DATABASE QUERIES

Section 1. Simple Database Queries in SQL

Up to this point we have only been working with the data definition features of SQL. In this section we shall examine some of the data manipulation (or query) features of SQL. It is these features that will allow us to extract useful information from the data that is stored in the database.

Example: Consider the SALESPEOPLE – CUSTOMER – ORDERS database from previous sections. The following are examples of some of the types of information one might want from this database.

1. Get the names of salespeople who have an order with customer Abernathy.
2. Get the name and age name of salespeople who have an order with customer Abernathy.
3. Get the name and salary of the salespeople who have an order with a customer from Charlotte.
4. List the total sales for each salesperson.
5. List the total sales from all orders with customer Abernathy.
6. List the total sales from all orders with customer Abernathy if the total sales are at least 3000.
7. List the salespeople by age.

SQL has one fundamental data manipulation command, 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! We shall begin with the simplest cases of queries that involve only one table.

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. Such an operation is known as a projection.

   Example:

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 operators AND, OR, NOT.

   Examples:

3. 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

\[ \text{old-attribute-name AS alias-name} \quad \text{or simply} \]
\[ \text{old-attribute-name alias-name} \]

• 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 with a space or special character does not require the quotes or square brackets.

Examples:

4. Functions and calculations in the attribute list

Besides simply providing column headers and an order for displaying values, one can insert entries in the attribute list to perform calculations or to invoke some functions that SQL provides.

Examples:

5. 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 on groups of tuples.

```
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 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:

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
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:
6. 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 into the SELECT clause, however, one can designate that duplicate output values be removed.

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

```

7. 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 clause in a SELECT statement.

```
SELECT attribute-list
FROM table-name
WHERE condition1
GROUP BY column-list
HAVING condition2
ORDER BY ordering-list

```

Here ordering-list is a comma-separated list of attribute names that prescribe a *cascading order sequence* (that is, the first attribute in the list is the primary ordering attribute, for items that match on the dominant attribute, the second attribute is invoked, for matches on the first two attribute values, the third attribute in the ordering list is invoked, etc.).

By default each ordering is in ascending order. This can be changed for a given ordering-list attribute by placing DESC after that attribute’s name.

**Examples:**

---

Section 2. Database Queries That Use Several Tables

Many queries require that we use data values come from more than one table, The two most common ways to do this are by products and joins, or by nested queries.

*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 a table list in the FROM clause of a SELECT statement.

```
SELECT attribute-list
FROM table-list

```

This has the effect, logically, of forming the Cartesian product of the tables in the table list.

**Example:**
There is almost never a need to work with a Cartesian product of tables for its own sake. Rather, the product is the first step in forming an equijoin of pairs of tables. What we mean here is that in almost all cases where we use more than one table in a query there are attributes in each table that are naturally paired (usually a foreign key and the primary key it references). What we want to do is join the tables on these attributes. That is, we want to create a table that appends the attributes of one table to those of another, but only accepts those tuples whose values agree on the “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 (att1 = attA) AND (att2 = attB) AND other-conditions;
```

Examples:

Note, it is quite common two have two attributes with the same name appearing in different tables in a database. This causes an ambiguity if just the attribute name is used, since we don’t know which attribute is being referenced. To resolve such ambiguities we prepend the name of the associated table to an attribute, separating the two with a period.

Examples:

Attribute name ambiguities may also arise in the case where a query needs 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:

Nested Queries

An alternative approach to using equijoins in multi-table queries is to use nested queries, that is a query in which the WHERE clause has a condition that uses the results on another query. There are two main approaches here.

- Use nested queries and the **IN** operator.
- Use nested queries and the **EXISTS** operator.

1. Nested Queries 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. It’s basic form is:

```
SELECT ...
FROM ...
WHERE att IN [val1, val2,...,valn];
```

Example:

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 att IN (SELECT ...
FROM ...
WHERE ...
WHERE att IN [val1, val2,...,valn]
);
```
Here we assume $att1$ and $att2$ are attributes that might otherwise be compared in an equijoin condition.

Examples:

2. *Nested Queries and the EXISTS Operator:*

   An 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

   ```sql
   SELECT ... 
   FROM table1 
   WHERE EXISTS 
   ( 
     SELECT * 
     FROM table2 
     WHERE condition 
   ) 
   ```

   Usually the condition posed in the nested SELECT statement contains as one of its component of the form $att1 = att2$

   Examples:

   You may wonder why someone would ever want to use a nested query 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.
Examples of Queries

The following queries are with respect to the EMPLOYEE-PROJECT-ASSIGNEDTO database

- EMPLOYEE(EmpNum, Name)
- PROJECT(ProjectNum, ProjectName, ChiefArchitect)
- ASSIGNEDTO(ProjectNum, EmpNum)

1. Get the employee number of employees working on project COMP353.

2. Get the employee number and employee name of the employees working on project COMP353.

3. Get the employee number and employee name of the employees working on a "Database" project.

4. Get the employee number and employee name of the employees working on both COMP353 and COMP354.

5. Find the employee numbers of employees who are not working on COMP353.
The following queries are with respect to the following database

\[
\begin{align*}
\text{SPECIES} & (\text{SName}, \text{BColor}, \text{MaxHt}) \\
\text{TREE} & (\text{TreeNum}, \text{SName}, \text{Planted}) \\
\text{MEASUREMENT} & (\text{MeasNum}, \text{TreeNum}, \text{Year}, \text{Month}, \text{Trunk}, \text{Height}, \text{NBranch})
\end{align*}
\]

- **Species** attributes are the species name (\textit{SName}), bark color \textit{BColor}, and maximum height (\textit{MaxHt}). Attribute \textit{sname} is Species primary attribute.
- **Tree** has the attributes tree number (\textit{TreeNum}) and the year it was planted \textit{Planted}). Its primary attributes is \textit{TreeNum}.
- **Measurement** is described by a measurement number (\textit{MeasNum}), \textit{Year} and \textit{Month} performed, trunk size (\textit{Trunk}), \textit{Height}, and the number of branches \textit{NBranch}). The Measurement's primary attribute is \textit{MeasNum}.

1. Get the tree number of any trees planted in 1985 that have reached a height that is at least 80% of its species-attributed maximal height.

2. Get the species name of each species with red bark where at least one tree of that species was planted since 1985 (inclusive).

3. Get the species name of each species with red bark where every recorded tree of that species was planted since 1985 (inclusive).

4. Get the name and bark color of each species with a maximum height exceeding 60 feet where a tree of that species had a measurement since 1970 (inclusive) showing a height greater than 30 feet.