Chapter 3
Structured Query Language - SQL

Structured Query Language, or SQL (pronounced S-Q-L by some, and "sequel" by others) is a programming language that has become the de facto standard for working with relational databases. As suggested by the "Q" in its name, SQL has one set of constructs for expressing queries to a relational database, and this will be the focus of this chapter. SQL also has constructs for data definition, which includes defining tables, and for data manipulation, which involves inserting, deleting, or modifying the data already in the database[1]. We discuss these aspects of the language in Chapter 5.

Section 1. Basic Querying in SQL

SQL has one fundamental command for querying a database -- the **SELECT** command. Since all data retrieval is done with this command you might suspect that it supports enough options to accommodate a wide range of possible types of data requests, and you will be right! Indeed, one of the difficulties faced by SQL novices (and veterans who have used one approach to querying for a long time) is that for all but the simplest of queries there are usually several different ways to write the query in SQL. Often considerable time can be spent mulling over a "best" approach, when in fact this is often a minor consideration.

The basic form for the SELECT command is

```
SELECT attribute-list
FROM table-list
WHERE condition;
```

Here $TABLE_1, \ldots TABLE_n$ represents a table-list, $ATT_1, \ldots, ATT_m$ represents an attribute-list, and *condition* represents a condition or operation that yields a Boolean value. Don't be deceived by the word SELECT in the syntax into thinking that it associated with a selection operation from relational algebra. In fact the semantics of the SELECT operator can be conveyed via the following relational algebra expression:

$$\Pi_{ATT_1, \ldots, ATT_m} (\sigma_{condition}(TABLE_1 \times \ldots \times TABLE_n)$$

In SQL we can use the same name for two or more attributes in a query as long as the attributes are from different relations. If one needs to refer to an attribute in one table that has the same name as an attribute from another relation, however, it is necessary to qualify the attribute name with the relation/table name as shown below:

```
table-name.attribute-name
```

Finally, attribute name ambiguities may also arise in the case where a query needs two copies of the same relation. In this case one is allowed to declare one or more single-letter, alternative relation names (or aliases) as shown below

```
SELECT attribute-list
FROM table-name alt-name1 alt-name2
WHERE condition;
```

1. Attribute Projection Involving One Table

```
SELECT attribute-list
FROM table-name
```

This query returns only those attribute values (in the order listed) for each tuple in the table. In classical relational database terminology, selecting attributes is known as a *projection* operation.

[1] Technically the querying constructs also fall under the data manipulation facet of SQL but since what we treat later looks so different from querying, it is more appropriate to treat them in separate chapters.

If instead of an attribute list we use the symbol * then the query returns all of the attributes of the designated table, in the order in which they appeared when the table was created.

Examples: Done in class

2. Tuple Selection Involving One Table

```
SELECT attribute-list
FROM table-name
WHERE condition
```

This query returns the attribute values listed for each tuple in the table whose values satisfy the given condition. Such an operation is known as a *selection*.

- At this point we do not get too specific about what constitute valid conditions. It will be sufficient for our purposes to note that SQL supports the relational operators (`=, <, <=, >, >=, <>, !=`) and the logical operator `AND, OR, NOT`.
- SQL also supports an operator `is null` which, when following an attribute name, tests as ***true*** when applied to a tuple that has **null** for the associated attribute value. Likewise there is an **`is not null`** operator.

Examples: Done in class

3. Elimination of Duplicate Data

Relations are not allowed to have identical tuples and this can be enforced by a DBMS. Conceptually, the output of a SELECT statement is another (anonymous) relation, and as well shall see in the next section, such a relation can, in turn, be used in another SELECT statement.

Alas, it is very expensive in terms of execution time to eliminate duplicate tuples form the output of a query; consequently, the default option is to leave them in. By inserting the keyword **DISTINCT** after the word SELECT, however, one can designate that duplicate tuples be removed.

```
SELECT DISTINCT attribute-list
FROM table-name
.rest of query
```

Examples: Done in class

4. Data Ordering

The order in which tuples are displayed in SELECT statements is, by default, unspecified, and is generally related to the algorithms used to implement the SELECT statement. A user can control this ordering, however, by means of the **ORDER BY** clause, which appears as the last line in a SELECT statement

```
SELECT attribute-list
FROM table-name
WHERE condition1
ORDER BY ordering-list
```

Here ***ordering-list*** is a comma-separated list of attribute names, , aliases, or column positions (integers, with the first attribute in the column list representing position 1) that prescribe a *cascading order sequence* (that is, the first attribute in the list is the primary ordering attribute, for items that match on this primary attribute, the second attribute is invoked, for matches on the first two attribute values, the third attribute in the ordering list is invoked, etc.).

If one immediately follows an item in the ordering list with the option **DESC** then ordering will be based on descending order for values of that item.. By default the ordering is ascending, but this can be made explicit with the qualifier **ASC**.

Examples: Done in class

5. Functions and calculations in the attribute list

Besides simply using attribute names of existing attributes as headers as column labels for query results, one can insert new entries in the attribute list to perform calculations or to invoke some functions that SQL provides. Here are some of the most commonly used functions:

COUNT	returns the number of values in the returned attribute column
SUM	returns the sum of the values in the returned attribute column
AVG	returns the mean of the values in the returned attribute column
MAX	returns the largest value in the returned attribute column
MIN	returns the smallest value in the returned attribute column

A list of all the function supported in MySQL can be found on the MySQL website at

`dev.mysql.com/doc/refman/8.0/en/dynindex-function.html`

Examples: Done in class

6. Renaming Column Headers

In displaying the results of queries up to this point, SQL uses as column headers the names of the respective attributes as listed in the attribute list component of SELECT (or in the case of *, all of the attribute names for the table). For use as column headers, however, one can replace an attribute name in the attribute list with an alias, subject to the following rules

- The general form is

  old-attribute-name **AS** *alias-name* or simply

  old-attribute-name alias-name

- If the alias contains a space or special character it must be enclosed in double quotes ("*alias-name*" or square brackets [*alias-name*])
- An alias without a space or special character does not require the quotes or square brackets.

Examples: Done in class

7. Data Aggregation – The "Group By" and "Having" Clauses

Some of our examples above used built-in functions such as COUNT, SUM, AVG, MAX, or MIN to retrieve summary data from (selected) tuples taken from an entire relation. By incorporating a GROUP BY clause into the SELECT statement we can carry out these same operations on groups of tuples.

```
SELECT attribute-list
FROM table-name(s)
WHERE condition
GROUP BY column-list
```

Once again, to be meaningful the attribute list must contain one of the "summary" functions SUM, AVG, MAX, MIN, or COUNT. Also, one cannot include in the "attribute-list" column summary names that are not in "column-list".

The effect of the GROUP BY clause is to cause the tuples whose values satisfy the WHERE condition to be grouped into subsets where the values of each tuple in a subset agree on the values from those in the column list. For each of these subsets SELECT will then output a single row of values as prescribed by attribute- list. Any calculations of functions used in the attribute list are carried out on the tuples in each subgroup, not the entire table.

Examples: Done in class

Having taken the step of creating groups of tuples, we can incorporate a form of group-level selection into our queries via the HAVING clause.

```
SELECT attribute-list
FROM table-name(s)
WHERE condition1
GROUP BY column-list
HAVING condition2
```

The HAVING clause condition is applied after the subsets/groups are formed and is applied to the groups. Consequently one must be sure that any attribute references in *condition2* relate to those in *attribute-list*.

Examples: Done in class

Section 5. Database Queries That Use Several Tables

Many queries require that we use data values coming from more than one table. The two most common ways to do this are by join operations, or by subqueries. We will consider joins in this section and subqueries in the Section 6..

Database Queries Involving Products and Equijoins

The simplest way to involve more than one table in a query (or more than one copy of a table) is to use comma-separated list of tables in the **FROM** portion of a **SELECT** statement

```
SELECT attribute-list
FROM table1, table2,...tablen
rest of query
```

This has the effect, logically, of forming the Cartesian product of the tables in the table list, a structure one never uses directly, but which forms the basis for an equijoin, which is very useful.

Example: Cartesian product: illustrated in class

As we noted above, there is never a need to work with a Cartesian product of tables for its own sake (other than to see what one looks like). Rather, the Cartesian product is the first step in forming an *equijoin* of pairs of tables. What we mean here is that in all realistic cases where we use more than one table in a query there are attributes in each table that are naturally paired (typically a foreign key and the primary key it references). What we want to do is to combine (or "join") the tables on these attributes, creating a table that appends the attributes of one table to those of another, but only accepts those tuples whose values agree on these so-called ":join attributes." We specify this latter condition by specifically listing the attribute equality condition(s) in the **WHERE** clause. This means our SELECT statement assumes a form such as the following

```
SELECT attribute-list
FROM table1, table2, …
WHERE (table1.att1 = table2.attA) AND
      (table1.att2 = table2.attB) AND
      other-similar-conditions-if-necessary;
```

Examples: Equijoins: illustrated in class

Attribute name ambiguities will arise in the case where a query uses two copies of the same relation. In this case one can declare one or more single-letter, *table aliases* as shown below