling this story about how the SQL query optimizer is not very smart... Yes, this references the previous footnote...

make	model
Honda	Civic
Ford	Focus
Volkswagen	Golf
Audi	A4
Volkswagen	Jetta
Toyota	Camry
Chevrolet	Corvette
Genesis	Genesis

There are fewer tuples in this relation than there were in the original relation. This is because there are duplicates ("Honda"/"Civic", for example). In the mathematical world a relation is a set and therefore duplicates are not permitted. The order of the tuples is not specified either. The attributes will be listed in the order they are provided in the project clause, which we may choose arbitrarily. If there is a reason to have them in a certain order, then put them in that order. You can also ask for all attributes in the relation; this may simply re-order the attributes (columns) if you ask for all of them.

In SQL projection is done through our application of the `SELECT` clause. In the previous examples we used `*` which specified all attributes. In production the use of `SELECT *` is discouraged; usually the correct approach is to ask specifically for the things you want and discard all the things you don't want. So, the SQL equivalent of the previous mathematical operation is: `SELECT make, model FROM VEHICLE;` which returns:

make	model
Honda	Civic
Ford	Focus
Volkswagen	Golf
Audi	A4
Volkswagen	Jetta
Toyota	Camry
Toyota	Camry
Honda	Civic
Chevrolet	Corvette
Genesis	Genesis

This does not look the same as the previous output. What gives? Detection of duplicates is relatively difficult (computationally expensive). SQL does not remove them unless you ask, so technically what you get back is more akin to a list than to a set. You could also consider it a multiset. Nevertheless, if you wish to remove duplicates you may use the keyword `DISTINCT`: `SELECT DISTINCT make, model FROM VEHICLE;` which then eliminates duplicates from the results. The opposite of `DISTINCT` is the `ALL` keyword, but since this is the default behaviour you are unlikely to need to write it.

The SQL `SELECT` statement is then both selection and projection. In what order do these operations take place? Sometimes it does not matter, e.g., if we asked for `SELECT *...` then the projection does nothing. Here's a hint: this statement is perfectly valid: `SELECT make, model from VEHICLE WHERE year = '2016';` and returns:

make	model
Volkswagen	Jetta
Honda	Civic

Clearly the selection operation must take place first: if we cut the attributes down to just make and model, then we lose the information we need to tell whether the year is 2016 or not. Admittedly, when we get into the topic about how queries are carried out, we may find it is better to cut down the result relation with projection before applying the selection, then reducing it further. But let us not skip ahead too much.

The SELECT statement in SQL lets us do some interesting things other than select attributes alone. The select clause may contain arithmetic expressions using the operators $+, -, *, /$ operating on either attributes or tuples [SKS11]. Thus, to get a 2 digit date out of a 4 digit date we could do: `SELECT year - 2000 FROM VEHICLE;`. Assuming, of course, that the types are compatible. The utility of using the mathematical functions is perhaps somewhat limited for now, but there will be occasions in the future.

We can also select constants if we wish, by writing that constant instead of the name of the attribute. `SELECT 'Car', make, model FROM vehicle WHERE year = '2016';` produces:

Car	make	model
Car	Volkswagen	Jetta
Car	Honda	Civic

That maybe looks silly, asking for a constant value to be returned in a relation. There are some uses for the selection of a constant, as we will see shortly.

Union. We can combine two relations using the union operation. It takes two relations and produces a single relation from them.

The mathematical symbol is $\cup$, the same as the mathematical set union operator. Typically this is used in conjunction with other operations (selection, projection) to get the data that you want. The query $\sigma_{make=''Ford''}(vehicle) \cup \sigma_{make=''Audi''}(vehicle)$ produces the results:

VIN	year	make	model	license_plate_number
4UZACLBW87CZ42980	2011	Ford	Focus	YYYY 995
1J4FT38L7KL506678	2017	Audi	A4	VNTY PLT

The first selection statement returns a relation that contains one tuple. The second selection statement also returns a relation that contains one tuple. The union operator then takes those results and combines them into a single relation that contains both tuples. If we had asked for a projection we could either apply it to each selection operation, or to the result of the union.

In SQL the keyword is `UNION`. Use of parenthesis is recommended (if not mandatory) to help the SQL query parser figure out what it is you are asking for.
```
(SELECT license_plate_number FROM vehicle WHERE make = 'Ford')
UNION (SELECT license_plate_number FROM vehicle WHERE make = 'Audi');
```

But wait, you say, this is silly. Why did you not simply write $\sigma_{make=''Ford'' \vee make=''Audi''}(vehicle)$ (or its SQL equivalent)? It would produce the exact same results. And so indeed it would. The use of the union operation when selecting from a single relation is not wrong, but not necessary in this situation since we could use OR.

Where we will more often see union applied is when we want to combine data from two different relations. Suppose our database also has a relation (table) for the addresses of employees that has employee ID, name, street, city, province, and postal code attributes. Thus to get all the addresses in the system, the query in math notation is:

$(\Pi_{name,street,city,province,postal\_code}(\sigma(owner\_address))) \cup (\Pi_{name,street,city,province,postal\_code}(\sigma(employee)))$

In SQL:

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

The union operation can only succeed if the types are compatible. For the union operation to make sense, two conditions must hold [SKS11]:

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.

The previous example meets the criteria because we specified the attributes in the same order and they are presumed to be the same type, or at least, compatible domains.

The names do not have to be the same: if the employee relation called it "postcode" instead of "postal_code" then the union operation could succeed as long as they are both defined as the same type (e.g., string of length 7). Your SQL server might let you get away with things though: it might allow you to take the union of a license plate set and a postal code set because the domain of each of those is string. This makes no logical sense, but the database may let you do it. In some cases it will let you combine otherwise incompatible types if it can figure out a way to make it work... such as converting an integer to a string... but it is not recommended!

This example also is a situation where we might make use of a selection with a constant. If we wanted to produce a relation with all addresses, and wish to include some type information, we could do this:

```
(SELECT 'Vehicle Owner', name, street, city, province, postal_code FROM owner_address)
UNION
(SELECT 'MTO Employee', name, street, city, province, postal_code FROM employee);
```

	name	**street**	**city**	**province**	**postal_code**
Vehicle Owner	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
Vehicle Owner	Thomas Anderson	1234 Main St	Waterloo	ON	A0A 0A0
Vehicle Owner	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
MTO Employee	Kim Morgan	491 Example Road	Waterloo	ON	N2N 2N2
MTO Employee	Jordan Singh	24 Eastern Avenue	London	ON	N6A 0B0

This combined result made use of projection, selection, and union.

References

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