10. SQL- Data Manipulation Language

SQL's DML includes statements to do alter the rows in a table, and to get data from one or more tables. Notice that this statements do NOT alter the schema at all, only the data in the table. The main statements are:

- **INSERT INTO** This statement allows you to insert one or more rows into a table.
- **DELETE FROM** This statement allows you to delete one or more rows from a table (but leaves the schema intact)
- **UPDATE** This statement allows you to change one or more fields from one or more rows.
- **SELECT** This statement allows you to get data from one or more tables. It does NOT change the data in any way.

To help illustrate the statements with examples, we will use the following two tables (we will define other tables when needed)

```
CREATE TABLE Book (
      ISBN  CHAR(10)     PRIMARY KEY,
      Title VARCHAR(40) UNIQUE NOT NULL,
      Pages Integer
);
```

Example 1: A simple book table

```
CREATE TABLE Student (
      Id     INTEGER     PRIMARY KEY,
      SSN    CHAR(9) UNIQUE NOT NULL,
      Name   VARCHAR(20) NOT NULL,
      DoB DATE,
      Gender CHAR(1) CHECK(Gender='F' OR Gender='M'),
      Major  CHAR(3),
      Credits INTEGER
);
```

Example 2: A simple Student table

10.1. INSERT INTO

This statement allows you to add a row to a table. Its basic syntax is as follows:

```
INSERT INTO tableName (field1,field2,...) VALUES (value1, value2,...)
```

Example 3: INSERT INTO syntax

It is not necessary to list all fields, but it is a good idea. If you don't list fields, then it is assumed that

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

the values correspond to all the fields in the table in the order they were defined (which means that if the schema changes, your statement will probably result in a syntax error). As mentioned previously, character constants are enclosed in single quotes (').

As an example, if we were to insert one row into the Book table defined as in Example 1, the statement would look like this:

```
INSERT INTO Book (ISBN,Title,Pages)
      VALUES ('0201700735','The C++ Programming Language',1030);
```

Example 4: INSERT INTO example. Notice single quotes for string constants

Notice that the constraints set on the table will be enforced by the DBMS, and that a row that does not satisfy those constraints will be rejected; for example, the statement in example 4 would generate an error message if a row with that ISBN already existed.

Example 5 shows an insert statement for the student table defined in Example 2. Notice that we are specifying a date by using a string representation for it; the particular string representation will be determined by your DBMS and its settings (since people use different formats for specifying dates; issues like whether the day or month goes first, whether we do month in letters or digits, and what do we use for separating the fields); the example displays one possible representation, that works with the default settings of both Oracle and PostgreSQL.

```
INSERT INTO Student(Id,SSN,Name,DoB,Gender,Major,Credits)
      VALUES(1,'123456789','Orlando','01-JUN-1970','M','CS',20);
```

Example 5: INSERT INTO example. Notice how dates are entered

10.2. DELETE

The DELETE FROM statement allows you to delete one or more rows from a table. If it includes a WHERE clause, it deletes only those rows that satisfy the predicate, if there is no WHERE clause then all rows will be deleted. Notice this statement does NOT alter the schema of the table, even if it deletes all the rows.

Example 6 shows SQL code for deleting all male students (that is, rows with 'M' on their gender) from the student table.

```
DELETE FROM Student
WHERE Gender='M'
```

Example 6: DELETE all rows for male students from the student table

10.3. UPDATE

The UPDATE statement allows you to change one or more fields from one or more rows of a table. Like DELETE, an UPDATE modifies all rows or just some of them depending on the WHERE clause.

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

Example 7 shows SQL code for increasing the number of credits and changing the major of students in the BW major.

```
UPDATE Student
SET Credits=Credits+5, Major='BWT'
WHERE Major='BW'
```

Example 7: Increase the number of credits by 5 and set the major to BWT for students in the BW major.

10.4. The SELECT Statement

The SELECT statement allows you to get data from one or more tables. It does NOT change the data stored in any way, it just allows you to read it. The SELECT Statement is probably the most complex SQL statement, since it needs to allow you for complex conditions and to get data from several tables.

In its basic form, the SELECT statement contains three clauses, although the WHERE clause is actually optional. After the SELECT keyword, we specify the fields (or expressions) we want to retrieve; then we add the FROM keyword and the table from which we are getting the data, and finally a WHERE clause that specifies which rows are to be returned.

In order to illustrate the SELECT statement, we will use a table called Person containing five fields, and the sample data as in the following table (Table 1). We record the person's id, name, age, gender and the country they were born in (using two-letter ISO country codes for the countries, so us is the United States of America, mx is Mexico, and in is India)

Id	Name	Age	Gender	Country
1	Orlando	35	M	us
2	Lina	25	F	mx
3	Jose	45	M	us
4	Krishnapriya	27	F	in

*Table 1: Sample data for **Person** table for simple select examples*

The simplest select statement uses * instead of a list of attributes (retrieving ALL columns from the table) and does not use a WHERE clause (retrieving ALL rows), so the statement to list all data in the person table would be:

```
SELECT *
FROM Person
```

Example 9: Retrieving all data from Person table

We can also specify a list of fields, or even expressions. When using expressions, it is often convenient

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

to give the expression a nice name, so SQL allows us to alias a field or expression, by writing the keyword AS and then alias after the field name or expression (actually, the AS keyword is optional in most DBMSs, although PostgreSQL requires it).

So the SQL code in Example 10 would return the data as shown in Table 2. Notice that the SELECT statement does NOT alter the original table in any way (we have neither changed the field name nor the number of credits in the Student table); it just returns a new temporary result calculated based on the original table.

```
SELECT Id AS Identifier, Name,Age+5 AS AgeIn5
FROM Person;
```

Example 10: Retrieving specific data from Person table, illustrating the aliasing done with the AS keyword.

Identifier	Name	AgeIn5
1	Orlando	40
2	Lina	30
3	Jose	50
4	Krishnapriya	32

Table 2: Sample result from Example 10, notice change on column names

Identifier	Name	AgeIn5
1	Orlando	40
2	Lina	30
3	Jose	50
4	Krishnapriya	32

Table 3: Sample result from Example 11, notice change on column names

Notice the last field retrieved uses an expression, Age+5. Most DBMSs support the standard math operators, many of the basic math functions (floor, sin, cos etc), and many string functions (upper, lower, substr etc); consult your DBMS's manual.

SQL statements become slightly more interesting when we add conditions, which allow us to retrieve only certain rows, rather than the whole table. We can add conditions with the WHERE clause, and can use the traditional comparison operators (<, <= etc) and combine them with the traditional logical operators (AND, OR, NOT). Differently from most programming languages, SQL uses the actual words AND OR and NOT for those operators. The precedence of the operators is the usual (NOT, then AND, finally OR), and you can use parenthesis to alter that precedence.

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

So, a query that would return only females who are older than 20 would be:

```
SELECT *
FROM Person
WHERE Gender='F' and Age>20
```

Example 11: Retrieving Females who are older than 20

10.4.1. Aggregate Functions

SQL also allows you to use functions that compute a result based on several rows; we call those **aggregate functions**. SQL 92 supports the functions **COUNT, SUM, AVG, MAX** and **MIN**, with the meaning suggested by their name.

For example, we can get the number of rows in the Person table by writing:

```
SELECT COUNT(*)
FROM Person;
```

Example 12: Retrieving number of rows from person table

We can, of course, use a WHERE clause to control wich rows gets counted, or included in the aggregate. For example, if we wanted to know how many females there are we would write:

```
SELECT COUNT(*)
FROM Person
WHERE Gender='F';
```

Example 13: Retrieving number of females from person table

Now, what if we need to get the number of people per gender ? Given that there are only two genders, maybe we could just write two separate queries, one for females and one for males, and put together the results in a piece of paper; but what if we need the number of people per country of origin ? We will probably have many countries, plus our set of queries will only be valid for a particular moment in time, since the set of countries may change.

SQL allows computing aggregates over groups of rows, where the rows in a table are divided into groups based on some field or expressions over those fields; we do that by using the aggregates, and adding a GROUP BY clause to our SELECT statement. For example, the following query:

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

```
SELECT Country, COUNT(*) as NumPeople, AVG(Age)
FROM Person
GROUP BY Country;
```

Example 14: Retrieving number of people per country

Would retrieve a table containing all countries in the person table, with the number of people per country and their average age. All rows in the person table are grouped according to their country, and the aggregate (count) is calculated for each group. Conceptually, we can think of the table being sorted on the fields mentioned in the GROUP BY clause, and then the totals calculated per group.

Conceptually, we can view the sample data organized as in Table 3 (each group of rows is marked in a different color):

Id	Name	Age	Gender	Country
1	Orlando	35	M	us
3	Jose	45	M	us
2	Lina	25	F	mx
4	Krishnapriya	27	F	in

*Table 4: Sample data for **Person** table grouped according to country (same as table 1, resorted)*

And then the totals calculated per group, yielding the result in table 4:

Country	NumPepole	AvgAge
us	2	40
mx	1	25
in	1	27

Table 5: Result for executing example 15 on the sample data of table 3

Notice that every field retrieved by the SELECT needs to either be an aggregate, or be mentioned in the GROUP BY expression. This is because if we tried to retrieve a normal field without grouping by it, the different rows in the group may have a different value for that field !

Also, sometimes we want to restrict the results of the query based on some condition calculated *after* the grouping, say get only countries with more than one person; we can use the HAVING clause to do that; for example, to modify the above query to only return countries with more than one person we would write:

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

```
SELECT Country, COUNT(*) as NumPeople, AVG(Age)
FROM Person
GROUP BY Country
HAVING COUNT(*) >= 1
```

Example 15: Retrieving number of people per country, only countries with more than one person

10.4.2. JOINS

Up to now, we have only seen how to get results from a single table; but we oftentimes need to combine information from several tables; in fact, good relational design requires us to divide our information into many tables, and to get useful information we need to combine those tables.

The main operation to combine information from several tables is the **join** operation. We can think of a join as a combination of a **cartesian product** of two tables (which gets us all pairs of rows, with one coming from each table) followed by a selection process (which allows us to get only those pairs that 'match' according to some predicate).

SQL supports several ways to do joins. The simplest way, which I call an **implicit join** is to put two or more tables in the FROM clause, separated by commas. The join condition (that is, the predicate that decides which rows match) is added to the WHERE clause.

For example, assume we have another table, called country, that gives us the name of the country given their iso code (we have an extra country that we'll use later):

Code	Name
us	United States of America
mx	Mexico
in	India
cn	People's republic of China

*Table 6: Sample data for **Country** table*

So, if we want to get the names of all people with the name of the country they were born in, we can use the following query:

```
SELECT Person.Name, Country.Name
FROM Person, Country
WHERE Person.Country=Country.Code
```

Example 16: Retrieve names of all people with their country's name

Notice how we need to use Person.Name and Country.Name in the first line, to disambiguate to which field we are referring to. For clarity, we also do it on the third line, but it is not necessary there (since

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

there are no two fields called Country or Code). To save on typing but keep the clarity, I usually alias the tables to their initial letter, so the query looks like this:

```
SELECT P.Name, C.Name
FROM Person P, Country C
WHERE P.Country=C.Code
```

Example 17: Retrieve names of all people with their country's name, aliasing tables

For a simple query like this, an implicit join is appropriate; however, when you are joining several tables, and have many conditions, some arising from the join and others from other requirements, the implicit join syntax can easily lead to mistakes.

SQL provides another syntax for join statements, which can make the intent easier. Since this syntax includes the JOIN keyword, I call this an **explicit join** statement. The syntax involves writing the table names, with the keyword JOIN between them, followed by the keyword ON and a condition between parenthesis; for example,

```
SELECT Person.Name, Country.Name
FROM Person JOIN Country ON (Person.Country=Country.Code)
```

Example 18: Retrieve names of all people with their country's name

10.4.3. NATURAL JOIN and USING

Most join conditions involve the equality predicate. SQL supports the concept of a NATURAL JOIN, which is a join in which the join condition is created implicitly, by requiring equality of all fields with the same name in both tables.

In order to illustrate the syntax, if we want to get all people with the same name as their country (hey, it's my example :) we would type the following query:

```
SELECT *
FROM Person NATURAL JOIN Country
```

Example 19: Retrieve info of all people with the same name as their country

I consider natural joins to brittle to use in real life applications; since adding a field to the table may silently change the results of the query. SQL also supports a JOIN ... USING syntax that is much better. Rather than writing ON and the join condition, we write USING and then a list of fields which must match; so the above query with USING would look like:

```
SELECT *
FROM Person JOIN Country USING (Name)
```

Example 20: Retrieve info of all people with the same name as their country

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

10.4.4. OUTER JOIN

Many times we want to make sure ALL rows from a certain table appear in a join, even when there is no corresponding join on the other table. We can achieve this by using an OUTER JOIN. We need to specify whether we want a LEFT (that is, all the rows from the table on the left appear), RIGHT or FULL (rows from both tables re guaranteed to appear) OUTER JOIN.

For example, if we want to get the name of each person with the name of their country, and we want to make sure ALL people appear, including those with no country in the database (so the country would be null) we need to do an OUTER join; as follows:

```
SELECT P.Name, C.Name
FROM Person P LEFT OUTER JOIN Country C ON (P.Country=C.Code)
```

Example 21: Retrieve names of all people with their country's name, making sure ALL people appear

Notice in this case we are using a LEFT outer join, since the Person table is to the left of the JOIN keyword and we want all rows in that table to appear. If we wanted to make sure all countries appear, we would use a RIGHT outer join in this case; and if we want to make sure both all people and all countries appear we would use a FULL outer join.

10.4.5. Subqueries

In most places where we use a table name or an expression that returns a list, SQL allows us to substitute another SQL query (a subquery). We can use subqueries in FROM clauses, or instead of a list, for operators that accept a list (IN, EXISTS, =ANY etc)

For example, imagine that we add a new table, HasLived, that reflects the fact that a person has lived in a country. The table with sample data could look like Table x

Person	Country
1	mx
1	us
2	mx
3	us
4	in
4	us

*Table 7: Sample data for **HasLived** table*

From this and the previous tables, we can see that Orlando (with id 1) has lived in Mexico and the USA, that Lina has lived in Mexico etc

Now if we are asked to find the names of people who have lived in the country with id 'us' (namely the USA); the query would look like:

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

```
SELECT P.Name
FROM Person P JOIN HasLived Hl ON (P.Id=Hl.Person)
WHERE Hl.country = 'us';
```

Example 22: Names of people who've lived in the country with id 'us'

But imagine that we are asked for the names of people who have NOT lived in the 'us'; we could be tempted to modify the condition in the WHERE clause above, as follows:

```
SELECT P.Name
FROM Person P JOIN HasLived Hl ON (P.Id=Hl.Person)
WHERE Hl.country != 'us';
```

Example 23: (wrong) Names of people who've NOT lived in the country with id 'us'. Actually does names of people who've lived somewhere other than in 'us'

The problem is that this query actually asks for the people who have lived in a country other than 'us', which, given that there may be more that one row per person (i.e. a person may have lived in more than one place), it is NOT an equivalent query.

In our example, the only person who has not lived in 'us' is Lina (person with id 2); but the above query would also return the people with id 1 and 4, since they've lived in other countries.

The right solution would look like:

```
SELECT P.Name
FROM Person P WHERE Id NOT IN (
        SELECT Person
        FROM HasLived
        WHERE Country='us'
);
```

Example 24: Names of people who've NOT lived in the country with id 'us'

Here the inner query selects the id's of people who have lived in 'us' and the main query selects all people except those.

As another example, say we're asked to find people who have lived in both the country with id 'mx' and the one with id 'us'; again a naïve (and wrong) attempt would be:

```
SELECT P.Name
FROM Person P JOIN HasLived Hl ON (P.Id=Hl.Person)
WHERE Hl.country = 'us' AND Hl.country='mx'
```

Example 25: (wrong) Names of people who've lived in both 'us' and 'mx

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/

This query would actually return no rows at all. The condition in the where clause, is tested for the *same* row, so we are actually requiring that the *same* country field be 'us' and 'mx', which is obviously impossible, since a field can only have one value for a given row.

The correct query would use IN, as follows:

```
SELECT P.Name
FROM Person P WHERE Id IN (
     SELECT Person
     FROM HasLived
     WHERE Country='us'
) AND Id IN (
     SELECT Person
     FROM HasLived
     WHERE Country='mx'
);
```

Example 26: Names of people who've lived in both 'us' and 'mx

Licensed under Creative Commons, Attribution, Share-Alike http://creativecommons.org/licenses/by-sa/3.0/