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 of Renaming Column Headers: Done in class

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

Some of our examples have 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 not on the entire set of tuples, but rather on groups of tuples, where the groups are defined by those that have common values across one or more attributes..

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

a. To first select from the given table all of the tuples whose values satisfy the WHERE condition.

b. Next, these tuples are collected into groups. All tuples in a given group agree on the values from those in the *column-list*.

c. Finally, for each of these groups 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 group***, not on the entire set of tuples returned by step a. above.

Examples of Data Aggregation Using Group By: Done in class

One we formed groups of tuples with the GROUP BY clause, we can incorporate a form of group-level selection into our queries via the HAVING clause.

```
SELECT attribute-list
FROM table-name
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 of Data Aggregation Using Group By-Having: Done in class

Section 2. Introduction to Subqueries

Consider the SCO database from Chapter 1. Suppose someone wanted to know those salespeople whose salaries were below the average salary of a salesperson. One might think that a way to proceed is by the following query

```
SELECT name
FROM salesperson
WHERE Salary < AVG(salary)
```

However in attempting to execute this query in MySQL (or any other dbms) and error will arise. In MySQL Workbench one gets the cryptic message "Invalid use of group function."

A moment's reflection may convince you, however, that there is no way this query can be done with one SELECT statement; indeed two SELECTs are required:

1. One SELECT statement to determine first what the average salary is.

2. A second SELECT statement to determine those salespeople whose salaries are below the average salary as determined by the first query.

   This would require, however, that we replace AVG(Salary) in the above select statement with the appropriate **SELECT** statement

   ```
   SELECT AVG(Salary)
   FROM salesperson
   ```

   Fortunately, if we surround the above query with parentheses SQL will indeed allow us to modify the initial query to use this query.

   ```
   SELECT name
   FROM salesperson
   WHERE Salary <
     (SELECT AVG(salary)
      FROM salesperson);
   ```

   Here the SELECT statement appearing in the WHERE clause is known as a *nested query*, or *subquery*.

   Example: In the SCO database, get all details of the salesperson with the highest salary.

   ```
   SELECT *
   FROM salesperson
   WHERE Salary =
     (SELECT MAX(salary)
      FROM salesperson);
   ```

   As we shall see in the next section, subqueries can be used (and misused) in much more general situations than we see here.

Section 3. Database Queries That Use Several Tables -- Multi-table Queries

Many queries require that we use data values coming from more than one table. The two most common ways to do this are by equijoins, or by subqueries.

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 of 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 of 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

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

Example of Table Aliases: Illustrated in class

Subqueries in Multi-Table Queries

An alternative approach to using equijoins in multi-table queries is to use subqueries. There are two main approaches here.

- Use sub queries and the **IN** operator.
- Use subqueries and the **EXISTS** operator.

1. Subqueries and the *IN* Operator:

The **IN** operator tests whether an attribute value matches one in a set of values. In its simplest use, one establishes a condition in a WHERE clause that tests whether an attribute values matches one in a prescribed set of values.

```
SELECT …
FROM …
WHERE att IN (val1, val2,…,valn);
```

Example of IN operator: Illustrated in class

The full power of the IN operator comes out, however when we replace the prescribed set of values with the results of another query

```
SELECT …
FROM table1
WHERE att1 IN
(
  SELECT att2
   FROM table2
   WHERE condition
)
```

Here we assume *att1* and *att2* are attributes that might otherwise be compared in an equijoin condition.

Examples of Subqueries with IN: Illustrated in class

2. Subqueries and the *EXISTS* Operator:

Another important operator in SQL is the **EXISTS** operator, which takes another **SELECT** statement as its argument and returns **true** or **false** depending on whether there is a tuple that satisfies the condition specified in its **WHERE** clause. A common form in which this would arise is

```
SELECT …
FROM table1
WHERE EXISTS
(
  SELECT *
   FROM table2
   WHERE condition
)
```

Usually the condition posed in the nested SELECT statement contains as one of its components and expression of the form

```
att1 = att2
```

You may wonder why someone would ever want to use a subquery with an EXISTS operator when an equijoin seems to work just as well and is more compact. It turns out that there are some queries that cannot be expressed clearly and easily in equijoin format but for which nested queries work much better. We shall not pursue these here, however, as they are best left for presentation and discussion in a database course.

The* Join *Operator and Its Variations

Finally we note that SQL provides a series of operators – the JOIN operators - that provide most of the same functionality as the equijoin in most cases, but also account for the possibility that values of some of the join conditions `att1 = att2` may be **null** for some tuples. This gives rise to such variants as **JOIN**, **INNER JOIN, LEFT OUTER JOIN**, **RIGHT OUTER JOIN**, **FULL OUTER JOIN**, **NATURAL JOIN**, etc. We shall not delve into the differences among these here however.

We give here, however, an example using the SCO database of the INNER JOIN operator (although one can also use simply JOIN). This is arguably the most widely used of the joins.

```
SELECT orders.Number, customer.Name, orders.Amount
FROM orders
     INNER JOIN customer
       ON orders.Customer=customer.Name;
```