1 Structured Query Language: Again

So far we've only seen the basic features of SQL. More often than not, you can get away with just using the basic `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements. Sometimes you do need to do rather complicated things (queries) that would be too cumbersome or inefficient to do via the simple means. Today we discuss some of the less-then-common SQL things.

Note that some things may not work in some databases. For example, most of these notes will not work in MySQL. Most of it should work in Oracle, Sybase, and SQL Server.

2 Joining Tables

Not necessarily uncommon—in fact, rather common operation—is joining tables. There are several ways of joining tables in SQL. One is by listing several tables you're trying to get data from:

```
SELECT *
FROM Table1 a, Table2 b
WHERE a.someid = b.someid ;
```

Another way of doing the same thing is by using an explicit `JOIN` operator:

```
Table1 INNER JOIN Table2
```

This basically means that if some field matches in both `Table1` and `Table2`, then that's used for joining.

There are four join types:

1. `inner join`: Only the matching fields are returned.
2. `left outer join`: Everything from left operand table, and whatever matches from the right. If no match, then those values are set to *null*.
3. `left outer join`: Everything from right operand table, and whatever matches from the left. If no match, then those values are set to *null*.
4. `full outer join`: This is combined left and right outer join.

There are also several conditions that can be used:

1. `natural`: When used, it indicates not to duplicate repeated fields.
2. `on <condition>`: This condition is similar to the `WHERE` clause in the `SELECT` statement.
3. `using(<column>,...)`: This tries to match that column in both tables.

3 Calculating Values

Using SQL, you can perform simple calculations, like computing functions, or calling built-in routines. For example, computing absolute values of some field is just as easy:

```
SELECT ABS(<some-field>)
FROM <table-name> ;
```

Usually there is a whole array of functions: such as `ABS()`, `POWER(n,m)`, `TRUNC(n,m)`, `ROUND(n,m)`. There is also usually a whole assortment of date conversion functions.

Obviously you can also use basic arithmetics; for example, to add 8 to any result, you can just do:

```
SELECT <some-field> + 8
FROM <table-name> ;
```

3.1 Column Alias

Using calculations or functions in the select field is useful, but the resulting column name is really long (usually representing an entire expressions used to calculate that field).

You can set a column alias, which will be return as the column name. For example:

```
SELECT start,end,end-start AS diff FROM somedata;
```

Now, instead of getting the result with 3 columns named: ”start”, ”end” and ”end-start”, you’d get a result with 3 columns named ”start”, ”end”, and ”diff”.

You can use these types of aliases to make output more readable, and queries a bit more manageable.

4 Grouping & Aggregate Functions

Group functions are functions that operate on groups of data (as opposed to a single value). These include averaging, counting, etc. A few common ones are described next:
AVG: Returns the average.
COUNT: Returns the count of values
MAX: Returns the maximum value
MIN: Returns the minimum value
SUM: Returns the sum of all

To use them, you have to select them. For example:

```
SELECT MAX(grade), AVG(grade) FROM testdata;
```

The above would return the maximum and average grades from testdata table.

The data has to be properly grouped for these to work. For example, in the above query, we simply went through the whole table. If we wanted to generate a list of students with a maximum and average grade for each particular student, we’d have a problem. For that, we need to use the GROUP BY clause. This one is generally appended at the end of SQL statements (possibly after the ORDER BY clause). For example:

```
SELECT studentid, MAX(grade),AVG(grade)
FROM testdata GROUP BY studentid;
```

Now, we will get a list of studentids, with each one having two numbers, one for the maximum grade, and the other one for the average grade. (note that this depends on the actual layout of the table; in this case, we're assuming that testdata has studentid and grade, and has many of these entries - which is kind of unlikely practically).

There is also a `HAVING` clause that applies to groups. It is similar to a `WHERE`, except it applies to groups.

5 Duplicates

We can get rid of duplicates from a result of a query by specifying `DISTINCT` right before column name. For example:

```
SELECT DISTINCT name
FROM PERSON
```

6 Sub-Queries

Often it is useful to do a query inside a query. For example, lets say you want to find how many grades are above average. This naturally asks two questions: What is the average?, and then How many grades are above that. You can often do that with a single query.

7 Set Operations

A 'relation' is basically just a set of tuples. Which naturally leads to all the set operations—which we can perform on databases to find results we are interested in.

7.1 Union

Union combines results of two queries into a single result set. It automatically eliminates duplicates. So for example, let's say we wanted to make a list of all the faculty and students, we might do something like:

```
(SELECT LNAME,FNAME FROM FACULTY)
UNION
(SELECT LNAME,FNAME FROM STUDENT)
```

The end result of the above is a list of last and first names of all the faculty and students. It is important to note that for a union, the number of fields (and their types) have to match.

7.2 Union All

"Union All" is identical to Union, except that all the duplicates are maintained.

7.3 Intersect

Intersection is pretty much just like Union (all the set operations are similar), except it returns the common elements found in both queries.

7.4 Minus/Subtract

Minus (or Subtract) (or Set Difference) is another operation. For two sets, $A - B$, it returns everything that is in set A, but is not in set B. Same for tables.

This operator is also often known as "except" operator.

7.5 Simulating Other Set Operations

By having Union, Intersection, and Minus, you can do quite a bit using set operations (in fact, you can do pretty much anything). You can also simulate set operations using other set operations.

8 Set Membership

SQL has operations to check if some element is inside a set. For example, let's say we'd have a table of students, and a table of faculty, and we would like to find out if any of the faculty are also students. We might accomplish this task by:

```
SELECT DISTINCT SSN
FROM FACULTY
WHERE SSN IN (SELECT SSN FROM STUDENT)
```

There is also the 'not in' operator; which works in similar ways.

9 Set Comparison

Just like we could check for set membership, we can also check if the set has certain properties. We can ensure that all elements of a set meet some condition... of it *some* elements of the set meet some condition. For example:

```
SELECT SSN
FROM FACULTY
WHERE AGE > SOME (SELECT AGE FROM STUDENT)
```

The above would return SSN of faculty who are older than *some* of the students. Similarly, we can also use:

```
SELECT SSN
FROM FACULTY
WHERE AGE > ALL (SELECT AGE FROM STUDENT)
```

Which will return the SSN of faculty who are older than all students.

9.1 Use In Having

These types of operations can be used in the `HAVING` clause, to make decisions on a per-group basis.

10 More Set Operations

Similarly to the set membership, we can also check if a particular set is empty or not. We do that via the `EXISTS` operator. We can also check if all values in a set are unique, via the `UNIQUE` operator (or similarly, via the `NOT UNIQUE` operator).

11 Views

Database Views are logical database tables. They don't store any values and cannot be updated. They are defined by a select statement. For example, lets say for some application, all you need to know is the product_id and price, but don't really care for a product description. You can create a view that only gives you these few fields.

```
CREATE VIEW product_price_view
AS SELECT product_id, price FROM product;
```

We've created the a view, that's defined as a select statement. Now that we have the view, we can select from it just like from a table!

```
SELECT * FROM product_price_view;
```

This results in:

```
PRODUCT_ID      PRICE
---------- ----------
         3      26.99
         4      19.95
```

We can also do nifty table like things like:

```
DESCRIBE product_price_view;
```

Views are very useful when granting certain applications rights to some data. Views can build queries several tables, and provide only the needed information, in a relatively security limited way.

Views also tend to be optimized; which implies that your SQL statement inside a view will be optimized to execute very efficiently

Databases also tend to cache view results; which could speed up retrieval significantly if cache wasn't invalidated by an update.

12 Transactions: Oracle

A transaction is one or more related SQL commands that constitute a logical unit of work. They either all have to work, or they all have to fail.

Often, when you're interacting with the database, you're in a transaction. When you quit the database client, all of your operations are committed. The following applies to Oracle, but has nearly identical syntax/functionality under most other databases.

You commit a transaction by typing: `COMMIT`.

If something is not going well, you can roll back the transaction by typing: `ROLLBACK`.

Note that DDL is not rolled back (once you drop a table, you can't roll it back, or once you add a column, you can't remove it).

You can also define save points. These are points to which you can roll back the transaction (as opposed to rolling back the entire thing).

You define a save point by issuing: `SAVEPOINT savepointname`.

You rollback to that save point by: `ROLLBACK TO savepointname`.

13 Locking Records: Oracle

Sometimes you'd like to select records to be updated, but you don't want them to be updated in the meantime while you're looking at what you've selected.

For example: You search for an airline flight, after finding it and noticing that there is an open seat, you decide to buy it. After entering your credit card info, etc., the system notifies you that there are no available seats: someone has taken your seat while you were in the process of buying it.

Oracle avoids that by allowing you to lock a record when you select it. This is called selecting `FOR UPDATE`.

An example of a command may look like this:

```
SELECT *
FROM productorder
WHERE order_id = 1 AND
    product_id = 3 FOR UPDATE OF QUANTITY;
```

Which produces:

```
  ORDER_ID PRODUCT_ID   QUANTITY
---------- ---------- ----------
         1          3          2
```

This selects an order, where we may update the quantity. Nobody else is allowed to update the quantity. For example, if we go into another window and run the same command, the window will just sit there, waiting for the original update to complete.

Now, we may or may not update it! But we do have to `COMMIT` to release the lock. So, after we do a commit, the 2nd window locked those records, and now the first one will be locked out if we do that query.

What we could do now is attempt to lock it, and bomb out with an error if we can't. To do that, we add on `NOWAIT` to the SQL, which just says that we're not willing to wait for someone to update the other table. The error is: `ORA-00054: resource busy and acquire with NOWAIT` specified

Also note that once some select has selected something for update, nobody else can update that record until you do the commit.

14 Sample Questions

For the database below:

```
employee(employee-name, street, city)
works(employee-name, company-name, salary)
company(company-name, city)
manages(employee-name, manager-name)
```

1. Find 'John Doe's Manager's Name.
2. Find employee's whom 'John Doe' manages.
3. Find all companies that have exactly 75 employees.
4. Find in how many cities is 'First Bank Corp.' located.
5. Find how many employees work for 'CityBank' in New York.