WEEK 5

SQL: The Query Language
Basic SQL Query

- **relation-list**  A list of relation names (possibly with a range-variable after each name).
- **target-list**  A list of attributes of relations in relation-list
- **qualification**  Comparisons (Attr \( \text{op} \) const or Attr1 \( \text{op} \) Attr2, where \( \text{op} \) is one of \(<, >, =, \leq, \geq, \neq\) combined using AND, OR and NOT.
- **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are **not** eliminated!

```sql
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
```

SQL Query

- Upto six clauses in an SQL query, but, only SELECT and FROM are mandatory.

- SELECT  <attribute list>
- FROM  <table list>
- [WHERE  <condition>]
- [GROUP BY  <grouping attribute(s)>]
- [HAVING  <group condition>]
- [ORDER BY  <attribute list>]

**SQL Query**

- **SELECT**: lists the attributes or functions to be retrieved.
- **FROM**: specifies all relations (or aliases) needed in query but not those needed in nested queries.
- **WHERE**: specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause.
- **GROUP BY**: specifies grouping attributes.
- **HAVING**: specifies a condition for:
  1) Selection of attributes given in GROUP BY clause
  2) Arguments to an aggregate operator
- **ORDER BY**: specifies an order for displaying the result of a query (ascending order is the default).

Order of evaluation is WHERE, GROUP BY, HAVING and then SELECT

---

**Conceptual Evaluation Strategy**

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of relation-list.
  - Discard resulting tuples if they fail qualifications.
  - Delete attributes that are not in target-list.
  - If DISTINCT is specified, eliminate duplicate rows.

- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.
Example of Conceptual Evaluation

```
SELECT Sx.sname
FROM S AS Sx, SP AS SPx
WHERE Sx.S# = SPx.S# AND SPx.P# = 'P5'
```

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
</tbody>
</table>

Example. Print an entire “P” relation.
SELECT *
FROM P

A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT Sx.sname
FROM S AS Sx, SP AS SPx
WHERE Sx.S# = SPx.S# AND SPx.P# = 'P5'
```

OR

```
SELECT S.ename
FROM S, SP
WHERE S.S# = SP.S#
AND SP.P# = 'P5'
```

It is good style, however, to use range variables always!
Find S# who supply at least one part

```
SELECT Sx.S#
FROM S AS Sx, SP AS SPx
WHERE Sx.S# = SPx.S#
```

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing S.S# by S.sname in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

```
SELECT Sx.status, status1=Sx.status-5, 2*Sx.status AS status2
FROM S AS Sx
WHERE Sx.sname LIKE 'P_%P'
```

- Illustrates use of arithmetic expressions and string pattern matching: Find triples (of status of suppliers and two fields defined by expressions) for suppliers whose names begin and end with P and contain at least three characters.

- AS and = are two ways to name fields in result.

- LIKE is used for string matching. ‘_’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.
Find S# of suppliers who supply a red or a green part

- **UNION**: Can be used to compute the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries).

- If we replace OR by AND in the first version, what do we get?

```sql
SELECT Sx.S#
FROM S AS Sx, P AS Px, SP AS SPx
WHERE Sx.S#=SPx.S# AND SPx.P#=Px.P#
AND (Px.color='red' OR Px.color='green')
```

```sql
SELECT Sx.S#
FROM S AS Sx, P AS Px, SP AS SPx
WHERE Sx.S#=SPx.S# AND SPx.P#=Px.P#
AND Px.color='red'
UNION
SELECT Sx.S#
FROM S AS Sx, P AS Px, SP AS SPx
WHERE Sx.S#=SPx.S# AND SPx.P#=Px.P#
AND Px.color='green'
```

Find S#’s of suppliers who supply both a red and a green parts

- **INTERSECT**: Can be used to compute the intersection of any two union-compatible sets of tuples.

- Included in the SQL/92 standard, but some systems don’t support it.

- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```sql
SELECT Sx.S#
FROM S AS S1, P AS P1, SP AS SP1,
P AS P2, SP AS SP2
WHERE S1.S#=SP1.S# AND SP1.P#=P1.P#
AND S1.S#=SP2.S# AND SP2.P#=P2.P#
AND (P1.color='red' AND P2.color='green')
```

```sql
SELECT S1.S#
FROM S AS S1, P AS P1, SP AS SP1
WHERE S1.S#=SP1.S# AND SP1.P#=P1.P# AND P1.color='red'
INTERSECT
SELECT S1.S#
FROM S AS S1, P AS P1, SP AS SP1
WHERE S1.S#=SP1.S# AND SP1.P#=P1.P# AND P1.color='green'
```
Nested Queries

Find names of suppliers who supply P# ‘P5’:

\[
\text{SELECT S1.sname} \\
\text{FROM S AS S1} \\
\text{WHERE S1.S# IN (SELECT SP1.S#} \\
\text{FROM SP AS SP1} \\
\text{WHERE SP1.P#='P5')} \\
\]

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- For suppliers who do not supply P# ‘P5’, use NOT IN.
- To understand semantics of nested queries, think of a nested loops evaluation: For each Supplier tuple, check the qualification by computing the subquery.

Nested Queries with Correlation

Find names of suppliers who supply part# ‘P5’:

\[
\text{SELECT S.sname} \\
\text{FROM S AS S1} \\
\text{WHERE EXISTS (SELECT *} \\
\text{FROM SP AS SP1} \\
\text{WHERE SP1.P#='P5' AND S1.S#=SP1.S#)} \\
\]

- EXISTS is another set comparison operator, like IN.
- If UNIQUE is used, and * is replaced by SP.P#, finds non-duplicate tuples for suppliers who supply part# ‘P5’. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by SP.P#)
- Illustrates why, in general, subquery must be re-computed for each Supplier tuple
Nested Queries with Correlation

Find names of suppliers who do not supply P5.

```sql
SELECT Sx.Sname
FROM S AS Sx
WHERE NOT EXISTS (SELECT *
FROM SP AS SPx
WHERE SPx.S# = Sx.S#
AND SPx.P# = 'P5')
```

More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: op ANY, op ALL, op IN, >, <, =, ≥, ≤, ≠
- Find suppliers whose status is greater than that of some other supplier called Jones:

```sql
SELECT *
FROM S AS S1
WHERE S1.status > ANY (SELECT S2.status
FROM S AS S2
WHERE S2.sname='Jones')
```
Rewriting INTERSECT Queries Using IN

Find S#’s of suppliers who supply both a red and a green part:

```sql
SELECT S.S#
FROM S AS S1, P AS P1, SP AS SP1
WHERE S1.S# = SP1.S# AND SP1.P# = P1.P# AND P1.color = 'red'
    AND S1.S# IN (SELECT S2.S#
                   FROM S AS S2, P AS P2, SP AS SP2
                   WHERE S2.S# = SP2.S# AND SP2.P# = P2.P#
                     AND P2.color = 'green')
```

- To find names (not S#’s) of Suppliers who supply both red and green parts, just replace S.S# by S.sname in SELECT clause. (What about INTERSECT query?)

Division in SQL

Find suppliers who supply all parts.

```sql
SELECT S.sname
FROM S AS Sx
WHERE NOT EXISTS (SELECT Px.P#
                   FROM P AS Px
                   WHERE NOT EXISTS (SELECT SPx.P#
                                       FROM SP AS SPx
                                       WHERE SPx.P# = Px.P#
                                         AND SPx.S# = Sx.S#))
```

Supplier Sx such that there is no Part Px without an SP tuple showing Sx supplies Px
Aggregate Operators

- Significant extension of relational algebra.

**COUNT** (*

**SUM** ([DISTINCT] A)

**AVG** ([DISTINCT] A)

**MAX** (A)

**MIN** (A)

```
SELECT COUNT(*)
FROM S AS Sx

SELECT COUNT ([DISTINCT] A)
FROM S AS Sx

SELECT SUM ([DISTINCT] A)
FROM S AS Sx

SELECT AVG ([DISTINCT] A)
FROM S AS Sx

WHERE Sx.city='London'

SELECT COUNT ([DISTINCT] Sx.status)
FROM S AS Sx

WHERE Sx.sname='Bob'

SELECT AVG ([DISTINCT] Sx.status)
FROM S AS Sx

WHERE Sx.status=10
```

---

**Ex:** Find the number of shipments for part P2

```
SELECT COUNT(*)
FROM SP
WHERE P# = 'P2'
```

**Ex:** Find the total quantity of part P2 supplied

```
SELECT SUM (Qty)
FROM SP
WHERE P# = 'P2'
```
Find name and status of the supplier(s) with the highest status.

- The first query is illegal! (We’ll look into the reason a bit later, when we discuss Group By.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

\[
\begin{align*}
\text{SELECT} & \quad \text{Sx.sname, MAX (Sx.status)} \\
\text{FROM} & \quad \text{S} \text{ AS Sx}
\end{align*}
\]

So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several groups of tuples.

- Consider: Find the status of the supplier with the lowest status for each city.
  - In general, we don’t know how many cities exist.
  - Suppose we know that there are 10 cities; we can write 10 queries that look like this (!):

\[
\begin{align*}
\text{SELECT} & \quad \text{MIN (Sx.city)} \\
<\text{For } i = 1,2,\ldots,10: \text{city}(i)> & \quad \text{FROM} \quad \text{S} \text{ AS Sx} \\
\text{WHERE} & \quad \text{Sx.city} = \text{city}(i)
\end{align*}
\]
Queries With GROUP BY and HAVING

SELECT (DISTINCT) target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification

- The target-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (S.status)).
  - The attribute list (i) must be a subset of grouping-list. Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group. (A group is a set of tuples that have the same value for all attributes in grouping-list.)

Conceptual Evaluation

- The cross-product of relation-list is computed, tuples that fail qualification are discarded, `unnecessary` fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- The group-qualification (Having) is then applied to eliminate some groups. Expressions in group-qualification must have a single value per group!
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.
**Queries With GROUP BY**

Find P# and the total shipment quantity for the part.

```
SELECT P#, SUM (Qty)
FROM SP
GROUP BY P#
```

**Get status of the supplier with the smallest status ≥ 10, for each city that has at least 2 such suppliers.**

```
SELECT S.City, MIN (S.Status) AS Minstat
FROM S AS Sx
WHERE Sx.Status >= 10
GROUP BY Sx.City
HAVING COUNT (*) > 1
```

- Only City and Status are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes ‘unnecessary’.

<table>
<thead>
<tr>
<th>CITY</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>22</td>
</tr>
<tr>
<td>Paris</td>
<td>10</td>
</tr>
<tr>
<td>Paris</td>
<td>30</td>
</tr>
<tr>
<td>London</td>
<td>20</td>
</tr>
<tr>
<td>Athens</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CITY</th>
<th>Minstat</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>20</td>
</tr>
<tr>
<td>Paris</td>
<td>10</td>
</tr>
</tbody>
</table>
Get status of the supplier with the smallest status > 10, for each city that has at least 2 such suppliers.

```
SELECT  S1.City, MIN (S1.status)
FROM    S AS S1
WHERE   S1.status > 10
GROUP BY S1.City
HAVING  1  < (SELECT  COUNT (*)
               FROM    S AS S2
               WHERE   S1.City=S2.City)
```

- Shows HAVING clause can also contain a subquery.

For each red part, find the number of suppliers for this part (Assume S, P, SPJ)

```
SELECT  P.P#, COUNT (*) AS Scount
FROM    SPJ AS SPJx, P AS Px, S AS Sx
WHERE   Sx.S#=SPJx.S# AND SPJx.P#=Px.P# AND Px.color='red'
GROUP BY P.P#
```

- Grouping over a join of three relations.
- What do we get if we remove P.color='red' from the WHERE clause and add a HAVING clause with this condition?
- What if we drop suppliers and the condition involving S.S#?
Find those cities for which the average status is the minimum among all other cities

- Aggregate operations cannot be nested! **WRONG:**

```
SELECT S1.city
FROM S AS S1
WHERE S1.status = (SELECT MIN (AVG (S2.status)) FROM S AS S2)
```

- **Correct solution (in SQL/92):**

```
SELECT Temp.city, Temp.avgstatus
FROM (SELECT Sx.city, AVG (Sx.status) AS avgstatus
      FROM S AS Sx
      GROUP BY Sx.city) AS Temp
WHERE Temp.avgstatus = (SELECT MIN (Temp.avgstatus)
                           FROM Temp)
```

**Null Values**

- Field values in a tuple are sometimes **unknown** (e.g., a status has not been assigned) or **inapplicable** (e.g., no spouse's name).
  - SQL provides a special value `null` for such situations.

- **The presence of null complicates many issues. E.g.**:
  - Special operators needed to check if value is/is not null.
  - Is status>8 true or false when status is equal to null? What about `AND`, `OR` and `NOT` connectives?
  - We need a **3-valued logic** (true, false and unknown).
  - Meaning of constructs must be defined carefully. (e.g., `WHERE` clause eliminates rows that don’t evaluate to true.)
  - New operators (in particular, **outer joins**) possible/needed.