# SQL Concepts

**Database System Concepts, 6th Ed.**

# Data Definition Language

The SQL **Data Definition language (DDL)** allows the specification of information about relations, including:

- The schema for each relation
- The domain of values associated with each attribute
- Integrity constraints
- Also other information such as
  - The set of indices to be maintained for each relations
  - Security and authorization information for relation by access rights
  - The physical storage structure of each relation on disk
- Example: CREATE, ALTER, DROP, RENAME, and TRUNCATE

# Data Manipulation Language

The SQL **Data Manipulation language (DML)** provides the ability

- To query information from the database
- To insert tuples into
- To delete tuples from
- To modify tuples in the database
  - Example: SELECT, INSERT, UPDATE, and DELETE.

# Data Control Language

The SQL **Data Control language (DCL)** used for providing security to database objects.

- Example: GRANT and REVOKE.

# Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n*
- **int.** Integer (a finite subset of the integers that is machine-dependent)
- **smallint.** Small integer (a machine-dependent subset of the integer domain type)
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision
- **float(n).** Floating point number, with user-specified precision of at least *n* digits
- Date, Boolean, etc.

# Special Value - NULL

- Each type may include a special value called the **null value**
- **A null value** indicates an absent value that may exist but be unknown or that may not exist at all.
- In certain cases, we may wish to prohibit null values from being entered for integrity constraints.

# Char vs Varchar

- **char data type** stores fixed length strings.
  - For example, an attribute *A of type char(10). If we store a string "Avi" in this attribute, 7 spaces* are appended to the string to make it 10 characters long.
- **varchar data type**
  - if store **"Avi" in attribute *B, no spaces would be*** added.
- When comparing two values of type **char, if they are of different lengths** extra spaces are automatically added to the shorter one to make them the same size, before comparison.
- When comparing a **char type with a varchar type, one may expect extra spaces** to be added to the **varchar type to make the lengths equal, before comparison;**
  - may or may not to be done, depends on the database system.

# Create Database

- An SQL relation is defined using the **create Database** command:

  Syntax: CREATE DATABASE *databasename*;

  Eg. CREATE DATABASE testDB;

- The DROP DATABASE statement is used to drop an existing SQL database.

  Syntax: DROP DATABASE *databasename*;

  Eg. DROP DATABASE testDB;

# Create Table Construct

- An SQL relation is defined using the **create table** command:

  **create table** $r$ ($A_1$ $D_1$, $A_2$ $D_2$, ..., $A_n$ $D_n$,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$));

  - $r$ is the name of the relation
  - each $A_i$ is an attribute name in the schema of relation $r$
  - $D_i$ is the data type of values in the domain of attribute $A_i$

# Create Table and Insert

- **Example:** Create table department having department name, building information and budget.
- **create table *department***

  (*deptname* ***varchar (20),***

  *building* ***varchar (15),***

  *budget* ***numeric (12,2)****);*

# Create Table

- Example: Create Table instructor having ID, Name, Salary and department to which he belongs
- **create table** *instructor* (

  *ID* **char**(5),

  *Name* **varchar**(20),

  *dept_name* **varchar**(20),

  *Salary* **numeric**(8,2));

# Integrity Constraints in Create Table

- **not null**
- **primary key** $(A_1, ..., A_n)$
- **foreign key** $(A_m, ..., A_n)$ **references** $r$

# Integrity Constraints in Create Table

- **not null**

- The **not null constraint on an attribute specifies that the null value** is not allowed for that attribute
  - Excludes the null value from the domain of that attribute
  - For example, the **not null constraint on the *name attribute of the instructor relation ensures that*** the name of an instructor cannot be null.

  - The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

  - CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int );

**Not NULL on ALTER Table:**

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

# Integrity Constraints in Create Table

- primary key ($A_1$, ..., $A_n$ )

•The primary-key specification says that attributes $A_1$, ..., $A_n$ form the primary key for the relation

•The primary key attributes are required to be non null and unique

  •No tuple can have a null value for a primary-key attribute, and

  •No two tuples in the relation can be equal on all the primary-key attributes

•Although the primary-key specification is optional, it is generally a good idea to specify a primary key for each relation

•primary key declaration on an attribute automatically ensures not null

# Create Table and Insert

- **Example:** Create table department having department name, building information and budget.
- **create table *department***

  (*dept_name* ***varchar (20),***

  *building* ***varchar (15),***
  *budget* ***numeric (12,2),***

  ***Primary key (dept_name)****);*

# Integrity Constraints in Create Table

- foreign key ($A_m$, ..., $A_n$ ) references r

•The foreign key specification says that the values of attributes ($A_m$, ..., $A_n$ ) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation s

•The definition of the course table has a declaration "foreign key (dept name)
references department"

- •This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the primary key attribute (dept name) of the department relation.
- •Without this constraint, it is possible for a course to specify a nonexistent department name.

# Integrity Constraints in Create Table

- **not null**
- **primary key** ($A_1$, ..., $A_n$ )
- **foreign key** ($A_m$, ..., $A_n$ ) **references** *r*

Example: Declare *ID* as the primary key for *instructor*, *name requires value and dept_name should belong to department table*

.

**create table** *instructor* (
ID char(5),
name varchar(20) not null,
dept_name varchar(20),
salary numeric(8,2),
primary key (ID),
foreign key (dept_name) references department (dept_name));

**primary key** declaration on an attribute automatically ensures **not null**

# Schema of University

1. ***department****(deptname, building, budget)  (Note: building is a foreign key from Classroom table)*
2. ***course****(courseID, title, deptname, credits)*
3. ***student****(SID, name, deptname, tot cred)*
4. ***classroom****(building, roomnumber, capacity)*
5. ***instructor****(Inst_ID, name, deptname, salary)*
6. ***section****(courseID, secID, semester, year, building, roomnumber, timeslotID)*
7. ***teaches****(Inst_ID, courseID, secID, semester, year)*
8. ***takes****(Inst_ID, courseID, secID, semester, year, grade)*
9. ***advisor****(SID, Inst_ID)*
10. ***prereq****(courseID, prereqID)*
11. ***time slot****(timeslotID, day, starttime, endtime)*

# And a Few More Relation Definitions

**create table** student (

ID **varchar**(5), name **varchar**(20) not null, dept_name **varchar**(20),
tot_cred **numeric**(3,0), **primary key** (ID),
**foreign key** (dept_name) **references** department (dept_name) );

**Create table** takes (

ID varchar(5), course_id varchar(8), sec_id **varchar**(8), semester **varchar**(6),
grade **varchar**(2), year numeric(4,0),
**primary key** (ID, course_id, sec_id, semester, year),
**foreign key** (ID) **references** student,
**foreign key** (course_id, sec_id, semester, year) **references** section );

- Note: sec_id can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

# And more still

- **create table** *course* (
  *course_id* **varchar**(8) **primary key**,
  *title* **varchar(**50),
  *dept_name* **varchar**(20),
  *credits* **numeric**(2,0),
  **foreign key** *(dept_name*) **references** *department)* );
  - Primary key declaration can be combined with attribute declaration as shown above

# Alter Table Construct

- **Alter table**
  - To modify the structure of the table
  - **alter table** r **add** A D

    where A is the name of the attribute to be added to relation r and D is the domain of A

    ALTER TABLE table_name
    ADD column_name datatype;

    ALTER TABLE employee
    ADD Email varchar(255);

    **All tuples in the relation are assigned null as the value for the new attribute**.
  - **alter table** r **drop** A

    where A is the name of an attribute of relation r

ALTER TABLE *table_name*
DROP COLUMN *column_name*;

The following SQL deletes the "Email" column from the " employee " table:

ALTER TABLE employee
DROP COLUMN Email;

# Drop Table Construct

- **drop table** *student*
  - Deletes the table and its structure
  - If dropped the table, cannot get it back and all the references to the table will not be valid

DROP TABLE *table_name*;

e.g. DROP TABLE trial;

# Truncate Table

- Removes all rows from the table, but the table structure and its columns, constraints, indexes and so on remain
- To remove the table definition with its data, one can use drop table

Truncate table <tablename>;

- Restrictions
  - Cannot use truncate on table that is referenced by a foreign key constraint

# Rename Table

- Rename Table OLD_NAME to NEW_NAME
- E.g.
  - Rename Table employee to New_Employee;

# DML

- Provides the ability to query information
  - Select
- Insert, delete and update tuples
  - Insert, Delete, Update

# Insert into Table

To insert the values without the column names

**INSERT INTO department (deptname, building, budget)**
**VALUES ('Biology', '4006', '220000');**

To insert the values with the corresponding column names

**insert into *instructor (Id, name, deptname, Salary)* values (1001, "Smith", "Biology", '66000');**

**insert into instructor values (10211, "Semi", "Biology", '60000');**

# Insert into Table

To insert values from other table

- All the columns

  **insert into** *instructor* **select * from old_instructor;**

- Specific column

  **insert into** *instructor (id, name)* **select** id,
  name **from old_instructor;**

- Specific rows

  **insert into** *instructor* **select * from old_instructor**
  **where id = '10211';**

# The Select Clause (Cont.)

- A typical SQL query has the form:

  **select** $A_1, A_2, ..., A_n$ **from** $r_1, r_2, ..., r_m$ **where** $P$

  - $A_i$ represents an attribute
  - $R_i$ represents a relation
  - $P$ is a predicate.

- **The result of an SQL query is a relation**

# The Select Clause

- The **select** clause list the attributes desired in the result of a query
  - Corresponds to the projection operation of the relational algebra

- Example: Find the names of all instructors:

  **select** *name* **from** *instructor;*

# The Select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- **Syntax:**

  SELECT DISTINCT *column1*, *column2*, ... FROM *table_name*;

- Find the names of all departments with instructor, and remove duplicates
  - **select distinct** *deptname* **from** *instructor;*

Difference between following:

1. SELECT Country FROM Customers;

2. SELECT DISTINCT Country FROM Customers;

1. SQL statement selects all (including the duplicates) values from the "Country" column in the "Customers" table.

2. SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table:

# The select Clause (Cont.)

- An asterisk in the select clause denotes "all attributes"

  **select** * **from** *instructor;*

- The **select** clause can contain
  - Arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples
  - Character Functions (uppercase…), Numeric functions (round…), Truncating numeric data, Concatenating character data, etc.
- The query:

  **select** *ID*, *name*, *salary/12* from *instructor;*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

# The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra
- To find all instructors in Computer dept with salary < 80000

**Select name from instructor**
**where deptname = "Computer" AND salary < 80000;**

- Comparison results can be combined using the logical connectives **and, or,** and **not**

- Comparisons can be applied to results of arithmetic expressions

# The from Clause

- Display data from more than one table
- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  **select** * **from** *instructor*, *teaches;*
  - **Generates every possible instructor – teaches pair, with all attributes from both relations**
- Cartesian product
  - Not very useful directly
  - But useful combined with where-clause condition (selection operation in relational algebra)

# Cartesian Product: *instructor X teaches*

*instructor*

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 60000 |

*teaches*

| *ID* | *course_id* | *sec_id* | *semester* | *year* |
|---|---|---|---|---|
| 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | FIN-201 | 1 | Spring | 2010 |
| 15151 | MU-199 | 1 | Spring | 2010 |
| 22222 | PHY-101 | 1 | Fall | 2009 |

**select * from** *instructor, teaches*

| *inst.ID* | *name* | *dept_name* | *salary* | *teaches.ID* | *course_id* | *sec_id* | *semester* | *year* |
|---|---|---|---|---|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-347 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 12121 | FIN-201 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 15151 | MU-199 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 22222 | PHY-101 | 1 | Fall | 2009 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 12121 | Wu | Finance | 90000 | 10101 | CS-101 | 1 | Fall | 2009 |
| 12121 | Wu | Finance | 90000 | 10101 | CS-315 | 1 | Spring | 2010 |
| 12121 | Wu | Finance | 90000 | 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | Wu | Finance | 90000 | 12121 | FIN-201 | 1 | Spring | 2010 |
| 12121 | Wu | Finance | 90000 | 15151 | MU-199 | 1 | Spring | 2010 |
| 12121 | Wu | Finance | 90000 | 22222 | PHY-101 | 1 | Fall | 2009 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

# DELETE

It is a Data Manipulation Language Command (DML). It is used to delete one or more tuples of a table.
With the help of the “DELETE” command, we can either delete all the rows in one go or can delete rows one by one. i.e., we can use it as per the requirement or the condition using the Where clause.
It is comparatively slower than the TRUNCATE command. The TRUNCATE command does not remove the structure of the table.

SYNTAX –
If we want to delete all the rows of the table:
DELETE from;

SYNTAX –
If we want to delete the row of the table as per the condition then we use the WHERE clause,
DELETE from  WHERE  ;

# Delete From Table

- **delete from <tablename>** *[where condition]*
  - **Retaining the table structure and**

    **Deletes all the rows if written without where condition**

    Deletes only the rows satisfying the condition

    delete * from <tablename>;

- The following SQL statement deletes the customer "Alfad Ali" from the "Customers" table:

  DELETE FROM employee WHERE Name = 'Alfad Ali';

- The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

  DELETE FROM Customers;

# Update

- Update single or multiple columns as per requirement
  - **Update** instructor **set** salary = 60000;

    Single column, for all the Rows
  - **Update instructor set name = 'Sudarshan' where name = 'Srinivasan';**

  Only for the specific Row
  - **Update instructor set name = 'Srinivasan', salary='40000' where name = 'Sudarshan';**

  Multiple columns for the specific row

# The SQL AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators
The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND are TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.
- The NOT operator displays a record if the condition(s) is NOT TRUE.

AND Syntax:

SELECT *column1*, *column2*, ... FROM *table_name* WHERE *condition1* AND *condition2* AND *condition3* ...;

E.G.

1. SELECT * FROM customer WHERE country='Germany' AND city='Berlin';

OR Syntax:

SELECT *column1*, *column2*, ... FROM *table_name* WHERE *condition1* OR *condition2* OR *condition3* ...;

E.g.

1. SELECT * FROM customer WHERE city='Berlin' OR city='München';
2. SELECT * FROM customer WHERE country='Germany' OR country='Spain';

NOT Syntax:

SELECT *column1*, *column2*, ... FROM *table_name* WHERE NOT *condition*;

e.g.

1. SELECT * FROM customer WHERE NOT city='Surat';

# as clause

- The SQL allows renaming relations and attributes using the **as** clause: old-name **as** new-name

  E.g.

  **select** ID, name, salary/12 **as** monthly_salary **from** instructor;

- The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

E.g.

**select** contact **as** CustomerContact, name **as** CustomerName **from** customer;

# String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator "like" uses patterns that are described using following special characters:

Wildcard Characters in SQL Server

| Symbol | Description | Example |
|---|---|---|
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| _ | Represents a single character | h_t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ^ | Represents any character not in the brackets | h[^oa]t finds hit, but not hot and hat |
| - | Represents any single character within the specified range | c[a-b]t finds cat and cbt |

# String Operations

- Find the names of all instructors whose name includes the substring “mit”.

  SELECT name FROM instructor WHERE name LIKE '%mit%';

- SQL statement selects all customers with a City starting with "ber“

  SELECT * FROM customer WHERE city LIKE 'ber%';

- The following SQL statement selects all customers with a City containing the pattern “al“:

  SELECT * FROM customer WHERE city LIKE '%al%';

- SELECT * FROM department WHERE dept_name LIKE 'Ma___';

- SELECT * FROM department WHERE dept_name REGEXP '^M[aeiou]ths$';

  OR

  SELECT * FROM department WHERE dept_name LIKE 'M[aeiou]ths';

# String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
  - ‘Intro%’

    matches any string beginning with “Intro”.
  - ‘%Comp%’

    matches any string containing “Comp” as a substring.
  - ‘_ _ _’

    matches any string of exactly three characters.
  - ‘_ _ _ %’

    matches any string of at least three characters.

# Ordering the Display of Tuples

- List in alphabetic order the names of all instructors
  - **select distinct** *name*
    **from***instructor*
    **order by** *name;*
- Specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name*, *name*

# Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between $100,000 and $500,000
  - **select** *name*
    **from** *instructor*
    **where** *salary* **between 10**0000 **and 500000**;

# Set Operations

- Set operations are union, intersect, and except
- Each of the above operations automatically eliminates duplicates.
- To retain all duplicates use the corresponding multiset versions union all, intersect all and except all.

# Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010 or both

```
(SELECT courseID FROM section WHERE semester = 'Fall' AND year = 2009)

UNION

(SELECT courseID FROM section WHERE semester = 'Spring' AND year = 2010);
```

- Find courses that ran in Fall 2009 and in Spring 2010

```
(SELECT courseID FROM section WHERE semester = 'Fall' AND year = 2009)

INTERSECT

(SELECT courseID FROM section WHERE semester = 'Spring' AND year = 2010);
```

# Set Operations

- Find courses that ran in Fall 2009 but not in Spring 2010

  (SELECT courseID FROM section WHERE semester = 'Fall' AND year = 2009)

  EXCEPT

  (SELECT courseID FROM section WHERE semester = 'Spring' AND year = 2010);

# Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an **unknown value** or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

```
select name
from instructor
where salary is null;
```

# Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value
- **avg:** average value
- **min:** minimum value
- **max:** maximum value
- **sum:** sum of values
- **count:** number of values

# Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department

  select avg (salary) from instructor where deptname= ‘ComputerScience';

  SELECT COUNT(DISTINCT courseID) FROM section WHERE semester = 'Spring' AND year = 2010;

  **OR**

  SELECT COUNT(*) FROM (SELECT DISTINCT courseID FROM section WHERE semester = 'Spring' AND year = 2010) AS subquery;

- Find the number of tuples in the *department* relation

  select count(*) from department;

**MIN() Syntax**

SELECT MIN(*column_name*) FROM *table_name* WHERE*condition*;

SELECT MIN(budget) AS SmallestBudget FROM department;

**MAX() Syntax**

SELECT MAX(*column_name*) FROM *table_name*
WHERE *condition*;

SELECT MAX(budget) AS LargestBudget FROM department;

**SUM()**

The following SQL statement finds the sum of the "salary" fields in the "instructor" table:

```
SELECT        SUM(salary)
FROM instructor;
```

# Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - select deptname, avg (salary) from instructor group by deptname;
  - select deptname, avg (salary) from instructor group by deptname DESC;
  - Note: departments with no instructor will not appear in result

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 76766 | Crick | Biology | 72000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| 98345 | Kim | Elec. Eng. | 80000 |
| 12121 | Wu | Finance | 90000 |
| 76543 | Singh | Finance | 80000 |
| 32343 | El Said | History | 60000 |
| 58583 | Califieri | History | 62000 |
| 15151 | Mozart | Music | 40000 |
| 33456 | Gold | Physics | 87000 |
| 22222 | Einstein | Physics | 95000 |

| *dept_name* | *avg_salary* |
|---|---|
| Biology | 72000 |
| Comp. Sci. | 77333 |
| Elec. Eng. | 80000 |
| Finance | 85000 |
| History | 61000 |
| Music | 40000 |
| Physics | 91000 |

# Aggregate Functions – Having Clause

- Useful to state a condition that applies to groups rather than to tuples.
- Find the names and average salaries of all departments whose average salary is greater than 42000

**select** *deptname*, **avg** (*salary*)**from** *instructor* **group by** *deptname* **having avg** (*salary*) > 42000;

Note:
predicates in the **having** clause are applied **after the formation of groups** whereas predicates in the **where** clause are applied **before forming groups But not with Aggregate function**

SELECT deptname, AVG(salary) FROM instructor WHERE salary > 42000 GROUP BY deptname;

# Null Values and Aggregates

- Salary: 20000

  30000

  null

  40000

- Total all salaries

  **select sum**(*salary*) **from** *instructor;*

  - Above statement ignores null amounts

# The SQL SELECT TOP Clause

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

Note: Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records.

The following SQL statement selects the first three records from the "instructor" table

```
SELECT * FROM  instructor LIMIT 3;
SELECT TOP 3 * FROM instructor ;
```

# The SQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a short and for multiple OR conditions.

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name* IN (*value1*, *value2*, ...);koi operator na av

The following SQL statement selects all customers that are located in "Germany", "France" or "UK":

**SELECT * FROM customer WHERE Country IN ('Germany', 'India', 'UK');**

**OR**

SELECT column_name(s) FROM
table_name
WHERE column_name IN (SELECT STATEMENT);

The following SQL statement selects all customers that are from the same countries as the instructor:

SELECT * FROM customer WHERE country IN (SELECT country FROM instructor);

# Modification of the Database

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating values in some tuples in a given relation

# Modification of the Database – Deletion

- Delete all instructors

  **delete from** *instructor;*

- Delete all instructors from the Finance department

  **delete from instructor where Name = 'Finance';**

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the 100 building number.

  **delete from** *instructor* **where** *deptname* **in** (**select** *deptname* **from** *department* **where** *building* = ’100’);

# Modification of the Database – Insertion

- Add a new tuple to *course*

  **insert into** *course* **values** (’CS-437’, ’Database Systems’, ’Comp. Sci.’, 4);

- or equivalently
  **insert into** *course* (*courseID*, *title*, *deptname*, *credits*) **values** (’CS-437’, ’Database Systems’, ’Comp. Sci.’, 4);

- Add a new tuple to *student* with *tot_creds* set to null
  **insert into** *student* **values** (’3003’, ’Green’, ’Finance’, *null*);

# Modification of the Database – Updates

Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise

Write two **update** statements:

**update** *instructor* **set** *salary* = *salary* * 1.03 **where** *salary* > 100000;

**update** *instructor* **set** *salary* = *salary* * 1.05 **where** *salary* <= 100000;

# Nested Subqueries

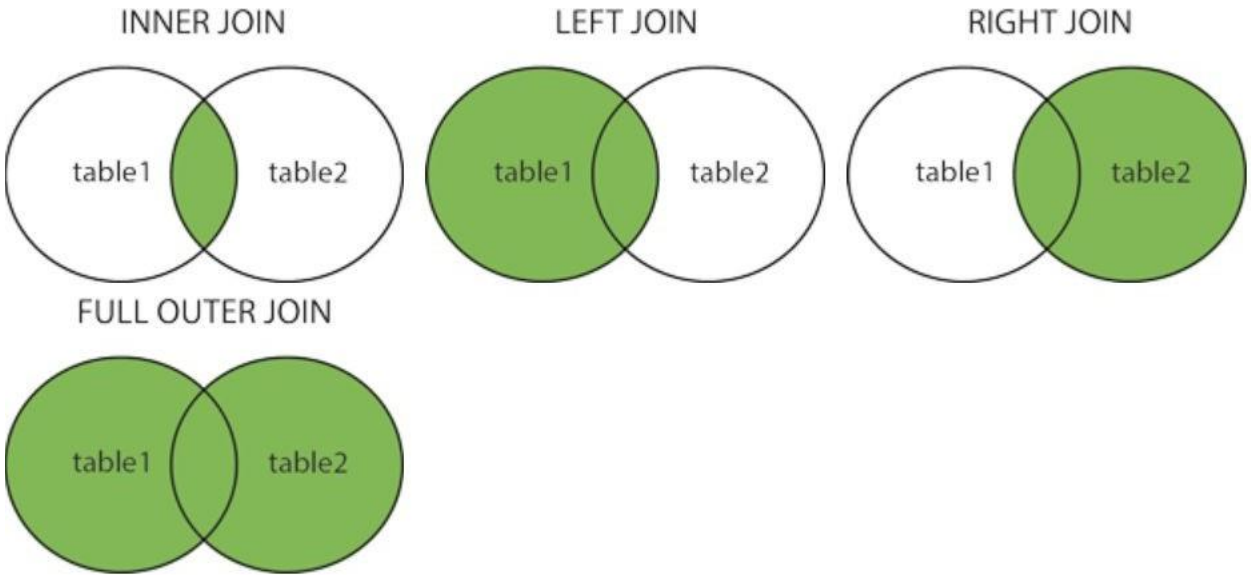
- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.
- Can use
  - With where clause
  - With from clause

# JOINS

**SQL Join** statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are as follows:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table



# INNER Join

The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

Syntax:

```
SELECT  table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

table1: First table.
table2: Second table
matching_column: Column common to both the tables.

## Student

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | HARSH | DELHI | XXXXXXXXXX | 18 |
| 2 | PRATIK | BIHAR | XXXXXXXXXX | 19 |
| 3 | RIYANKA | SILIGURI | XXXXXXXXXX | 20 |
| 4 | DEEP | RAMNAGAR | XXXXXXXXXX | 18 |
| 5 | SAPTARHI | KOLKATA | XXXXXXXXXX | 19 |
| 6 | DHANRAJ | BARABAJAR | XXXXXXXXXX | 20 |
| 7 | ROHIT | BALURGHAT | XXXXXXXXXX | 18 |
| 8 | NIRAJ | ALIPUR | XXXXXXXXXX | 19 |

## StudentCourse

| COURSE_ID | ROLL_NO |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 2 | 3 |
| 3 | 4 |
| 1 | 5 |
| 4 | 9 |
| 5 | 10 |
| 4 | 11 |

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

This query will show the names and age of students enrolled in different courses.

| COURSE_ID | NAME | Age |
|---|---|---|
| 1 | HARSH | 18 |
| 2 | PRATIK | 19 |
| 2 | RIYANKA | 20 |
| 3 | DEEP | 18 |
| 1 | SAPTARHI | 19 |

# LEFT JOIN

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax:

```
SELECT  table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```

Example Queries(LEFT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | *NULL* |
| ROHIT | *NULL* |
| NIRAJ | *NULL* |

# RIGHT JOIN

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

```
SELECT  table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| *NULL* | 4 |
| *NULL* | 5 |
| *NULL* | 4 |

# FULL JOIN

FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain *NULL* values.

```
SELECT table1.column1,table1.column2,table2.column1,.... FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
```

```
SELECT Student.NAME,StudentCourse.COURSE_ID FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | NULL |
| ROHIT | NULL |
| NIRAJ | NULL |
| NULL | 4 |
| NULL | 5 |
| NULL | 4 |

# The SQL ANY and ALL Operators

The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.

The SQL ANY Operator:

returns TRUE if ANY of the subquery values meet the condition.

ANY means that the condition will be true if the operation is true for any of the values in the range.

```
SELECT ProductName FROM Products WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity equal to 10

different nested table mate use thai shake

```
SELECT   ProductName
FROM Products
WHERE   ProductID   =   ANY
   (SELECT ProductID FROM
   OrderDetails          WHERE
   Quantity > 99);
```

SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 99.

**SQL ALL operator:**

- returns TRUE if ALL of the subquery values meet the condition
- is used with SELECT, WHERE and HAVING statements

ALL means that the condition will be true only if the operation is true for all values in the range.

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL
   (SELECT ProductID FROM
   OrderDetails WHERE
   Quantity = 10);
```

SQL statement lists the ProductName if ALL the records in the OrderDetails table has Quantity equal to 10.

# AUTO INCREMENT Field

- Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

- Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

- The following SQL statement defines the "Personid" column to be an auto-increment primary key field in the "Persons" table:

```
CREATE TABLE Persons
(
    Personid int NOT NULL AUTO_INCREMENT,
    LastName varchar(255) NOT NULL, FirstName
    varchar(255),
    Age int,
    PRIMARY KEY (Personid)
);
```

MySQL uses the AUTO_INCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.

To let the AUTO_INCREMENT sequence start with another value, use the following SQL statement:

```
ALTER TABLE Persons AUTO_INCREMENT=100;
```

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "Personid" column (a unique value will be added automatically):

```
INSERT INTO Persons
(FirstName,LastName) VALUES
('Lars','Monsen');
```

# Data control language(Authorization)

Authorization in SQL controls what actions users can perform on a database. It ensures security by granting or revoking permissions for different users or roles.

**Granting and Revoking Privileges:-**
SQL provides GRANT and REVOKE statements to manage permissions.

1. **Granting Privileges**

   **GRANT SELECT, INSERT ON employees TO user1;**
   Allows user1 to read (SELECT) and add (INSERT) records to the employees table.

   **GRANT ALL PRIVILEGES ON employees TO user1;**
   Gives user1 all permissions (SELECT, INSERT, UPDATE, DELETE, etc.) on the employees table.

   **GRANT CREATE, DROP ON DATABASE company_db TO admin_user;**
   Grants admin_user the ability to create and drop tables in company_db.

# Data control language(Authorization)

2. **Revoking Privileges**

**REVOKE INSERT ON employees FROM user1;**
Removes the INSERT privilege from user1 on the employees table.

**REVOKE ALL PRIVILEGES ON employees FROM user1;**
Removes all permissions from user1 on the employees table.