ECE356: Database Systems Winter 2018

Lecture 5 — More SQL

Jeff Zarnett

SQL: the Sequel

With some theoretical understanding of the relational algebra that makes up the basis for SQL we can now take some time to learn a little more of the practical ins and outs of the language that will allow us to do the things we need our database to do.

String Operations. String operations are complicated in any language and SQL is no exception. We have used some string matching already so we have a pretty good idea how it works: by default a string is enclosed in single-quote characters such as `'AAAA 111'`. Double quotes can also be used, especially if you need to enclose a single-quote literal in your string such as: `"Burk's Falls"`.

The SQL standard says that string comparison with = is case sensitive, but MySQL (and some others) do not respect this and perform case-insensitive comparison [SKS11]. Standards are optional, I guess? MySQL has gotten better and made this a configurable parameter, but it's still something to be wary of.

Some functions familiar from C-like languages exist: `UPPER()`, `LOWER()`, `TRIM()`, et cetera. Pattern matching can be done using `LIKE` which takes two special characters as wildcards: `%` (match any substring) and `_` (match any single character) [SKS11]. If you need those characters explicitly, they are escaped using the `\` as one might expect. The backslash itself can also be escaped using the backslash.

Some examples might clarify how it works: the clause `LIKE 'Marc%'` would match the strings "Marc", "Marco", "Marcelline", and so on. The clause `LIKE 'Marc_'` would match only "Marco" from that previous set (exactly one character is required).

Null Values. If you have programmed with Java or C# or similar then you probably know that the existence of `null` sometimes complicates your life (`NullPointerException` anyone?). A null attribute indicates a lack of a value and it causes a certain chaos with some of our operations.

Arithmetic expressions involving null (e.g., addition, subtraction, etc) always results in null if any one of the operands is null. Comparisons involving null are also a hassle; saying whether 1 is "less than" null is not sensible so the value will be "unknown" [SKS11].

Use of "unknown" values in boolean operators is also complicated [SKS11]:

- AND: Comparing true AND unknown results in unknown; false AND unknown results in false; unknown AND unknown results in unknown.
- OR: Comparing true OR unknown results in true; false OR unknown results in unknown; unknown OR unknown results in unknown.
- NOT unknown also results in unknown.

If a where clause for a particular tuple evaluates to false, it is not added to the tuples to be returned or changed.

It is possible, though, to test whether something is null, or is not null. As you might expect, those are `IS NULL` and `IS NOT NULL`. Example: `SELECT * from VEHICLE WHERE year IS NULL;` finds all tuples in the vehicle relation where the year attribute is null. It is correct to use the `IS NULL` or `IS NOT NULL` constructions rather than something like `= null` or `<> null`.

Ordering. Thus far if multiple tuples are returned by a query, we get them in no specified order. You may see consistent behaviour across repeated queries, but no promises are made about the order in which they are delivered. We can change that, if we want, through the use of an `ORDER BY` clause. We specify the attribute to sort by, and we can choose ascending (`ASC`) or descending (`DESC`). An example: `SELECT * FROM VEHICLE ORDER BY make ASC`.

We may specify multiple attributes in the `ORDER BY` clause, such as `ORDER BY make DESC, model DESC`. This sorts by the "make" attribute (descending), and if the values of two tuples are equal in that attribute, then by "model" (descending). If no two tuples have the same first sort attribute then the second sort does nothing (why should it?). And you can, if desired, choose divergent directions: mixing ascending and descending. They are always handled from left to right so they cannot conflict.

The field used to order the tuples does not have to be returned in in the result set you have selected. The order will still be imposed but it may not be obvious from looking at the result relation what the ordering actually is.

Aggregate Functions. Aggregate functions perform a reduction on the data. Reduction is something you've done a lot of, most likely, without knowing it: if you are given an array of integers and asked to sum it up, that's a reduction. You condense an array of values into a single one. That's a subject that gets some more attention in ECE 459: Programming for Performance. But the database can do some of these things for us!

The aggregate functions in SQL mostly are [SKS11]:

1. `AVG`: Average
2. `MAX`: Maximum
3. `MIN`: Minimum
4. `SUM`: Sum (total)
5. `COUNT`: Count (obviously)

These work more or less like you would expect: you can `SELECT SUM( salary ) FROM employee;` to get the total salary of all employees, or `SELECT AVG( salary ) FROM employee;` to get the average salary of all employees. Maximum and minimum are also pretty self-explanatory. Sum and average work only on numbers, but non-numeric types can be used for maximum and minimum.

`SELECT COUNT( salary ) FROM employee;` tells you the number of entries in the table where there is a salary. More commonly you would see `SELECT COUNT(*)` to count the number of entries in the relation. You could also put the primary key in the parenthesis instead of the asterisk if the asterisk gives you worries about performance nightmares.

One can also put `DISTINCT` in the count expression. If you wished to count the number of distinct model names, you could write `SELECT COUNT( DISTINCT make ) FROM vehicle;` which would remove duplicates.

According to [SKS11] SQL does not permit combination of `DISTINCT` and `COUNT(*)`. Although it seems like this is accepted by MySQL, it is nonstandard behaviour if it does work.

It is nice to be able to sum up or average over all tuples matching some clause, but we can get grouping if we want as well. It might be interesting to know there are 150 different models of car, and if we are curious we can add a where clause to find out how many are made by Audi, and then another one where we find out how many are made by BMW... But that is inefficient. The solution is grouping:

Example: `SELECT make, COUNT(model) FROM vehicle GROUP BY make;` might produce a result like the table shown below:

make	count(model)
Honda	1
Ford	1
Volkswagen	2
Totoya	1
Chevrolet	1
Genesis	1

One may of course use the `AS` keyword to rename the count attribute if desired or restrict the query with a where clause. We could group by more than one criterion, such as grouping by both make and model, in which case we would could get a count of the number of Honda Civics on the database, for example. Grouping works as you would generally expect.

The select statement, obviously, selects over some attributes of the relation and they must be either (1) in the group by clause or (2) aggregated in some way. The statement `SELECT make, COUNT(model), year FROM vehicle GROUP BY make;` is invalid because the attribute "year" is not in the group by clause nor is it aggregated. This is sensible: there are multiple values for the "year" attribute and we would have no way of knowing which is the correct value to output here.

That's nice, but what if we wanted to return only those rows where a certain property holds, such as count is greater than 1? For that there is the `HAVING` clause which would be applied as follows: `SELECT make, COUNT(model) FROM vehicle GROUP BY make HAVING COUNT(model) > 1;` This would return only the one tuple (Volkswagen).

The textbook [SKS11] gives the following evaluation order for such a query:

1. The `FROM` clause is evaluated to get the relation.
2. The `WHERE` clauses is evaluated on the tuples in the relation specified in the `FROM` clause.
3. Tuples matching the `WHERE` clause are placed into groups according to the `GROUP BY` clause (if this is absent, then it's all one big group).
4. The `HAVING` clause, if any, is evaluated and removes groups that don't meet the having condition.
5. The `SELECT` operation is performed, producing the tuple for each group.

In general, aggregate operations on null values tend to ignore any nulls they encounter: if the SQL query sums salaries and there is a null value it will not affect the sum (i.e. it will be treated like zero). The `COUNT` function does not, however. As usual, nulls have the potential to cause chaos.

Subqueries & Set Comparison. We can have subqueries (nested queries) where the result of one query is used as input to another query in a single statement. The keyword we are going to use extensively here is `IN`.

The first use of a set comparison we might think of involves replacing a query that looks like this: `SELECT * FROM employee WHERE id = 2419 OR id = 2917 OR id = 4058 OR id = 5911;` with: `SELECT * FROM employee WHERE id IN (2419, 2917, 4058, 5911);`. This is more compact and easier to read; in a simple example like this there is not much possibility of confusion of the OR clauses, but in a query with multiple parts joined by AND and OR clauses, parenthesis are soon needed to avoid confusion.

Instead of an explicit list, the list could be generated by a query. Suppose the ministry of transport has discovered that license plates in the range CCCC 001 through CCCC 999 were manufactured incorrectly and are prone to falling apart easily. The ministry can send an e-mail to everyone affected to tell them of the problem and offer a free replacement. `SELECT name, street, city, province, postal_code FROM owner_address WHERE id IN (SELECT owner_address_id FROM license_plate WHERE number LIKE 'CCCC%');`

Let us pretend that an auto manufacturer has a recall, say, they faked the emissions tests data. The ministry of transport could send letters to everyone who owns such a vehicle to let them know that if they do not get the

dealer repair they will be unable to renew their license plates. In that case we need to have three tables linked and we can do that with subqueries.

```
SELECT name, street, city, province, postal_code
FROM owner_address WHERE id IN
  (SELECT id FROM license_plate WHERE number IN
    (SELECT license_plate_number FROM vehicle WHERE make = 'Volkswagen' AND model = 'Golf')
  );
```

We can usually replace a subquery with a join query, and vice versa. In many cases it is a question of taste, but sometimes a subquery is easier to read and understand (especially if it is complex).

Other Things. This has by no means been an exhaustive list of all the SQL keywords one could possibly use (e.g., `UNIQUE`, `WITH`, et cetera). We may introduce additional syntax later if it is relevant to some other operation. For the moment, however, we have enough syntax covered to continue on to modification as well as design.

Modification

Until now all the operations have been of the look-but-don't-touch variety – we have looked at the data, combined it, sliced and diced it, and generated some temporary values – but that's it; we haven't actually changed the data in any permanent way. There are three basic operations we can do: add, remove, and change.

Keep in mind that modification operations are transactions (that is to say, they are supposed to succeed or be as if it never started). So what really happens is that a new relation is created and that replaces the original relation. These operations can also be expressed using relational algebra with an assignment.

Insert. The insert statement creates a new tuple and adds it to the relation, or creates a set of tuples and then adds that set to the relation.

If we are creating just one, then the simple, manual approach is sufficient. In SQL the keywords are `INSERT INTO` and `VALUES`. To add a new license plate we would write: `INSERT INTO license_plate( number, expiry, owner_address_id ) VALUES( 'HOLMES01', '2018-12-10', 86753 );`

Observations: we specify the relation and then we give a list of attributes that we want to assign, followed by the values that we wish to be assigned. In some situations we can leave off the parenthesis-enclosed list of attributes but that is not recommended. The SQL server will try to carry out the assignment of values in the order that the attributes are defined in the database and this may result in undesired behaviour. I therefore insist that the use of the order of attributes is important and you should use it everywhere; doing less is just bad practice.

Furthermore, we can put the attributes to be inserted in any order as long as the value types match: `INSERT INTO license_plate( expiry, number, owner_address_id ) VALUES( '2018-12-10', 'HOLMES01', 86753 );` is completely equivalent to the previous statement.

It is permissible to leave some attributes out of the insert statement. `INSERT INTO license_plate( expiry, number ) VALUES( '2018-12-10', 'HOLMES01' );` would create a tuple where the owner address id attribute contains null. This must, however, be permitted by the table definition: i.e., the field must allow null attributes or have a defined default (a boolean attribute, for example, may have "default false" as part of its definition).

We may also insert multiple elements into a relation based on the result of a select statement. `INSERT INTO owner_address ( SELECT * FROM employee_address ) );` is a minimal statement that takes the result of the selection and uses it as a set of tuples to insert. This statement leaves off the attribute specification (which is possible, but not recommended) and has no predicate (where clauses) so it is a rather dangerous statement to write. These sorts of statements are hopefully rare occurrences in your database because they have a risk of going wrong or duplicating data. But sometimes, when, for example, two tables are being merged, they are unavoidable.

In relational algebra, if the relation is r and the new tuples are t, then the insert statement is $r \leftarrow (r \cup t)$.

Delete. The delete operation can only be used to remove a set of tuples from a single relation; it does not alter attributes in the tuples [SKS11]. If the goal is to "clear fields" the update statement is appropriate. A delete statement cannot be used on multiple relations at once.

The keyword in SQL is `DELETE`[1] and the statement `DELETE FROM license_plate where number = 'BBCC 394';` will remove from the license plate relation any and all tuples that match the predicate. If no predicate is specified, then ALL tuples in the relation are removed (yikes!).

Like an insertion we may use the result of a subquery as the predicate in a where clause, so we could delete all license plates where the registered address province is not Ontario with something like `DELETE FROM license_plate WHERE owner_address_id in (SELECT id FROM OWNER_ADDRESS WHERE province <> 'ON');`

The textbook [SKS11] has an example about deletion where the salary is less than the average. This highlights that the deletion operation first figures out the average, then decides what tuples to delete, before carrying out the operation. If this were not the case, after each deletion the average might change and we might delete all tuples (or the result would otherwise be strange). If it carried out the deletion in such a manner it would also break the rule we have about this being a transaction: the tuples to be deleted should be removed in one swift stroke with no partially completed state visible.

In relational algebra then the relation r is modified by a deletion with predicate p as follows: $r \leftarrow (r - \sigma_p(r))$.

Deletion should be used very carefully. When data is removed from the database, it's gone and not coming back (except, well, from backups... You do take backups, right?). Sometimes rather than actually deleting things you may be well advised to "deactivate" them, whether by having a boolean attribute for "deactivated" or having a second relation as a sort of "recycle bin"/"trash can" where tuples to be removed will eventually go.

Update. The update statement changes one or more attributes of the tuple without changing all the values in the tuple. The update statement, like the delete operation, operates on a set of tuples: one may specify the predicate with the `WHERE` clause just as before.

In SQL the keywords we need are `UPDATE` and `SET`. To update someone's address: `UPDATE OWNER_ADDRESS SET postal_code = 'B1B 2B2' WHERE id = '24601';`.

By now there should be nothing unexpected in this statement: we specify what fields to assign, their new values, and we limit the tuples to be affected by use of the `WHERE` clause and its predicate.

It is necessary to specify at least one attribute to change, although it is possible that no tuples are changed because none match the where-predicate. Similarly, an update may fail if we try to violate one of the rules: if the new text is too long for the attribute, for example. There are also some rules about updating an identifier for this relation.

In [SKS11] there is an example involving salaries where the statement runs along the following lines: `UPDATE employee SET salary = salary * 1.05 WHERE id = 1;`. This takes the current value of salary for the employee with id "1" and increases it by 5% (the CEO is ever so generous to the CEO, no?). In this case the attribute salary is both read and written. It is perfectly alright to use and update a value like this, in much the same way that it is permissible to write a statement like `x = x + 5;` in a C-like language; the assignment takes place after the right hand side expression has been evaluated.

Similarly, if the statement is `UPDATE employee SET salary = salary * 1.05 WHERE salary > 100000;`, the behaviour is like deletion in that the set of salaries to update is determined before any modifications begin. The whole statement, no matter how many tuples it affects, is viewed as a transaction.

We could model this in relational calculus as simultaneous deletion of the old tuple and insertion of the new one.

[1] `https://www.youtube.com/watch?v=4ecWDo-HpbE`

References

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