SQL - A Relational DB Language

SQL

- Structured Query Language
- Includes both Data Definition and Data Manipulation languages.
- Originally SEQUEL (Structured English QUEry Language)
- Developed by IBM for System-R (System R was an experimental database system)
- Has become the inter-galactic standard database language.

SQL Standards

- Due to popularity of SQL, ANSI (NIST) and OSI have standardize the language.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL - 1</td>
<td>1986 by ANSI</td>
</tr>
<tr>
<td>SQL - 2 or (SQL-92)</td>
<td>1992, revised and expanded</td>
</tr>
<tr>
<td>SQL - 3 or (SQL-99)</td>
<td>1999</td>
</tr>
<tr>
<td>SQL 2003</td>
<td>2003</td>
</tr>
<tr>
<td>SQL 2007</td>
<td>2007, Underway. XML extensions</td>
</tr>
</tbody>
</table>
Features of SQL:

- Data Definition Statements
  (Defining, maintaining database objects)
- Data manipulation Statements
  (Accessing and modifying data)
- Views
- Indexes
- Embedded SQL

Data Definition Commands of SQL (DDL):

- Create
- Alter
- Drop

Data Manipulation Commands of SQL (DML):

- Select
- Insert
- Delete
- Update
Data Definition

Creating/Deleting Schemas
Creating/Deleting/Altering Tables
Creating/Deleting User Defined Domains
Creating/Deleting Views
Creating/Deleting Indexes
Creating Schemas:

- Syntax:
  
  ```
  Create Schema <schema_name>  Authorization <owner_name>;
  ```

- Example:
  
  ```
  Create Schema Company  Authorization Willard_Hill;
  ```

- Multiple Schemas may be placed in a common catalog.

- Schemas within a common catalog may share elements such as domain definitions, etc.
Creating Tables:

Syntax:  (Database Dependent)

```sql
Create Table <table_name>
    (<field-name> <field-type> [NOT NULL],
    .
    .
    .
    <field-name> <field-type> [NOT NULL],
    <primary key clause>
    <unique clause>         // alternate key
    <foreign key clause>   
    etc.                   
) ;
```

Field Types:  (Database dependent)

- INTEGER signed full word binary
- SMALLINT signed half word binary
- DECIMAL(p,q) signed packed dec, p digits, dec-pt q from right.
- FLOAT signed double word floating point
- CHAR(n) fixed length char string
- VARCHAR(n) Variable length char str max=n
- others
Example:

```
Create Table Employee
(FName VarChar(15) NOT NULL,
  ... 
SSN Char(9),
Bdate Date,
  ... 
Salary Decimal(10,2),
  ... 
DNO Integer NOT NULL,
  primary key (SSN),
  foreign key (super_ssn) references Employee(SSN),
  foreign key (DNO) references Department (DNumber),
);
```
Constraints Used during Data Definition:

1) **NOT NULL** Clause:
   - Used to ensure the value of an attribute is never left as NULL.
   - Used for primary key attributes.
   - MGRSSN  char(9) NOT NULL,

2) **Default** Value Clause:
   - Used to provide a default value for an attribute, when the user does not provide one)
   - MGRSSN  char(9) NOT NULL  Default "999999999",
   - DNO    int NOT NULL  Default 1,

3) **Primary Key** Clause:
   - Used to define one or more attributes as primary key.
   - Primary key (SSN),

4) **Unique** Clause:
   - Used to define one or more attributes as alternate or candidate key.
   - Unique (LName),    // Last name is a candidate key

3) **Foreign Key** Clause:
   - Used to define referential integrity constraints.
Referential Integrity

- Referential Integrity can be violated when:
  1) Deleting tuples with a primary key field.
  2) Modifying a foreign key value to a non-existing primary key value.
  3) Inserting a new tuple with a foreign key which points to a non-existing primary key.

- Referential Integrity Triggered Actions:

  - SQL-2 provides the following built-in actions when using a Foreign key clause:
    1) SET NULL (When the tuple with primary key is deleted or updated, set all the corresponding foreign key values to NULL.)
    2) CASCADE (If the tuple with primary key is updated, then cascade that change to all the foreign key values.)
    3) SET DEFAULT (When deleting a foreign key, use the default value specified in the default constraint.)
Example:
Example: [Elmasri94]

Create Table Employee
(
    ...
    DNO Int NOT NULL Default 1,
primary key (SSN),

CONSTRAINT EMPSUPERFK // Constraint Name **
foreign key (super_ssn) references Employee(SSN)
ON DELETE SET NULL // When a tuple of a supervising employee is deleted, the values of super_ssn for all employee tuples referencing the deleted tuple should be SET to NULL
ON UPDATE CASCADE, // When a tuple of a supervising employee is updated, the new value should be CASCADED to all tuples referencing the updated employee.

CONSTRAINT EMPDEPTFK
foreign key (DNO) references Department (DNumber)
ON DELETE SET DEFAULT // When a tuple in the Department relation is deleted, the value of the DNO in Employee relation should be SET to its DEFAULT.
ON UPDATE CASCADE, // when a tuple in the Department relation is updated, all the tuples in Employee relation which reference that tuple should be updated as well (CASCADE)

);

** Constraint Names are used to identify particular constraints, in case the constraint must be dropped later and/or replaced with another constraint. (Giving names to constraints are optional.)
Creating User Defined Domains:

- SQL-2 allows the user to create new domains. (Most PC-based SQL versions do not allow this) (i.e. FoxPro, Access)

- Improves Schema Readability

  Similar to Constants in Pascal or C.

- Syntax:

  ```
  Create Domain  <new_type>  as  <exiting_type> ;
  ```

- Example:

  ```
  Create Domain  SSN_TYPE  as  char(9) ;
  ```
Base vs. Virtual Tables

Base Tables:

- Or base relations are relations that are created via the CREATE TABLE command.
- Are actually stored as a file by the DBMS.

Virtual Tables:

- Or virtual relations are relations that are created through the Create View command. May or may not be actual files.
Creating Views

- Syntax:
  CREATE VIEW <view-name>
  [( <fld-name> {,<fld-name>} ... )]
  AS <select Statement>;

- Example:

  CREATE VIEW Female_Emps
  (Fname, Lname, Salary)
  AS
  Select Fname, Lname, Salary from Employee
  where Sex = "F";

Deleting Views

- DROP VIEW <view-name>;

- Example:

  DROP VIEW Female_Emps;
Creating Indexes:

- Syntax:
  ```sql
  CREATE [UNIQUE] INDEX <idx-name>
  ON <base-tbl-name>
  (<fld-name> [ASC|DESC],
  . . .
  <fld-name> [ASC|DESC])
  [CLUSTER];
  ```

- UNIQUE specifies "super key"

- CLUSTER specifies physical order must match logical field order.

UNIQUE plus CLUSTER on a single field produces a primary key.

CLUSTER alone = clustering index.

Neither is a secondary index.

Example:

- This example creates an index consisting of the fields “Home Phone” and “Extension” in the “Employees” table.

  ```sql
  CREATE INDEX NewIndex
  ON Employees (HomePhone, Extension);
  ```

- This example creates an index on a hypothetical ODBC linked table. The table's remote database is unaware of and unaffected by the new index.

  ```sql
  CREATE UNIQUE INDEX OrderID
  ON OrderDetails (OrderID);
  ```
**Create Index in MS-ACCESS:**

Creates a new index on an existing table.

**Note:** For non-Jet databases, the Microsoft Jet database engine doesn't support the use of CREATE INDEX (except to create a pseudo index on an ODBC attached table) or any of the data definition language (DDL) statements. Use the data access object Create methods instead. For more information, see the Remarks section.

**Syntax**

CREATE [ UNIQUE ] INDEX index 
ON table (field [ASC|DESC],[, field [ASC|DESC], ...])
[WITH { PRIMARY | DISALLOW NULL | IGNORE NULL }]

The CREATE INDEX statement has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The name of the index to be created.</td>
</tr>
<tr>
<td>table</td>
<td>The name of the existing table the index will be created on.</td>
</tr>
<tr>
<td>field</td>
<td>The name of the field or fields to be indexed. To create a single-field index, list the field name in parentheses following the table name. To create a multiple-field index, list the name of each field to be included in the index. To create descending indexes, use the DESC reserved word; otherwise, indexes are assumed to be ascending.</td>
</tr>
</tbody>
</table>

This example creates an index on the “Employees” table using the “Social Security Number” field. No two records can have the same data in the SSN field, and no Null values are allowed.

```sql
CREATE UNIQUE INDEX CustID 
ON Customers (CustomerID) 
WITH DISALLOW NULL;
```
Create Index in MYSQL:

Creates a new index on an existing table.

    ALTER TABLE `student` ADD INDEX ( `DeptID` )
Deleting Indexes:

- DROP INDEX `<index-name>`;

Deleting an Index in MySQL:

- `ALTER TABLE `student` DROP INDEX `DeptID`;`
Deleting Schemas

- Delete the schema (possibly including all its tables, domains, etc.)

- Syntax:

  \[ \text{DROP SCHEMA } <\text{schema\_name}> <\text{drop\_behavior}> \]

  *drop\_behavior* can be RESTRICT or CASCADE

- Example:

  \[ \text{DROP SCHEMA COMPANY Cascade;} \quad \text{// Drop the schema and all its tables.} \]

  \[ \text{DROP SCHEMA COMPANY Restrict;} \quad \text{// Drop the schema only if it is empty.} \]
Deleting a Table

- Syntax:
  DROP TABLE <table_name> <drop_behavior>;

  drop_behavior can be RESTRICT or CASCADE

- Example:

  DROP TABLE Dependent Cascade; // Drop the table and all other tables which refer to this table. (i.e. tables that have a foreign key pointing to this table.)

  DROP TABLE Dependent Restrict; // Drop the table only if it is not referenced in any constraints or views.
Altering the Structure of a Table

- Allow the user to modify the definition of a base table. Items such as adding, deleting or modifying attributes, constraints, etc.

- Syntax:

  ALTER TABLE <table-name>
  <COMMAND> <attribute data_type> [options];

- Commands can be (Add, Alter, Drop)
- Options may be:
  - Drop Options: (Cascade, Restrict..)
  - Alter Options: (Drop, Set, ..)

- Example 1: (ADD)

  - Add a new attribute to the employee relation.

    ALTER TABLE Company.Employee
    ADD Job VARCHAR(12);

  - At this point the "Job" attribute must be populated. Either by using the update command on each tuple or by using the DEFAULT clause. (The default currently will be NULL)

- Example 2: (ADD) (Examples form MS-Access)

  This example adds a "Salary" field with a data type of Currency to the "Employees" table.

    ALTER TABLE Employees
    ADD COLUMN Salary CURRENCY;

  This example adds a foreign key to the "Orders" table. The foreign key is based on the "Employee ID" field and refers to the "Employee ID" field of the "Employees" table. In this example, you don't have to list the "Employee ID" field after the "Employees" table in the REFERENCES clause because "Employee ID" is the primary key of the "Employees" table.

    ALTER TABLE Orders
ADD CONSTRAINT OrdersRelationship FOREIGN KEY (EmployeeID)
REFERENCES Employees (EmployeeID);
Example 1: (DROP)

- Delete an attribute from the employee relation.

```
ALTER TABLE Company.Employee
    DROP  Address CASCADE;
```

- Drop Options can be: (Cascade, Restrict)

If CASCADE is used:

- All constraints and views that reference the column are dropped automatically.

If RESTRICT is used:

- The command will only work if NO constraints or views reference the column

Example 2: (DROP) (Examples from MS-Access)

This example removes the "Salary" field from the "Employees" table.

```
ALTER TABLE Employees DROP COLUMN Salary;
```

This example removes the foreign key from the "Orders" table.
```
ALTER TABLE Orders DROP CONSTRAINT OrdersRelationship;
```
Example: (ALTER)

- Change the specification of attribute (MGRSSN). Drop the current default from the MGRSSN attribute:

  \[
  \text{ALTER TABLE Company.Employee} \\
  \hspace{1em} \text{ALTER MGRSSN DROP DEFAULT;}
  \]

- Change the specification of attribute (MGRSSN). Set the current default for MGRSSN to "999-99-9999":

  \[
  \text{ALTER TABLE Company.Employee} \\
  \hspace{1em} \text{ALTER MGRSSN SET DEFAULT "999-99-9999";}
  \]

- Alter Options can be: (Drop, Set,..)

Example: (DROP CONSTRAINT)

- Delete one of the constraints placed on the employee relation. Drop the foreign key constraints placed on SUPERSSN.

  \[
  \text{ALTER TABLE Company.Employee} \\
  \hspace{1em} \text{DROP CONSTRAINT EMPSUPERFK CASCADE;}
  \]
Data Manipulation Language of SQL (DML):

- SQL provides one basic statement for retrieving data from the database.

The "SELECT" statement!

- SELECT statement is not the same as the "selection operation" described in relational algebra.)

Basic Syntax:

```
SELECT <Attribute-List>
FROM <Table-List>
WHERE <condition>;
```

About the SELECT Clause:

The SELECT clause usually provides a mechanism for **vertical** selection of fields (Project operator in Relational algebra)

About the FROM Clause:

The FROM clause basically provides a Cartesian product of the relations or tables.

About the WHERE Clause:

The WHERE clause usually provides a mechanism for **horizontal** selection of records. (Select operator in Relational algebra).
The WHERE clause also provide the condition for any join operations.

- Example Queries:

  1) Retrieve the Birth Date and Address of the employees whose name is "John Smith"

```sql
SELECT BDate, Address
FROM Employee
WHERE FName = "John" AND LName = "Smith";
```
Example Queries:

2) Retrieve the Name and Address of all employees who work for Research department.

```
SELECT FName, LName, Address
FROM Employee, Department
WHERE DName = "Research" AND DNumber = DNO;
```

- Note that this operation is the same as Select, Project and Join operation in relational algebra.

- In the where clause the:
  - DName = "Research" is a Select operation.
  - DNumber = DNO is a Join operation.

- In the Select clause the:
  - SELECT FName, LName, Address is a Project operation.
Example Queries:

3) For every project located in "Stafford", list the Project Number, the Controlling Department, and the Department Manager's Last name, Address and Birth date.

```sql
SELECT PNumber, DNUM, LName, Address, Bdate
FROM Project, Employee, Department
WHERE Dnum = DNumber AND
      MGRSSN = SSN AND
      PLocation = "Stafford";
```
Ambiguity

Dealing with Ambiguous Attribute Names:

- To resolve any ambiguity the programmer must use **fully qualified** notation to refer to attributes. (i.e. prefix the attribute name with its relations name.

- Example:  (Assume the department number in the Employee relation was also called DNumber)

```sql
SELECT Employee.FName, Employee.LName
FROM Employee, Department
WHERE Department.Dname= "Research"  AND
      Department.DNumber = Employee.DNumber;
```
Ambiguity and Recursive Relationships:

- How should we deal with queries which require a relation to appear \textit{twice} in the FROM Clause?

- Creating alias relations using the \textbf{FROM} clause.

- Example Query:

  - Retrieve the First and Last name of each Employee and the First and Last name of his or her Supervisor.

    \begin{verbatim}
    SELECT Emp.FName, Emp.LName, Sup.FName, Sup.LName
    FROM Employee AS Emp , Employee AS Sup
    WHERE Emp.SuperSSN = Sup.SSN;
    \end{verbatim}

- The "Emp" and "Sup" are called \textbf{Aliases}.
Creating Aliases for the Attribute Names:

- SQL allows the user to create aliases for the attributes of a relation. (See the FROM clause)

- Example:

  ```sql
  SELECT Emp.FN, Emp.LN, Emp.ADDR
  FROM Employee AS Emp (FN, MI, LN, SSN, BD, ADDR,..)
  WHERE Emp.FN = "James";
  ```
The Missing WHERE Clause:

- A missing WHERE clause simply indicates "No Condition" on tuple selection.

- Example:

  ```sql
  SELECT FName, LName, Salary
  FROM Employee;
  //All tuples of the relation specified in the FROM clause are selected.
  ```

- If more than one relation name is specified in the FROM clause, then the Cartesian Product is selected. (All possible combinations.)

- Example:

  ```sql
  SELECT SSN, DName
  FROM Employee, Department;
  //see page 194, fig 7.2 (f)
  ```
Wild Card Characters:

- The "*" character may be used to specify the selection of **ALL** attributes.

- Example 1:

  Retrieve all attributes of the department relation where DNUMBER = 5.

  ```sql
  SELECT * 
  FROM Department 
  WHERE DNUMBER = 5;
  ```

- Example 2:

  Retrieve all attributes of the Cartesian Product of employee and Department relations.

  ```sql
  SELECT * 
  FROM Employee, Department;
  ```
String Operations in SQL

- Strings are specified by enclosing them in single quotes.

  - Example: ‘Database’

- To include a single quote within a string we use two single quotes.

  - Example: ‘I’l’l be there.’
Pattern Matching

- Pattern matching is accomplished using the LIKE operator

- Example:

```
SELECT Customer_name
FROM Customer
WHERE Customer_address LIKE 'Main%'
```

Wild Card Characters for Pattern Matching

<table>
<thead>
<tr>
<th>Wild Card</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>% (Percent)</td>
<td>Match any substring</td>
</tr>
<tr>
<td>_ (Underscore)</td>
<td>Match any single character</td>
</tr>
</tbody>
</table>

- Examples:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘IU%’</td>
<td>Matches any string beginning with ‘IU’ such as (IUSB, IUFW, IUPUI)</td>
</tr>
<tr>
<td>‘%idge%’</td>
<td>Matches any string containing ‘idge’ as a substring, such as (Perryridge, Rockridge, Bridge, idgedege)</td>
</tr>
<tr>
<td>‘ ___ ‘</td>
<td>Matches any string of exactly 3 characters.</td>
</tr>
<tr>
<td>‘ ___ %’</td>
<td>Matches any string of at least 3 characters.</td>
</tr>
</tbody>
</table>
Escape Character ("\")

- If the string is to include a ‘%’ or ‘_’ or ‘\’, then SQL allows the user to use the Backslash as the escape character.

- Example:

  Find strings that include the substring “\ab%cd”

  ```
  Select ...
  From ...  
  Where ... LIKE ‘ab\%cd’
  ```

  Find strings that include the substring “\ab\cd”

  ```
  Select ...
  From ...  
  Where ... LIKE ‘ab\\cd’
  ```

To Search for Mismatch

- SQL allows the use of the NOT operator

  ```
  Select ...
  From ...  
  Where ... NOT LIKE ‘abcd’
  ```
Other string manipulation operators exist. Consult the specific SQL implementation.
Table Vs. Sets in SQL:

- SQL **relations are Less Restrictive than Sets** in relational algebra.

- Set can **not** have duplicate tuples.

- SQL relations allow duplicate tuples!
  - Usually in the course of producing the result of a query.
  - The user can **explicitly** exclude Duplicate tuples! (See Distinct!)

**Why Allow Duplicates?**

1) Eliminating duplicates is expensive. (Sort, followed by deletion of duplicates.)

2) The user may want to see the duplicates in the result of a query.

3) When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates.

- Example:

  Count, how many times the name "John" appears in the data.

  (We will introduce the aggregate functions later.)
Eliminating Duplicate Tuples:

- To eliminate duplicates from the result of a query, SQL provides the **DISTINCT** keyword.

- Example 1:

  Retrieve the list of project location?

  ```sql
  SELECT DISTINCT PLocation
  FROM Project;
  ```
Set Operations in SQL:

- SQL has implemented the following set operations:
  
  - UNION
  - INTERSECT
  - EXCEPT (Same as Difference)

- The result of these relations are sets by default. (i.e. No Duplicates) If the duplicates are desired the "ALL" clause can be used in some versions of SQL.

- The user may choose to keep the duplicates by specifying the ALL keyword.

- When using set operation (e.g. UNION, EXCEPT, INTERSECT), the relations must be Union Compatible.
Example 1: (UNION)

Find all the project numbers that "Smith" **works for** or **manages the controlling department**:

\[
\begin{align*}
( & \text{SELECT PNumber} \\
 & \text{FROM Project, Department, Employee} \\
 & \text{WHERE DNUM = DNumber AND} \\
 & \text{MGRSSN = SSN AND} \\
 & \text{LName = "Smith"}) \\
\text{UNION} \\
( & \text{SELECT PNumber} \\
 & \text{FROM Project, Works-On, Employee} \\
 & \text{WHERE PNumber = PNO AND} \\
 & \text{ESSN = SSN AND} \\
 & \text{LName = "Smith"});
\end{align*}
\]
Example 2 (INTERSECT):

Find the social security numbers and names of all student teaching assistants:

(\text{SELECT} \ SSN, \ Name \\
\text{FROM} \ Employee) \\
\text{INTERSECT} \\
(\text{SELECT} \ SSN, \ Name \\
\text{FROM} \ Student); \\

Example 3 (EXCEPT):

Find the social security numbers and names of all student teaching assistants: (Assuming employees can be faculty or students only.)

(\text{SELECT} \ SSN, \ Name \\
\text{FROM} \ Employee) \\
\text{EXCEPT} \\
(\text{SELECT} \ SSN, \ Name \\
\text{FROM} \ Faculty);
Nested Queries:

- A select query within the WHERE clause of another query. This is accomplished using the **IN** operator.

- The **IN** operator determines whether the value of an expression is equal to any of several values in a specified list.

- **NOT IN** can be used to determine the absence of items in a list.

- Example 1:

  Find all the project numbers that "Smith" **works for** or **manages the controlling department**:

  ```sql
  SELECT DISTINCT PNumber // Outer Query
  FROM Project
  WHERE PNumber IN
  (SELECT PNumber // Inner Query
   FROM Project, Employee, Department
   WHERE DNum = DNumber AND
   MGRSSN = SSN AND
   LName = "Smith")
  OR
  Pnumber IN
  (SELECT PNO
   FROM Works-ON, Employee,
   WHERE ESSN = SSN AND
   LName = "Smith");
  ```

- Example 2 from MS-Access:
  Determine which orders are shipped to a set of specified regions:

  ```sql
  SELECT * 
  ```
FROM Orders
WHERE ShipRegion In ('East', 'MID-West', 'South');
Aggregate Functions:

- **Built-in Functions:**

  - **Count** // Returns the number of tuples resulted from a query.
  - **Sum** // Applied to numeric attributes
  - **Max**
  - **Min**
  - **Avg**

- **Example -1**

  Find the SUM of salaries of all employees in the "RESEARCH" department, as well as MAX, MIN and AVG salary in that department.

  ```sql
  SELECT SUM(salary), MAX(salary),
  MIN(salary), AVG(salary)
  FROM Employee, Department
  WHERE DNO = DNumber AND
  DName = "Research";
  ```

- **Example -2**

  Calculate the number of employees in the "RESEARCH" department.

  ```sql
  SELECT COUNT(*) // count the number of tuples
  FROM Employee, Department
  WHERE DNO = DNumber AND
  ```
Example -3

Find all employees who have 2 or more dependents.

```
SELECT LName, FName
FROM Employee
WHERE (SELECT COUNT(*)
       FROM Dependent
       WHERE SSN = ESSN) >= 2;
```
Grouping Tuples:

- Many queries and reports require the use of grouping.
- This can be accomplished via the SQL **GROUP BY** clause.

**Syntax:**

```
SELECT <Attribute-List>
FROM <Table-List>
WHERE <condition>
GROUP BY <Grouping-Attr(s)>;
```

**Example -1**

Find the average salary of employees in each department.

```
SELECT DNO, AVG(salary)
FROM Employee
GROUP BY DNO;
```

<table>
<thead>
<tr>
<th>DNO</th>
<th>AVG(salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>33250</td>
</tr>
<tr>
<td>4</td>
<td>31000</td>
</tr>
<tr>
<td>1</td>
<td>55000</td>
</tr>
</tbody>
</table>

---

**DNO is the Grouping Attribute.**

DNO should appear in the final selection so that the tuples could be grouped by it.

**Result:**

<table>
<thead>
<tr>
<th>DNO</th>
<th>AVG(salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>33250</td>
</tr>
<tr>
<td>4</td>
<td>31000</td>
</tr>
<tr>
<td>1</td>
<td>55000</td>
</tr>
</tbody>
</table>
Example -2

For each project, retrieve the Project Number, Project Name, and the Number of Employees who Work-On that project.

```
SELECT PNumber, PName, COUNT(*)
FROM Project, Works_On
WHERE PNumber = PNO  // Join
GROUP BY PNUMBER, PNAME;  // Group By

NOTE: The Join is done first, then the grouping and aggregate functions are applied.
```
Having Clause:

- Used in conjunction with the **Group By** clause.

- Syntax:

  ```
  SELECT <Attribute-List>
  FROM <Table-List>
  WHERE <condition>
  GROUP BY <groupfieldlist>
  HAVING <groupcriteria>;
  ```

- Provides a condition on the group of tuples associated with each value of the grouping attribute. (only the groups that satisfy the **Having** condition are retrieved in the result of the query).

- Example:

  For each Project on which more than two employees work retrieve the project number and the number of employees who work on that project.

  ```
  SELECT PNumber, PName, COUNT (*)
  FROM Project, Works_On
  WHERE PNumber = PNO
  GROUP BY PNumber, PName
  HAVING COUNT (*) > 2;
  ```
Sorting the Results of a Query:

- ORDER BY clause.
- Syntax:

```sql
SELECT <Attribute-List>
FROM <Table-List>
WHERE <condition>
ORDER BY <ATTR-LIST>;
```

- Example:

Retrieve a list of Employees and the Projects they are working on. Sorted by Department. Within each department, alphabetically sort by the Last name and then First name.

```sql
SELECT DName, LName, FName, PName
FROM Department, Employee, Works-On, Project
WHERE Dnumber = DNO AND
      SSN = ESSN AND
      PNO = PNumber
ORDER BY DName, LName, FName;
```

- Default Sort Order is **Ascending**.

- Sort Order can be specified as **ASC** or **DESC**:

  ```sql
  i.e. ORDER BY DName DESC, LName ASC;
  ```
The INSERT INTO Command:

- Used to add a single tuple into a relation.
- Syntax:

  ```
  INSERT INTO <tbl-name>
  [ ( <fld> [,<fld>] ... ) ]
  VALUES
  ( const [, const] ... );
  ```

- Example 1: (Insert a whole record)

  ```
  INSERT INTO Employee
  VALUES
  ('John', 'J', 'Doe', '123456789', '30-Dec-52',
   '98 Oak Forest, Katy, TX', 'M', 37000,
   '987654321',
   4);
  ```

- Example 2: (Insert into specific attributes)

  ```
  INSERT INTO Employee
  (FName, LName, SSN)
  VALUES
  ('John', 'Doe', '123456789');
  ```
The DELETE Command:

- Remove one or more tuples from a relation.

- Syntax:
  ```sql
  DELETE
  FROM <table>
  WHERE <criteria>;
  ```

- Example 1:
  ```sql
  DELETE
  FROM Employee
  WHERE LName = 'Brown';
  ```

- Example 2:
  ```sql
  DELETE
  FROM Employee
  WHERE DNO IN (SELECT DNumber
                  FROM Department
                  WHERE DName = "Research");
  ```

- Example 3: (Delete all employees)
  ```sql
  DELETE
  FROM Employee;
  ```
The UPDATE Command:

- Update the attributes of one or more selected tuples.

**Syntax:**

```
UPDATE <able>
SET <field name = newvalue>
WHERE <criteria>;
```

**Example 1:**

```
UPDATE Project
SET PLocation = "Bellaire",
    DNum = 5;
WHERE PNumber = 10;
```

**Example 2:** (Modifying several tuples at the same time)

```
UPDATE Employee
SET Salary = Salary * 1.1
WHERE DNO IN (SELECT DNumber
               FROM Department
               WHERE DName = "Research");
```

- Update only works on one relation at a time. Unless a KEY ATTRIBUTE is updated where it should be propagated to foreign key values of the other relation. (SEE on UPDATE constraint in CREATE TABLE command)
Creating Views in SQL

- A view is a single table which is derived from other tables. (AKA virtual table)

- The system may or may not physically create a base table for the view. (implementation dependent)

- Depending on implementation views are often created for:
  1) Providing a restricted view of the database to the users.
     and/or
  2) Optimization of frequently performed queries (specially those that involve costly joins)

- Syntax:

  `CREATE VIEW <View_Name> AS (SELECT <attribute_list>
                              FROM <relation_list>
                              WHERE <condition>);`

- Example:

  The following creates a view (virtual table) from three tables where SSN=ESSN AND PNO=PNUMBER.

  `CREATE VIEW WORKS_ON_VIEW AS SELECT FNAME, LNAME, PNAME, HOURS
                              FROM EMPLOYEE, PROJECT, WORKS_ON
                              WHERE SSN=ESSN AND PNO=PNUMBER;`

  `WORKS_ON_VIEW`
Deleting Views

- Syntax:

  DROP VIEW <View_Name>;

- Example:

  DROP VIEW WORKS_ON_VIEW;
Creating Indexes

- Creating indexes for faster access.

- Syntax:

  ```sql
  CREATE INDEX <Index_Name>
  ON relation_name(attribute_name);
  ```

- Example:

  This example creates an index named LNAME_INDEX on the LNAME attribute of the table EMPLOYEE.

  ```sql
  CREATE INDEX LNAME_INDEX
  ON EMPLOYEE(LNAME);
  ```

Deleting Indexes

- Syntax:

  ```sql
  DROP INDEX <Index_Name>;
  ```

- Example:

  ```sql
  DROP INDEX LNAME_INDEX;
  ```