Weeks 3 & 4: SQL

The SQL Query Language

Select Statements

Joins, Aggregate and Nested Queries

Insertions, Deletions and Updates

Assertions, Views, Triggers and Access Control

SQL as a Query Language

→ SQL expresses queries in declarative way — queries specify the properties of the result, not the way to obtain it.

→ Queries are translated by the query optimizer into the procedural language internal to the DBMS.

→ The programmer focuses on readability, not on efficiency.

SQL Queries

→ SQL queries are expressed by the select statement.

→ Syntax:

\[
\text{select } \text{AttrExpr [as Alias ] } \{, \text{AttrExpr [as Alias ] } \}
\text{from } \text{Table [as Alias ] } \{, \text{[as Alias ] } \}
\text{[ where Condition ]}
\]

→ The three parts of the query are usually called: target list, from clause, where clause.

→ The query first builds the Cartesian product of the tables in the from clause, then selects only the rows that satisfy the condition in the where clause and for each row evaluates the attribute expressions in the target list.

Example Database

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
<th>FirstName</th>
<th>Surname</th>
<th>Dept</th>
<th>Office</th>
<th>Salary</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>Administration</td>
<td>10</td>
<td>45</td>
<td>London</td>
<td></td>
</tr>
<tr>
<td>Charles</td>
<td>White</td>
<td>Production</td>
<td>20</td>
<td>36</td>
<td>Toulouse</td>
<td></td>
</tr>
<tr>
<td>Gus</td>
<td>Green</td>
<td>Administration</td>
<td>20</td>
<td>40</td>
<td>Oxford</td>
<td></td>
</tr>
<tr>
<td>Jackson</td>
<td>Neri</td>
<td>Distribution</td>
<td>16</td>
<td>45</td>
<td>Dover</td>
<td></td>
</tr>
<tr>
<td>Charles</td>
<td>Brown</td>
<td>Planning</td>
<td>14</td>
<td>80</td>
<td>London</td>
<td></td>
</tr>
<tr>
<td>Laurence</td>
<td>Chen</td>
<td>Planning</td>
<td>7</td>
<td>73</td>
<td>Worthing</td>
<td></td>
</tr>
<tr>
<td>Pauline</td>
<td>Bradshaw</td>
<td>Administration</td>
<td>75</td>
<td>40</td>
<td>Brighton</td>
<td></td>
</tr>
<tr>
<td>Alice</td>
<td>Jackson</td>
<td>Production</td>
<td>20</td>
<td>46</td>
<td>Toulouse</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>DeptName</th>
<th>Address</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>Bond Street</td>
<td>London</td>
<td></td>
</tr>
<tr>
<td>Production</td>
<td>Rue Victor Hugo</td>
<td>Toulouse</td>
<td></td>
</tr>
<tr>
<td>Distribution</td>
<td>Pond Road</td>
<td>Brighton</td>
<td></td>
</tr>
<tr>
<td>Planning</td>
<td>Bond Street</td>
<td>London</td>
<td></td>
</tr>
<tr>
<td>Research</td>
<td>Sunset Street</td>
<td>San José</td>
<td></td>
</tr>
</tbody>
</table>
**Simple SQL Query**

→ "Find the salaries of employees named Brown":
   
   ```sql
   select Salary as Remuneration
   from Employee
   where Surname = 'Brown'
   ```

→ Result:

<table>
<thead>
<tr>
<th>Remuneration</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
</tr>
<tr>
<td>80</td>
</tr>
</tbody>
</table>

*** in the Target List**

→ "Find all the information relating to employees named Brown":
   
   ```sql
   select *
   from Employee
   where Surname = 'Brown'
   ```

→ Result:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
<th>Dept</th>
<th>Office</th>
<th>Salary</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>Administration</td>
<td>10</td>
<td>45</td>
<td>London</td>
</tr>
<tr>
<td>Charles</td>
<td>Brown</td>
<td>Planning</td>
<td>14</td>
<td>80</td>
<td>London</td>
</tr>
</tbody>
</table>

**Attribute Expressions**

→ Find the monthly salary of the employees named White:
   
   ```sql
   select Salary / 12 as MonthlySalary
   from Employee
   where Surname = 'White'
   ```

→ Result:

<table>
<thead>
<tr>
<th>MonthlySalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.00</td>
</tr>
</tbody>
</table>

**Simple Join Query**

→ "Find the names of employees and their cities of work":
   
   ```sql
   select Employee.FirstName, Employee.Surname, Department.City
   from Employee, Department
   where Employee.Dept = Department.DeptName
   ```

→ Result:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>London</td>
</tr>
<tr>
<td>Charles</td>
<td>White</td>
<td>Toulouse</td>
</tr>
<tr>
<td>Gus</td>
<td>Green</td>
<td>London</td>
</tr>
<tr>
<td>Jackson</td>
<td>Neri</td>
<td>Brighton</td>
</tr>
<tr>
<td>Charles</td>
<td>Brown</td>
<td>London</td>
</tr>
<tr>
<td>Laurence</td>
<td>Chen</td>
<td>London</td>
</tr>
<tr>
<td>Pauline</td>
<td>Bradshaw</td>
<td>London</td>
</tr>
<tr>
<td>Alice</td>
<td>Jackson</td>
<td>Toulouse</td>
</tr>
</tbody>
</table>
Table Aliases

"Find the names of employees and the cities where they work" (using an alias):

```sql
select FirstName, Surname, D.City
from Employee, Department D
where Dept = DeptName
```

Result:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>London</td>
</tr>
<tr>
<td>Charles</td>
<td>White</td>
<td>London</td>
</tr>
<tr>
<td>Gus</td>
<td>Green</td>
<td>London</td>
</tr>
<tr>
<td>Jackson</td>
<td>Neri</td>
<td>Brighton</td>
</tr>
<tr>
<td>Charles</td>
<td>Brown</td>
<td>London</td>
</tr>
<tr>
<td>Laurence</td>
<td>Chen</td>
<td>London</td>
</tr>
<tr>
<td>Pauline</td>
<td>Bradshaw</td>
<td>London</td>
</tr>
<tr>
<td>Alice</td>
<td>Jackson</td>
<td>Toulouse</td>
</tr>
</tbody>
</table>

Predicate Conjunction

"Find the first names and surnames of employees who work in office number 20 of the Administration department":

```sql
select FirstName, Surname
from Employee
where Office = '20' and Dept = 'Administration'
```

Result:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gus</td>
<td>Green</td>
</tr>
</tbody>
</table>

Predicate Disjunction

"Find the first names and surnames of employees who work in either the Administration or the Production department":

```sql
select FirstName, Surname
from Employee
where Dept = 'Administration' or Dept = 'Production'
```

Result:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
</tr>
<tr>
<td>Charles</td>
<td>White</td>
</tr>
<tr>
<td>Gus</td>
<td>Green</td>
</tr>
<tr>
<td>Pauline</td>
<td>Bradshaw</td>
</tr>
<tr>
<td>Alice</td>
<td>Jackson</td>
</tr>
</tbody>
</table>

Complex Logical Expressions

"Find the first names of employees named Brown who work in the Administration department or the Production department":

```sql
select FirstName
from Employee
where Surname = 'Brown' and (Dept = 'Administration' or Dept = 'Production')
```

Result:

<table>
<thead>
<tr>
<th>FirstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
</tr>
</tbody>
</table>
**Operator like**

→ "Find employees with surnames that have ‘r’ as the second letter and end in ‘n’":

```sql
select *
from Employee
where Surname like '_r%n'
```

→ Result:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
<th>Dept</th>
<th>Office</th>
<th>Salary</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>Administration</td>
<td>10</td>
<td>45</td>
<td>London</td>
</tr>
<tr>
<td>Gus</td>
<td>Green</td>
<td>Administration</td>
<td>20</td>
<td>40</td>
<td>Oxford</td>
</tr>
<tr>
<td>Charles</td>
<td>Brown</td>
<td>Planning</td>
<td>14</td>
<td>80</td>
<td>London</td>
</tr>
</tbody>
</table>

**Management of Null Values**

→ Null values may mean that:
  - a value is not applicable
  - a value is applicable but unknown
  - it is unknown if a value is applicable or not

→ SQL-89 uses a two-valued logic
  - a comparison with `null` returns FALSE

→ SQL-2 uses a three-valued logic
  - a comparison with `null` returns UNKNOWN

→ To test for null values:

```
Attribute is [ not ] null
```

**Algebraic Interpretation of SQL Queries**

→ The generic query:

```sql
select T1.Attr1, ..., Th.Attrhm
from Table1 T1, ..., Tablen Tn
where Condition
```

corresponds to the relational algebra query:

```
π_{T1.Attr1, ..., Th.Attrhm}(σ_{Condition}(T1 × ... × Tablen))
```

**Duplicates**

→ In the relational algebra and calculus the results of queries do not contain duplicates.
→ In SQL, tables may have identical rows.

→ Duplicates can be removed using the keyword `distinct`:

```sql
select City
from Department
```

```
City
London
Toulouse
Brighton
```

```sql
select distinct City
from Department
```

```
City
London
Toulouse
Brighton
San José
San José
```
Joins in SQL-2
SQL-2 introduced an alternative syntax for the representation of joins, representing them explicitly in the from clause:

```sql
select AttrExpr [[ as ] Alias ] {, AttrExpr [[ as ] Alias } 
{[Jojoin] join Table } 
[[ as ] Alias] on JoinConditions } 
[ where OtherCondition ]
```

- `JoinType` can be any of `inner`, `right [outer]`, `left [outer] or full [outer]`.
- The keyword `natural` may precede `JoinType` (rarely implemented).

Inner Join in SQL-2
"Find the names of the employees and the cities in which they work":

```sql
select FirstName, Surname, D.City 
from Employee inner join Department as D on Dept = DeptName
```

Result:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>London</td>
</tr>
<tr>
<td>Charles</td>
<td>White</td>
<td>Toulouse</td>
</tr>
<tr>
<td>Gus</td>
<td>Green</td>
<td>London</td>
</tr>
<tr>
<td>Jackson</td>
<td>Neri</td>
<td>Brighton</td>
</tr>
<tr>
<td>Charles</td>
<td>Brown</td>
<td>London</td>
</tr>
<tr>
<td>Laurence</td>
<td>Chen</td>
<td>London</td>
</tr>
<tr>
<td>Pauline</td>
<td>Bradshaw</td>
<td>London</td>
</tr>
<tr>
<td>Alice</td>
<td>Jackson</td>
<td>Toulouse</td>
</tr>
</tbody>
</table>

Another Example: Drivers and Cars

<table>
<thead>
<tr>
<th>DRIVER</th>
<th>FirstName</th>
<th>Surname</th>
<th>DriverID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>VR 2030020Y</td>
<td></td>
</tr>
<tr>
<td>Charles</td>
<td>White</td>
<td>PZ 1012436B</td>
<td></td>
</tr>
<tr>
<td>Marco</td>
<td>Neri</td>
<td>AP 4544442R</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AUTOMOBILE</th>
<th>CarRegNo</th>
<th>Make</th>
<th>Model</th>
<th>DriverID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC 123</td>
<td>BMW</td>
<td>323</td>
<td>VR 2030020Y</td>
<td></td>
</tr>
<tr>
<td>DEF 456</td>
<td>BMW</td>
<td>Z3</td>
<td>VR 2030020Y</td>
<td></td>
</tr>
<tr>
<td>GHI 789</td>
<td>Lancia</td>
<td>Delta</td>
<td>PZ 1012436B</td>
<td></td>
</tr>
<tr>
<td>BBB 421</td>
<td>BMW</td>
<td>316</td>
<td>MI 2020030U</td>
<td></td>
</tr>
</tbody>
</table>

Left Join
"Find all drivers and their cars, if any":

```sql
select FirstName, Surname, 
Driver.DriverID, CarRegNo, Make, Model 
from Driver left join Automobile on 
(Driver.DriverID = Automobile.DriverID)
```

Result:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
<th>DriverID</th>
<th>CarRegNo</th>
<th>Make</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>VR 2030020Y</td>
<td>ABC 123</td>
<td>BMW</td>
<td>323</td>
</tr>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>VR 2030020Y</td>
<td>DEF 456</td>
<td>BMW</td>
<td>Z3</td>
</tr>
<tr>
<td>Charles</td>
<td>White</td>
<td>PZ 1012436B</td>
<td>GHI 789</td>
<td>Lancia</td>
<td>Delta</td>
</tr>
<tr>
<td>Marco</td>
<td>Neri</td>
<td>AP 4544442R</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
### Full Join

"Find all possible drivers and their cars":

```sql
select
   FirstName, Surname, Driver.DriverID, CarRegNo, Make, Model
from Driver full join Automobile on
   (Driver.DriverID = Automobile.DriverID)
```

**Result:**

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
<th>DriverID</th>
<th>CarRegNo</th>
<th>Make</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>VR 2030020Y</td>
<td>ABC 123</td>
<td>BMW</td>
<td>323</td>
</tr>
<tr>
<td>Mary</td>
<td>Brown</td>
<td>VR 2030020Y</td>
<td>DEF 456</td>
<td>BMW</td>
<td>Z3</td>
</tr>
<tr>
<td>Charles</td>
<td>White</td>
<td>PZ 1012436B</td>
<td>GHI 789</td>
<td>Lancia</td>
<td>Delta</td>
</tr>
<tr>
<td>Marco</td>
<td>Neri</td>
<td>AP 4544442R</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>BBB 421</td>
<td>BMW</td>
<td>316</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>MI 2020030U</td>
<td>BMW</td>
<td>316</td>
</tr>
</tbody>
</table>

### Table Variables

Table aliases may be interpreted as table variables. These correspond to the renaming operator \( \rho \).

"Find all first names and surnames of employees who have the same surname and different first names with someone in the Administration department":

```sql
select E1.FirstName, E1.Surname
from Employee E1, Employee E2
where E1.Surname = E2.Surname and
   E1.FirstName <> E2.FirstName and
   E2.Dept = 'Administration'
```

**Result:**

<table>
<thead>
<tr>
<th>FirstName</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charles</td>
<td>Brown</td>
</tr>
</tbody>
</table>

### The order by Clause

`order by` — appearing at the end of a query — orders the rows of the result; syntax:

```sql
order by OrderingAttribute [ asc | desc ]
   {, OrderingAttribute [ asc | desc ]}
```

- Extract the content of the `Automobile` table in descending order with respect to make and model:

```sql
select *
from Automobile
order by Make desc, Model desc
```

**Result:**

<table>
<thead>
<tr>
<th>CarRegNo</th>
<th>Make</th>
<th>Model</th>
<th>DriverID</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHI 789</td>
<td>Lancia</td>
<td>Delta</td>
<td>PZ 1012436B</td>
</tr>
<tr>
<td>DEF 456</td>
<td>BMW</td>
<td>Z3</td>
<td>VR 2030020Y</td>
</tr>
<tr>
<td>ABC 123</td>
<td>BMW</td>
<td>323</td>
<td>VR 2030020Y</td>
</tr>
<tr>
<td>BBB 421</td>
<td>BMW</td>
<td>316</td>
<td>MI 2020030U</td>
</tr>
</tbody>
</table>

### Aggregate Queries

- Aggregate queries cannot be represented in relational algebra.
- The result of an aggregate query depends on functions that take as an argument a set of tuples.

- SQL-2 offers five aggregate operators:
  - `count`
  - `sum`
  - `max`
  - `min`
  - `avg`
**Operator count**

- `count` returns the number of elements (or, distinct elements) of its argument:
  
  ```sql
  count(*)
  ```

- "Find the number of employees":
  ```sql
  select count(*) from Employee
  ```

- "Find the number of different values on attribute Salary for all tuples in Employee":
  ```sql
  select count(distinct Salary) from Employee
  ```

- "Find the number of tuples in Employee having non-null values on the attribute Salary":
  ```sql
  select count(all Salary) from Employee
  ```

**Sum, Average, Maximum and Minimum**

- Syntax:
  ```sql
  <sum|max|min|avg> ([distinct|all] AttributeExpr)
  ```

- "Find the sum of all salaries for the Administration department":
  ```sql
  select sum(Salary) as SumSalary from Employee where Dept = 'Administration'
  ```

- Result:
  ```markdown
<table>
<thead>
<tr>
<th>SumSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
</tr>
</tbody>
</table>
  ```

**Aggregate Queries with Join**

- "Find the maximum salary among the employees who work in a department based in London":
  ```sql
  select max(Salary) as MaxLondonSal from Employee, Department
  where Dept = DeptName and Department.City = 'London'
  ```

- Result:
  ```markdown
<table>
<thead>
<tr>
<th>MaxLondonSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
</tr>
</tbody>
</table>
  ```

**Aggregate Queries and Target List**

- Incorrect query:
  ```sql
  select FirstName, Surname, max(Salary) from Employee, Department
  where Dept = DeptName and Department.City = 'London'
  ```

  (Whose name? The target list must be homogeneous!)

- Find the maximum and minimum salaries among all employees:
  ```sql
  select max(Salary) as MaxSal, min(Salary) as MinSal from Employee
  ```

- Result:
  ```markdown
<table>
<thead>
<tr>
<th>MaxSal</th>
<th>MinSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>36</td>
</tr>
</tbody>
</table>
  ```
Group by Queries

- Queries may apply aggregate operators to subsets of rows.
- "Find the sum of salaries of all the employees of the same department":
  
  ```sql
  select Dept, sum(Salary) as TotSal
  from Employee
  group by Dept
  
  Result:
<table>
<thead>
<tr>
<th>Dept</th>
<th>TotSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>125</td>
</tr>
<tr>
<td>Distribution</td>
<td>45</td>
</tr>
<tr>
<td>Planning</td>
<td>153</td>
</tr>
<tr>
<td>Production</td>
<td>82</td>
</tr>
</tbody>
</table>
  ```

Semantics of group by Queries - I

- First, the query is executed without `group by` and without aggregate operators:
  
  ```sql
  select Dept, Salary
  from Employee
  
  Result:
<table>
<thead>
<tr>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>45</td>
</tr>
<tr>
<td>Production</td>
<td>36</td>
</tr>
<tr>
<td>Administration</td>
<td>40</td>
</tr>
<tr>
<td>Distribution</td>
<td>45</td>
</tr>
<tr>
<td>Planning</td>
<td>80</td>
</tr>
<tr>
<td>Planning</td>
<td>73</td>
</tr>
<tr>
<td>Administration</td>
<td>40</td>
</tr>
<tr>
<td>Production</td>
<td>46</td>
</tr>
</tbody>
</table>
  ```

- Then the query result is divided in subsets characterized by the same values for the attributes appearing as argument of the `group by` clause (in this case attribute Dept):
  
  ```sql
  select Dept, Salary
  from Employee
  group by Dept
  
  Result:
<table>
<thead>
<tr>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>45</td>
</tr>
<tr>
<td>Administration</td>
<td>40</td>
</tr>
<tr>
<td>Administration</td>
<td>40</td>
</tr>
<tr>
<td>Distribution</td>
<td>45</td>
</tr>
<tr>
<td>Planning</td>
<td>80</td>
</tr>
<tr>
<td>Planning</td>
<td>73</td>
</tr>
<tr>
<td>Production</td>
<td>46</td>
</tr>
</tbody>
</table>
  ```

- Finally, the aggregate operator is applied separately to each subset
  
  ```sql
  select Dept, Salary
  from Employee
  group by Dept
  
  Result:
<table>
<thead>
<tr>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>125</td>
</tr>
<tr>
<td>Distribution</td>
<td>45</td>
</tr>
<tr>
<td>Planning</td>
<td>153</td>
</tr>
<tr>
<td>Production</td>
<td>82</td>
</tr>
</tbody>
</table>
  ```

Semantics of group by Queries - II

- Incorrect query:
  ```sql
  select Office from Employee
  group by Dept
  ```

- Incorrect query:
  ```sql
  select DeptName, count(*), D.City
  from Employee E join Department D
  on (E.Dept = D.DeptName)
  group by DeptName
  ```

- Correct query:
  ```sql
  select DeptName, count(*), D.City
  from Employee E join Department D
  on (E.Dept = D.DeptName)
  group by DeptName, D.City
  ```

Group by Queries and Target List

- Incorrect query:
  ```sql
  select Office from Employee
  group by Dept
  ```

- Incorrect query:
  ```sql
  select DeptName, count(*), D.City
  from Employee E join Department D
  on (E.Dept = D.DeptName)
  group by DeptName
  ```

- Correct query:
  ```sql
  select DeptName, count(*), D.City
  from Employee E join Department D
  on (E.Dept = D.DeptName)
  group by DeptName, D.City
  ```
Group Predicates

- When conditions are defined on the result of an aggregate operator, it is necessary to use the **having** clause.
- "Find which departments spend more than 100 on salaries":
  ```sql
  select Dept
  from Employee
  group by Dept
  having sum(Salary) > 100
  ```

  **Result:**
  - Administration
  - Planning

where or having?

- Only predicates containing aggregate operators should appear in the argument of the **having** clause.
- "Find the departments where the average salary of employees working in office number 20 is higher than 25":
  ```sql
  select Dept
  from Employee
  where Office = '20'
  group by Dept
  having avg(Salary) > 25
  ```

Syntax of an SQL Query

... so far!

- Considering all clauses discussed so far, the syntax of an SQL query is:

  ```sql
  select TargetList
  from TableList
  [where Condition ]
  [group by GroupingAttributeList ]
  [having AggregateCondition ]
  [order by OrderingAttributeList ]
  ```

Set Queries

- A single select statement cannot represent any set operation.
- Syntax:
  ```sql
  SelectSQL { <union | intersect | except >
  [all] SelectSQL }
  ```

- "Find all first names and surnames of employees":
  ```sql
  select FirstName as Name from Employee
  union
  select Surname as Name from Employee
  ```

- Duplicates are removed (unless the all option is used).
Intersection

→ "Find surnames of employees that are also first names":

\[
\text{select FirstName as Name from Employee intersect select Surname as Name from Employee (equivalent to: select E1.FirstName as Name from Employee E1, Employee E2 where E1.FirstName = E2.Surname )}
\]

Difference

→ "Find the surnames of employees that are not first names":

\[
\text{select Surname as Name from Employee except select FirstName as Name from Employee (Can also be represented with a nested query (see later.) )}
\]

Nested Queries

→ A where clause may include predicates that:

- Compare an attribute (or attribute expression) with the result of an SQL query;
  syntax: ScalarValue Op <any | all> SelectSQL any – the predicate is true if at least one row returned by SelectSQL satisfies the comparison all – predicate is true if all rows satisfy comparison;
- Use the existential quantifier on an SQL query;
  syntax: exists SelectSQL the predicate is true if SelectSQL is non-empty.

→ The query appearing in the where clause is called a nested query.

Simple Nested Query

→ "Find the employees who work in departments in London":

\[
\text{select FirstName, Surname from Employee where Dept = any (select DeptName from Department where City = 'London') (Equivalent to: select FirstName, Surname from Employee, Department D where Dept = DeptName and D.City = 'London')}
\]
...Another...

"Find employees of the Planning department, having the same first name as a member of the Production department":
✓ (with a nested query)

\[
\text{select FirstName, Surname from Employee}
\]
\[
\text{where Dept = 'Plan' and FirstName = any}
\]
\[
\text{(select FirstName from Employee}
\]
\[
\text{where Dept = 'Prod')}
\]
✓ (without nested query)

\[
\text{select E1.FirstName, E1.Surname}
\]
\[
\text{from Employee E1, Employee E2}
\]
\[
\text{where E1.FirstName=E2.FirstName and}
\]
\[
\text{E2.Dept='Prod' and E1.Dept='Plan'}
\]

Negation with Nested Queries

"Find departments where there is no one named Brown":

\[
\text{select DeptName}
\]
\[
\text{from Department}
\]
\[
\text{where DeptName <> all (select Dept from Employee}
\]
\[
\text{where Surname = 'Brown')}
\]
✓ (Alternatively:)

\[
\text{select DeptName from Department}
\]
\[
\text{except}
\]
\[
\text{select Dept from Employee}
\]
\[
\text{where Surname = 'Brown'}
\]

Operators in and not in

Operator in is a shorthand for = any

\[
\text{select FirstName, Surname from Employee}
\]
\[
\text{where Dept in (select DeptName}
\]
\[
\text{from Department}
\]
\[
\text{where City = 'London')}
\]
Operator not in is a shorthand for <> all

\[
\text{select DeptName}
\]
\[
\text{from Department}
\]
\[
\text{where DeptName not in}
\]
\[
\text{(select Dept from Employee}
\]
\[
\text{where Surname = 'Brown')}
\]

max and min within a Nested Query

Queries using the aggregate operators max and min can be expressed with nested queries

"Find the department of the employee earning the highest salary":
✓ with max:

\[
\text{select Dept from Employee}
\]
\[
\text{where Salary in (select max(Salary}
\]
\[
\text{from Employee)}
\]
✓ with a nested query:

\[
\text{select Dept from Employee}
\]
\[
\text{where Salary >= all (select Salary}
\]
\[
\text{from Employee)}
\]
A Complex Nested Query

- A nested query may use variables of the outer query ('transfer of bindings').
- Semantics: the nested query is evaluated for each row of the outer query.
- "Find all persons who have the same first name and surname with someone else ("synonyms"), but different tax codes":
  
  ```sql
  SELECT * FROM Person P
  WHERE EXISTS (SELECT * FROM Person P1
               WHERE P1.FirstName = P.FirstName
               AND P1.Surname = P.Surname
  ```

...Another...

- "Find all persons who have no synonyms":
  ```sql
  SELECT * FROM Person P
  WHERE NOT EXISTS (SELECT * FROM Person P1
                    WHERE P1.FirstName = P.FirstName
                    AND P1.Surname = P.Surname
  ```

Tuple Constructors

- The comparison within a nested query may involve several attributes bundled into a tuple.
- A tuple constructor is represented in terms of a pair of angle brackets.
- The previous query can also be expressed as:
  ```sql
  SELECT * FROM Person P
  WHERE <FirstName,Surname> NOT IN (SELECT FirstName,Surname
                                     FROM Person P1
  ```

Comments on Nested Queries

- The use of nested queries may produce less declarative queries, but often results in improved readability.
- Complex queries can become very difficult to understand.
- The use of variables must respect scoping conventions: a variable can be used only within the query where it is defined, or within a query that is recursively nested in the query where it is defined.
Scope of Variables

- Incorrect query:
  
  ```sql
  select * from Employee
  where Dept in
    (select DeptName from Department D1
     where DeptName = 'Production') or
  Dept in (select DeptName
           from Department D2
            where D2.City = D1.City)
  ```

- What's wrong?

Data Modification in SQL

- Modification statements include:
  - Insertions (insert);
  - Deletions (delete);
  - Updates of attribute values (update).

- All modification statements operate on a set of tuples (no duplicates.)

- In the condition part of an update statement it is possible to access other relations.

Insertions

- Syntax:
  ```sql
  insert into TableName [(AttributeList)]
    < values (ListOfValues) | SelectSQL >
  ```

- Using values:
  ```sql
  insert into Department(DeptName,City)
    values('Production','Toulouse')
  ```

- Using a subquery:
  ```sql
  insert into LondonProducts
    (select Code, Description
     from Product
      where ProdArea = 'London')
  ```

Notes on Insertions

- The ordering of attributes (if present) and of values is meaningful -- first value for the first attribute, etc.

- If `AttributeList` is omitted, all the relation attributes are considered, in the order they appear in the table definition.

- If `AttributeList` does not contain all the relation attributes, left-out attributes are assigned default values (if defined) or the null value.
Deletions

→ Syntax:
  
  ```sql
  delete from TableName [where Condition ]
  ```

→ "Remove the Production department":
  ```sql
  delete from Department
  where DeptName = 'Production'
  ```

→ "Remove departments with no employees":
  ```sql
  delete from Department
  where DeptName not in
  (select Dept from Employee)
  ```

Notes on Deletions

→ The delete statement removes from a table all tuples that satisfy a condition.

→ The removal may produce deletions from other tables — if a referential integrity constraint with cascade policy has been defined.

→ If the where clause is omitted, delete removes all tuples. For example, to remove all tuples from Department (keeping the table schema):
  ```sql
  delete from Department
  ```

→ To remove table Department completely (content and schema):
  ```sql
  drop table Department cascade
  ```

Updates

→ Syntax:
  ```sql
  update TableName
  set Attribute = < Expression | SelectSQL | null | default >
  {[ where Condition ]}
  ```

→ Examples:
  ```sql
  update Employee set Salary = Salary + 5
  where RegNo = 'M2047'
  ```
  ```sql
  update Employee set Salary = Salary * 1.1
  where Dept = 'Administration'
  ```

Notes on Updates

→ As with any side effect statement, the order of updates is important:
  ```sql
  update Employee
  set Salary = Salary * 1.1
  where Salary <= 30
  ```
  ```sql
  update Employee
  set Salary = Salary * 1.15
  where Salary > 30
  ```

→ In this example, some employees may get a double raise! How can we fix this?
**Generic Integrity Constraints**

- The **check** clause can be used to express arbitrary constraints during schema definition.
- **Syntax:**
  ```sql```
  ```
  check (Condition)
  ```
  ```
  Condition is what can appear in a where clause — including nested queries.
  ```
  ```
  For example, the definition of an attribute **Superior** in the schema of table **Employee**:
  ```
  Superior character(6)
  check (RegNo like "1%" or Dept = (select Dept from Employee E
  where E.RegNo = Superior))
  ```

**Assertions**

- Assertions permit the definition of constraints independently of table definitions.
- **Syntax:**
  ```sql```
  ```
  create assertion AssertName check (Condition)
  ```
  ```
  “There must always be at least one tuple in table **Employee**”:
  ```
  create assertion AlwaysOneEmployee check (1 <= (select count(*)
  from Employee))
  ```

**Views**

- Views are "virtual tables" whose rows are computed from other tables (**base relations**).
- **Syntax:**
  ```sql```
  ```
  create view ViewName [(AttributeList)] as SelectSQL
  [with [local|cascaded] check option]
  ```
  ```
  Examples:
  ```
  create view AdminEmployee
  (RegNo,FirstName,Surname,Salary) as
  select RegNo,FirstName,Surname,Salary
  from Employee
  where Dept = 'Admin' and Salary > 10
  create view JuniorAdminEmployee as
  select * from AdminEmployee
  where Salary < 50 with check option
  ```

**Notes on Views**

- SQL views cannot be mutually dependent (no recursion).
- **check option** executes when a view is updated.
- Views can be used to formulate complex queries — views decompose a problem and produce more readable solutions.
- Views are sometimes necessary to express certain queries:
  - Queries that combine and nest several aggregate operators;
  - Queries that make fancy use of the union operator.
Views and Queries

→ "Find the department with highest salary expenditures" (without using a view):

    select Dept from Employee
    group by Dept
    having sum(Salary) >= all
    (select sum(Salary) from Employee
     group by Dept)

→ This solution may not work with all SQL systems.

Views and Queries

→ "Find the department with highest salary expenditures" (using a view):

    create view SalBudget(Dept, SalTotal) as
    select Dept, sum(Salary)
    from Employee
    group by Dept

    select Dept from SalBudget
    where SalTotal = (select max(SalTotal) from SalBudget)

Views and Queries

→ "Find the average number of offices per department":

    Incorrect solution (SQL does not allow a cascade of aggregate operators):

    select avg(count(distinct Office))
    from Employee
    group by Dept

    Correct solution (using a view):

    create view DeptOff(Dept, NoOfOffices) as
    select Dept, count(distinct Office)
    from Employee
    group by Dept

    select avg(NoOfOffices)
    from DeptOffice

Access Control

→ Every element of a schema can be protected (tables, attributes, views, domains, etc.)

→ The owner of a resource (the creator) assigns privileges to the other users.

→ A predefined user _system represents the database administrator and has access to all resources.

→ A privilege is characterized by:
  ✓ a resource;
  ✓ the user who grants the privilege;
  ✓ the user who receives the privilege;
  ✓ the action that is allowed on the resource;
  ✓ whether or not the privilege can be passed on to other users.
Types of Privileges

SQL offers six types of privilege:

- **insert**: to insert a new object into the resource;
- **update**: to modify the resource content;
- **delete**: to remove an object from the resource;
- **select**: to access the resource content;
- **references**: to build a referential integrity constraint with the resource;
- **usage**: to use the resource in a schema definition (e.g., a domain)

grant and revoke

To grant a privilege to a user:

```
grant <Privileges | all privileges > on Resource to Users [with grant option]
```

The **grant option** specifies whether the privilege can be propagated to other users.

For example,

```
grant select on Department to Stefano
```

To take away privileges:

```
revoke Privileges on Resource from Users [restrict | cascade]
```

Database Triggers

Triggers (also known as ECA rules) are elements of the database schema.

General form:

```
on <event> when <condition> then <action>
```

- **Event**: request to execute database operation
- **Condition**: predicate evaluated on database state
- **Action**: execution of procedure that might involve database updates

Example:

```
on "updating maximum enrollment limit"
  if "# registered > new max enrollment limit"
  then "deregister students using LIFO policy"
```

Trigger Details

- **Activation**: occurrence of the **event** that activates the trigger.
- **Consideration**: the point, after activation, when **condition** is evaluated; this can be **immediate** or **deferred**.
  - **Deferred** means that **condition** is evaluated when the database operation (**transaction**) currently executing requests to commit.
- **Condition**: might refer to both the state before and the state after **event** occurs.
Trigger Execution

- This is the point when the action part of the trigger is carried out.
- With deferred consideration, execution is also deferred.
- With immediate consideration, execution can occur immediately after consideration or it can be deferred:
  - If execution is immediate, execution can occur before, after, or instead of triggering event.
  - Before triggers adapt naturally to maintaining integrity constraints: violation results in rejection of event.

Event Granularity

Event granularity can be:

- Row-level: the event involves change of a single row,
  - This means that a single update statement might result in multiple events;
- Statement-level: here events result from the execution of a whole statement; for example, a single update statement that changes multiple rows constitutes a single event.

Multiple Trigger Executions

- Should we allow multiple triggers to be activated by a single event?
- If so, how do we handle trigger execution?
  - Evaluate one condition at a time and if true immediately execute action; or
  - Evaluate all conditions, then execute all associated actions.
- The execution of an action can affect the truth of a subsequently evaluated condition so the choice is significant.

Triggers in SQL-3

- Events: insert, delete, or update statements or changes to individual rows caused by these statements.
- Condition: Anything allowed in a where clause.
- Action: An individual SQL statement or a program written in the language of Procedural Stored Modules (PSM) -- which can contain embedded SQL statements.
Triggers in SQL-3

- Consideration = *immediate* – condition can refer to both the state of the affected row or table before and after the event occurs.
- Execution = *immediate* – can be before or after the execution of the triggering event
- Note that the action of a before-trigger cannot modify the database.
- Granularity: Both *row-level* and *statement-level*.

Before-Trigger with Row Granularity

```
CREATE TRIGGER Max_EnrollCheck
BEFORE INSERT ON Transcript
REFERENCING NEW AS N
--row to be added
FOR EACH ROW
WHEN
((SELECT COUNT(T.StudId)
    FROM Transcript T
    WHERE T.CrsCode = N.CrsCode
    AND T.Semester = N.Semester)
>=
(SELECT C.MaxEnroll
    FROM Course C
    WHERE C.CrsCode = N.CrsCode ))
THEN ABORT TRANSACTION
```

Check that enrollment ≤ limit

Action

After-Trigger with Row Granularity

```
CREATE TRIGGER LimitSalaryRaise
AFTER UPDATE OF Salary ON Employee
REFERENCING OLD AS O
NEW AS N
FOR EACH ROW
WHEN (N.Salary - O.Salary > 0.05 * O.Salary)
THEN UPDATE Employee -- action
    SET Salary = 1.05 * O.Salary
    WHERE Id = O.Id
```

No salary raises greater than 5%

[Note: The action itself is a triggering event; however, in this case a chain reaction is not possible.]

After-Trigger with Statement Granularity

```
CREATE TRIGGER RecordNewAverage
AFTER UPDATE OF Salary ON Employee
FOR EACH STATEMENT
THEN INSERT INTO Log
VALUES (CURRENT_DATE,
    SELECT AVG(Salary)
    FROM Employee)
```

Keep track of salary averages in the log