CS2 Current Technologies note 8

Querying Relational Data: SQL

Peter Buneman

8.1 Introduction

Let's start by observing some points about SQL.

- The original proposal for SQL was called SEQUEL, for Structured English QUEry Language. In fact it syntactically close to what mathematicians have called *Zermelo Frankel* notation and to what in programming languages are known as *comprehensions*. An example of the former is $\{(x, y) \mid \exists z.x^2 + y^2 = z^2 \wedge z < 30\}$ However, this specification is a bit loose: are x, y, z reals or integers; are we supposed to consider only positive numbers or arbitrary numbers. In programming language comprehensions, we are specific about what our variables range over so, for example, you can write in Python:

  ```
  [(x,y) for x in range(1,30)
             for y in range(1,30)
                 for z in range(1,30)
                 if x*x + y*y == z*z]
  ```

  The term `range(1,30)` generates the *list* of integers from 1 to 30. In some other programming languages you can write things like

  ```
  [{name:  s.name, cname:  t.cname} | s IN students, t IN takes,
                                     s.id=t.id AND t.marks > 65]
  ```

  in which the term {`name: s.name, cname: t.cname`} creates a single tuple with `name` and `cname` fields.

  It should be clear that these have a close syntactic similarity both to SQL and to XQuery. However in SQL we don't have variables, instead we can use aliases which look similar but mean something different. More on this when we study relational algebra.

- For querying and updating the database, SQL is typically the *only* interface you have. There is no accessible lower-level interface.

- One of the reasons for not allowing access to some lower-level interface is that SQL is highly *optimised*. For example, the database "comprehension" above looks as though it should be implemented with a double loop, but SQL can easily translate this into a much faster program.

- SQL, although the original syntactic idea was very clean, has a lot of bells and whistles. We shall only deal with basic, but very important, subset of SQL.

8.2 Putting SQL to work

It would be very easy to translate our tabular data back into XML and use XQuery in order to get information out of it and – if we wanted. Instead we shall study SQL. Again, it is hearsay that SQL is the most widely used programming language. If it is true, it is a "cheap shot" because most SQL is generated by other programs. When you order a book on the internet, the likelihood is that you are using a language like PHP or Python to generate SQL on the fly. Anyway, if you end up having anything to do with databases, the chances are that you will either have to write SQL or write programs that generate SQL, so we need to understand how it works (and later on how to make it work efficiently.)

On the face of it SQL is "rearranged" XQuery. Perhaps it would be better to say that XQuery, which is partly based on SQL, is rearranged SQL. The basic form of SQL is

```
SELECT  fields
FROM    tables
WHERE   condition
```

Let's look at some simple queries that involve only one table

```
SELECT  id, name
FROM    students
```

This just produces two columns of the `students` table. The `WHERE` part is missing. This is a projection $\pi_{\texttt{id,name}}$(students) in relational algebra.

```
SELECT  *
FROM    courses
WHERE   room = "C21"
```

This produces a table with the same structure as the `courses`. It shows only those tuples for which the `room` is `"C21"`. The "`*`" means select the whole tuple (all the fields). This is simply a selection $\sigma_{\texttt{room=C21}}$(courses) in relational algebra.

```
SELECT  id, marks
FROM    takes
WHERE   marks > 70 AND cname = "CompSci3"
```

You should be getting the idea that the `SELECT` clause determines the columns that are produced and the `WHERE` clause determines the rows. But there's a limit to what you can do with one table. The more interesting operation happens when you combine tables – recall natural join from relational algebra.

```
SELECT  students.name, takes.cname
FROM    students, takes
WHERE   students.id = takes.id AND takes.marks > 65
```

In this query we have used "fully qualified" field names of the form *table-name.attribute-name*.

We need them to distinguish between the `id` fields of the `students` and `takes` tables. SQL will allow you to drop the table name if there is no ambiguity. This is what was happening in our simple single-table queries, and we could have written the query above as:

```
SELECT  name, cname
FROM    students, takes
WHERE   students.id = takes.id AND marks > 65
```

Tables involving several joins are common:

```
SELECT  name
FROM    students, takes, courses
WHERE   students.id = takes.id AND takes.cname = courses.cname
        AND (teacher = "Sneezy" OR teacher = "Grumpy")
```

The naive evaluation method for such a query is to have three nested for-loops – one for each table in the `FROM` clause – and whenever the condition is met, emit the requested fields. Now assume 10,000 students, 1000 courses and that each student currently at the university has taken 20 courses on average, so that the `takes` table contains 200,000 tuples. These are reasonable figures for some universities which have shorter courses. This means that the number of iterations is $10,000 \times 200,000 \times 1,000 = 2 \times 10^{12}$! The surprising thing is that SQL can *optimise* this query so that it runs as fast as the same query written in XQuery on either of the two representations of the data. You'll discover how it does this in more advanced database courses.

Sometimes we need to use the same table twice in a query. Suppose we want to find all pairs of teachers who teach in the same room:

```
SELECT  C1.cname, C2.cname
FROM    courses C1, courses C2
WHERE   C1.room = C2.room
```

`C1` and `C2` are declared as aliases for the `courses` table. The purpose of this query is to join a table with itself.

To end this initial part of the introduction to SQL, consider the query that lists all the courses that are taken by someone:

```
SELECT  cname
FROM    takes
```

According to our definition of a table, we should get a *set* of singleton tuples back. I.e., SQL should eliminate duplicates. But for various reasons, SQL does not automatically do this. If we want duplicates eliminated we have to force the issue with a `UNIQUE` qualifier:

```
SELECT  UNIQUE cname
FROM    takes
```

8.3 Set/multiset operations

Remember that the idea in relational databases is that every operation produces a new table – a set of tuples. Therefore we should be able to use the familiar set operations: union, intersection

and (set) difference – recall the corresponding notions from relational algebra. SQL allows us to do this.

```
( SELECT room
  FROM   courses
  WHERE  teacher = "Grumpy")
UNION
( SELECT room
  FROM   courses
  WHERE  teacher = "Sneezy")
```

or

```
( SELECT cname
  FROM   courses
  WHERE  teacher = "Grumpy")
DIFFERENCE
( SELECT cname
  FROM   courses
  WHERE  room = "B21")
```

What do these mean, in English, and is there a better way of expressing them in SQL?

Duplicate elimination. Remember that vanilla `SELECT ... FROM ... WHERE` expressions do not eliminate duplicates, but what happens, for example, under `UNION`? The answer is that SQL *does* eliminate duplicates. In the example above, if Grumpy and Sneezy teach in the same room (or if Grumpy teaches two courses in the same room,) that room will only appear once in the result. One can prevent this happening by using `UNION ALL`, but the use of this is quite uncommon.

Union Compatibility. In the examples above the tables we are "unioning" or "differencing" have the same structure: the column names match, and the types of the columns match. Suppose we wanted to take the union of two tables whose column names do not match. An example of this would be to find the names of both students and teachers together with the rooms that they visit, but first let's recall the (abbreviated) schema:

```
students(id,name,email)
courses(cname,teacher,room)
takes(id,cname,marks)
```

Here's the query.

```
( SELECT name, room
  FROM   students, courses, takes
  WHERE students.id = takes.id AND takes.cname=courses.cname)
UNION
( SELECT teacher, room
  FROM   courses )
```

SQL will perform this union because it still deems the two tables that are unioned to be union compatible. Why? Because although the column names don't match, the order and types of the columns do match. The `name` and `cname` attributes have the same type. SQL will take the union and choose the column names from the first table in the union – in this case `name, room`.

Another way of thinking about union compatibility is through *relabelling*. We can always change the label of an attribute in the output of an SQL query:

```
SELECT  teacher AS name, room
FROM    courses
```

Here the `teacher` attribute is now called the `name` attribute. So we can think of union compatibility as SQL forcing an attribute label change on the second table to make the schemas identical.

Note that SQL again departs from the tenet of the relational model, that the column names are unimportant. For union, intersection and difference, it's the order that is important, not the column names. For example, I believe that SQL will not like the rather pointless query.

```
(SELECT id, marks FROM takes) UNION (SELECT marks, id FROM takes)
```

8.4 Universal Quantification

This is a fancy term for queries involving "all". For example, suppose we want to find the teachers who teach in all rooms. We can do this with the machinery we already have, but it's not straightforward. First, we need to decide on what we mean by "all rooms". Let's take an example of the relevant table.

courses:

cname	teacher	room
CompSci3	Bashful	B34
French3	Sneezy	C17
Hist3	Grumpy	C17
Hist2	Grumpy	B34

We assume that "all rooms" means all the rooms given in this table, and would be given by `SELECT room FROM courses`. Now to answer this query we go about it in a rather devious way. We first find the teachers and rooms for with the teacher is not teaching in that room.

Consider this table:

```
( SELECT c1.teacher, c2.room
  FROM   courses c1, courses c2)
DIFFERENCE
( SELECT teacher, room
  FROM courses)
```

This is a good test of your understanding of SQL. The first `SELECT` is a join with no condition! It simply pairs every possible room with every possible teacher so we get all possible teacher/room

pairs (6 in this case). Now we take away those teacher/room pairs for which the teacher *does* teach in that room, and we are left with a table – let's call it `notin` – which contains the teacher/room pairs such that the teacher does *not* teach in the room. The result is

`notin:`

teacher	room
Bashful	C17
Sneezy	B34

Now, any teacher who does not appear in this table must be teaching in all rooms. We can find these teachers by subtracting the teachers in this table from the teachers in the `courses` table. Again, note that we assume that "all" the teachers are given by those appearing in the `courses` table:

```
( SELECT teacher
  FROM   courses )
DIFFERENCE
( SELECT teacher
  FROM notin)
```

And this is the answer. Not straightforward. Notice that we had to make two uses of set difference to get the answer. If you have taken some logic, you should recall that you can replace a universal quantifier with an existential quantifier by introducing negation. That is what is going on here, and we'll return to it when we study relational algebra.

SQL is *compositional*. Wherever you can use a table name in a query, you can also use an expression (a `SELECT ...FROM ...`) in the same place. So we could take out the `notin` in the second query and replace it by the first query. Of course, that would make the query quite unreadable, and just as we do in mathematics or programming, it is good practice to decompose our expressions or programs into understandable and commented "chunks". To do this in SQL we need to introduce names such as `notin` for intermediate tables. This is a topic we'll discuss later.

SQL has some extra syntax that makes writing some programs involving universal quantification easier, but sometimes you need to resort to writing queries like the one we have just written. Try doing the following with the machinery we have established using the `takes` table which relates students, courses and marks.

- The student(s) who got the highest mark given in *any* course.
- The student(s) who got the highest mark in *some* course they were taking.
- The student(s) who got the highest mark in *all* courses they were taking.

8.5 Further features of SQL

What we have covered so far in SQL corresponds – roughly – to what one can express in first-order logic. We are now going to look at some of the extensions to SQL that have proved useful

as the language developed. It's important to note that beyond this "core" of SQL, much of the language was simply added in a rather *ad hoc* fashion in response to user demand.

The first thing is that we can do arithmetic and various string operations across attribute values. Example:

```
SELECT name, sqrt(hatsize + shoesize)
FROM person
WHERE height/weight > 10
```

This produces a table with two columns labelled with `name` and `sqrt(hatsize + shoesize)` which of course we can relabel (using `AS`) to anything we want.

We can, in a contorted way, use SQL to evaluate arbitrary arithmetic expressions. Provided you have a table, say of employees, lying around, `SELECT UNIQUE 2+2 FROM employees` produces the answer 4, but not exactly. It produces the single-column, single row table whose sole data entry is 4. Remember that SQL always produces a table.

The next extension doesn't give us any more expressive power; it is just a convenience. We can use predicates on sets (i.e. tables) to test membership, emptiness etc.

```
SELECT name
FROM   students
WHERE  id IN ( SELECT id
               FROM   takes
                      WHERE cname = "Hist4")
```

```
SELECT name
FROM   students
WHERE  id NOT IN (SELECT id
                  FROM takes)
```

These both have obvious meanings and can be rewritten without `IN`. How?

The predicate `EXISTS` is true if a set is non-empty

```
SELECTname
FROM  students
WHERE EXISTS (SELECT*
              FROM  takes
              WHERE students.id = takes.id
                    AND cname = "Hist4")
```

```
SELECT name
FROM   students
WHERE  NOT EXISTS (SELECT*
                   FROM  takes
                   WHERE students.id = takes.id)
```

More interesting is the ability to compare values with sets:

```
SELECT id
FROM   takes
WHERE  marks >= ALL (SELECT marks FROM takes)
```

```
SELECT id
FROM   takes t1
WHERE  marks >= ALL (SELECT marks
                     FROM   takes t2
                     WHERE  t1.cname = t2.cname)
```

What is the difference between these two queries? Although the second query can be expressed without using set comparisons, it is a bit tricky to do this. Try it!

In addition to `ALL` there is `ANY` whose meaning is obvious.

8.6 Aggregation

This is an extremely useful addition to SQL, which goes beyond what you can express with the "first order" core of SQL. It is the basis of all kinds of data visualisation techniques, and is regularly used in any kind of analysis of tabular data.

We'll use this table as a running example.

takes:

id	cname	marks
S0123	CompSci3	75
S0456	Hist4	62
S0123	French3	70
S0456	CompSci3	60
S0789	Hist4	77
s0111	Hist4	66

First, we can apply aggregate functions to the columns.

```
SELECT COUNT(*) FROM takes
```

```
SELECT AVG(marks) FROM TAKES
```

Again, these both produce one-row, one-column tables.

Now suppose we want to obtain the number of students in each course. Informally, suppose we could imagine "grouping" the table by course:

id	cname	marks
S0123	CompSci3	75
S0456	CompSci3	60
S0456	Hist4	62
S0789	Hist4	77
s0111	Hist4	66
S0123	French3	70

Now one could imagine doing `SELECT cname, COUNT(*)` on each table, eliminating duplicates, to get

cname	COUNT(*)
CompSci3	2
Hist4	3
French3	1

Finally one would merge these one-row tables into one table, ending up with a table that only contains what we want.

SQL allows us to do just this.

```
SELECT    cname, count(*)
FROM      takes
GROUP-BY  cname
```

Note that after `SELECT` we can put either the grouping attributes or an aggregate function on an attribute (`MAX`,`MIN`, `COUNT`, `AVG`), but we cannot put a non-grouping attribute. Thus

```
SELECT    name, count(*), id
FROM      takes
GROUP-BY  cname
```

is an error.

We can add a `WHERE` component to a group-by query:

```
SELECT    cname, count(*)
FROM      takes
GROUP-BY  cname
WHERE     cname IN (SELECT cname FROM courses WHERE teacher = "Sneezy")
```

However we might also want to select on the result of some aggregating function. For example, suppose we want to know the average marks in the courses with more than 10 students. We do this using a `HAVING` which acts as a condition on grouping attributes and aggregate functions.

```
SELECT    cname, avg(marks)
FROM      takes
GROUP BY  cname
HAVING    count(*) > 10
```

That's all there is to aggregates, but combined with joins, they can be remarkably useful for producing summary data. Incidentally, we can now see why it is useful *not* to eliminate duplicates in simple queries. Suppose we think of `SELECT AVG(marks) FROM takes` as producing a *set* of

marks and then computing the average of the numbers in that set. Would this give the desired result?

8.7 Updates

In addition to queries, a DML (data manipulation language) should also allow us to change the database. This is really straightforward.

Inserting is done by declaring the order of attributes and then giving the associated values for the tuple we want to insert:

```
INSERT INTO takes(id,cname,marks) VALUES("s0123", "French3", 78)
```

The positional correspondence between attribute names and values is important. Deletions are even simpler:

```
DELETE FROM takes WHERE id = "s0456"
```

Note that this always deletes a set. If we want to be sure of deleting just one tuple, we use a selection based on keys. Finally *modification* is accomplished by `UPDATE`:

```
UPDATE takes SET marks = marks*1.1 WHERE cname = "CompSci3"
```

Again, updates can change a set of fields. It typically makes no sense to update a key.

8.8 Views

There are all sorts of bells and whistles in SQL. For example, there are various ways of formatting the output and writing the result to a file.

For example one can write the output of a query into another table using `SELECT... INTO...`:

```
SELECT * INTO highmountains
FROM munros
WHERE heightf > 4000
```

Creating intermediate tables is often useful – if only to make a complicated query more understandable.

Views. A view is simply a query to which we give a name. E.g.,

```
CREATE VIEW highmountains AS
SELECT *
FROM munros
WHERE heightf > 4000
```

What is the difference between a view and a table that is created by `SELECT ... INTO ...`? The answer is that when the underlying database is changed, e.g., someone adds a new mountain or changes the height of one, the view is changed. Thus a view behaves like a *niladic* function – a function with no arguments – that is evaluated on demand.

However, this is a simplistic idea of how a view might be maintained. Another possibility is to *materialise* the view and then compute the changes needed when the database is updated. This may require much less work than re-evaluating the whole view whenever it is needed.

Most end users and applications programs see views, not the underlying data. We touched on this when we discussed the "three-level" architecture of traditional database systems. This leads to another issue: the *view update problem*. If one updates a view, how is the update expressed as a change to the underlying database. Only for very simple queries is the update unambiguous.

Embedded SQL. We have frequently stressed that most SQL is generated by other programs, not by people. How this is done is beyond the scope of these lectures, but roughly speaking the client program sends a string to the database server, and the server returns a sequence of tuples. The output of the query is read by the client in much the same way that as it would read a file.

8.9 Limitations of SQL

What cannot be expressed in SQL is an interesting and involved topic. Sooner or later, usually sooner, we find things that we cannot do in SQL so we transfer the data into some programming language and use the power of that language. The reason for having SQL is not that it is "simple" but that programs written in SQL can generally be optimised by the SQL interpreter. Go through the examples we have used here, and you will find that in many cases there is a "brute force" way of evaluating them and an efficient way. SQL, quite often, figures out the efficient way.

8.10 SQL Exercises

On the web page you will find brief instructions for connecting to the Postgres server. Once connected, use "`\h`" for help on the command syntax, especially for things like reading SQL from files. Also, for details on SQL use the postgres web site at `http://www.postgresql.org/docs/7.3/interactive/index.html`. This is a very readable interactive tutorial and manual.

There are two tables in the database, `munros`[1] and `hikers`. The DDL for `munros` is:

```
CREATE TABLE MUNROS(
        id INT,
        name TEXT,
        heightm INT,
        heightf INT,
```

[1]The data for this table is a *highly corrupted* version of information extracted from the web, from `http://www.peakbagging.com/` and from `http://www.multimap.com/`.

```
    trig TEXT,
    map CHAR(2),
    altmap CHAR(2),
    grid  char(2),
    easting INT,
    northing INT,
    PRIMARY KEY(id) );
```

For the uninitiated, a Munro is a mountain in Scotland whose height exceeds 3,000 feet. Here is a brief explanation of the attributes:

- `id` and `name`. The identifier and name of the mountain. `id is a key for the table`.
- `heightm` and `heightf`. The height of the peak expressed in metres and feet.
- `trig`. The existence of a "trig point". Ignore this.
- `map` and `altmap`. The name of the Ordnance Survey map that includes this mountain and an alternative map, if one exists.
- `grid`. The Ordnance Survey mapping system covers the UK with a square grid. Each grid square is 100km by 100km and is given a two-letter name, such as "`NN`".
- `easting` and `northing`. The east and north distances in *metres* from the south-west (bottom left) corner of the grid square. Think x, y co-ordinates!

The DDL that describes which hiker has climbed which peak is simple:

```
CREATE TABLE hikers (
     mid INT,
     hname TEXT,
     time INT,
     PRIMARY KEY(mid,hname),
     FOREIGN KEY (mid) REFERENCES munros(id));
```

In this, `mid` is the mountain that has been climbed, `hname` is the name of the hiker, and `time` is the duration of the climb – the time taken to get up and down again.

Write SQL programs that do the following.

1. Are the heights of the of the mountains consistent? A metre is approximately 3.28 feet.

   Answer:

   ```
   SELECT *
   FROM   munros
   WHERE  (heightf - 3.28 * heightm)^2 > 9;
   ```

The last line checks – not particularly efficiently – if the two heights differ by more than 3 feet. I inserted a "rogue" Munro, Beinn Scrui, in downtown Fort William, for which this check fails!

2. What are the lowest and highest mountain(s)?

   Answer:

```
SELECT *
FROM   munros
WHERE  heightm = ( SELECT min(heightm)
                   FROM   munros);
```

   Similarly for max. Or you could write a union of two queries, or a query with an `OR` in the `WHERE` clause to get both kinds of Munro from one query.

3. For each grid square, the name of the square, the average height (metres) and the number of mountains.

   Answer:

```
SELECT   grid, count(*), avg(heightm)
FROM     munros
GROUP BY grid;
```

4. The mountains within 10km of Ben Nevis (note that Ben Nevis is more than 10km from the boundaries of the grid square in which it lies)[2]. Use Pythagoras and watch out for integer overflow.

   Answer:

```
SELECT m.name
FROM   munros m, munros b
WHERE  b.name = 'Ben Nevis' AND b.grid = m.grid AND
       ((m.easting - b.easting)/1000.0)^2 +
       ((m.northing - b.northing)/1000.0)^2 <=100 ;
```

5. All the maps (including the alternate maps) together with a count of the mountains on each map.

   Answer:

```
SELECT   map, count(*)
FROM     ( ( SELECT id, map
             FROM   munros)
           UNION
           ( SELECT id, altmap AS map
             FROM   munros
             WHERE  altmap IS NOT NULL ) ) thingy
GROUP BY map;
```

[2]There was a typo in the original handout, which gave one of these figures as 20km.

or

```
SELECT   map, count(*)
FROM     ( ( SELECT map
             FROM   munros)
           UNION ALL
           ( SELECT altmap AS map
             FROM   munros
             WHERE  altmap IS NOT NULL ) ) thingy

GROUP BY map;
```

Notes:

- `UNION` eliminates duplicates, so one needs either to use `UNION ALL` or – better in my opinion – keep the `id` field so we don't have to worry about whether SQL is going to eliminate duplicates.
- The attribute alias `AS map` is not needed, but for some reason that eludes me, the table alias `thingy` is needed for an expression inside the `FROM` clause, even though it is never used!
- Without the `IS NOT NULL` predicate, the query would count the number of peaks on on "null" (alternative) map – this is highly dubious!

6. The hikers and the average height of the mountains they have climbed.

   Answer:

```
SELECT   hname, avg(heightm)
FROM     hikers, munros
WHERE    hikers.mid = munros.id
GROUP    BY hname;
```

7. The hikers and grid squares for which the hiker has climbed all the mountains in the grid square.

   Answer: This is tricky. This is worked using views, but SQL is out of its depth when it comes to optimising this.

```
/*All possible hiker name/mountain id pairs*/
CREATE VIEW allpairs AS
SELECT hname, id
FROM munros, hikers;

/*The hiker name/mountain id pairs such that the
hiker has climbed the mountain*/
CREATE VIEW hasclimbed AS
SELECT hname, id
FROM munros, hikers
WHERE hikers.mid = munros.id;
```

```
/*The hiker name/mountain id pairs such that the
hiker has *not* climbed the mountain*/
CREATE view notclimbed AS
SELECT *
FROM ((SELECT hname, id FROM allpairs)
      EXCEPT
      (SELECT hname, id FROM hasclimbed)) thingy;

/*The hiker name/grid square pairs such that the
hiker has not climbed a mountain in that square*/
CREATE view notingrid
AS SELECT hname, grid
FROM notclimbed, munros
WHERE notclimbed.id = munros.id;

/* Now subtract this from all possible hname/grid pairs
to get what we want */
SELECT hname, grid
FROM hikers, munros
EXCEPT
SELECT * FROM notingrid;
```