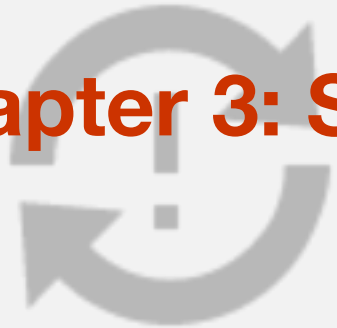
Database Engineering

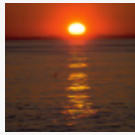


Chapter 3: SQL

Presented By:
Dr. Suvasini Panigrahi
Associate Professor, Department of CSE,

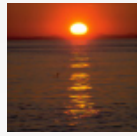
VSSUT, Burla

Database System Concepts, 5th Ed.



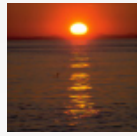
Chapter 3: SQL

- Structured Query Language (SQL)
- SQL Commands
- Types of SQL Commands
  - Data Definition Language (DDL)
  - Data Query Language (DQL)
  - Data Manipulation Language (DML)
  - Data Control Language (DCL)
  - Transaction Control Language (TCL)



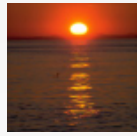
History

- SQL was invented in 1970s and was first commercially distributed by Oracle.
- This language was developed as part of System R project at the IBM San Jose Research Laboratory
- The original name was given by IBM as **Structured English Query Language**, abbreviated by the acronym **SEQUEL**.
- Renamed as **Structured Query Language (SQL)**



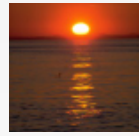
Structured Query Language

- **Structured Query Language (SQL)** is a standard Database language that is used to create, maintain, and retrieve the relational database.
- A relational database stores information in tabular form, with rows and columns representing different data attributes and the various relationships between the data values.
- We can use SQL statements to store, update, remove, search, and retrieve information from the database.
- SQL can also be used to maintain and optimize database performance.
- When data needs to be retrieved from a database, SQL is used to make the request. The DBMS processes the SQL query, retrieves the requested data and returns it to us.
- SQL is not case sensitive. But, it is a recommended practice to write keywords (like SELECT, UPDATE, CREATE, etc.) in capital letters and user-defined things (like table name, column name, etc.) in small letters.

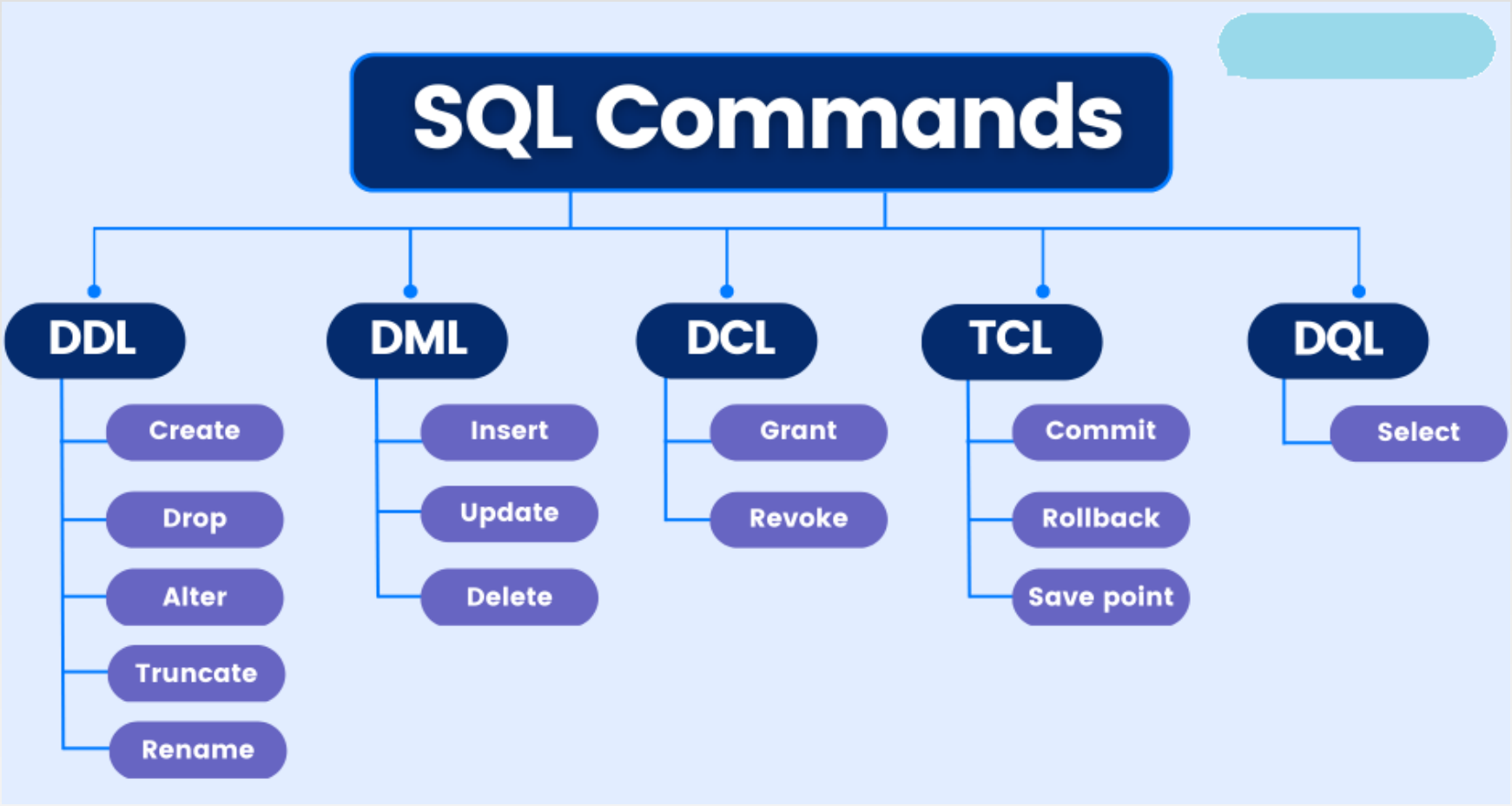


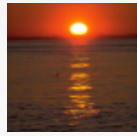
SQL Commands

- Structured query language (SQL) commands are specific keywords or SQL statements that developers use to manipulate the data stored in a relational database.
- The SQL commands can be categorized as follows:
  1. **Data Definition Language (DDL)**
  2. **Data Query Language (DQL)**
  3. **Data Manipulation Language (DML)**
  4. **Data Control Language (DCL)**
  5. **Transaction Control Language (TCL)**



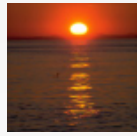
Types of SQL Commands





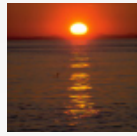
Types of SQL Commands

1. **Data Definition Language (DDL)**
   - Data Definition Language (DDL) refers to SQL commands that are used to design the database structure.
   - Database engineers use DDL to create and modify database objects based on the business requirements.
   - For example, the database engineer uses the CREATE command to create database objects such as tables, views, and indexes.
2. **Data Query Language (DQL)**
   - Data Query Language (DQL) consists of instructions for retrieving data stored in relational databases.
   - Software applications use the SELECT command to filter and return specific results from a database table.

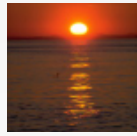


Types of SQL Commands

3. **Data Manipulation Language (DML)**
   - Data Manipulation Language (DML) statements are used to write new information or modify existing records in a relational database.
   - For example, an application uses the INSERT command to store a new record in the database.
4. **Data Control Language (DCL)**
   - Database administrators use the Data Control Language (DCL) to manage or authorize database access for various users.
   - For example, they can use the GRANT command to permit certain applications to manipulate one or more tables.
5. **Transaction Control Language (TCL)**
   - Transaction Control Language (TCL) commands are used to manage transactions in the database.
   - For example, the database uses the ROLLBACK command to undo an erroneous transaction.



Data Definition Language (DDL)

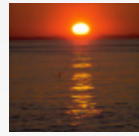


Data Definition Language (DDL)

DDL allows the specification of not only a set of relations but also information about each relation, including:

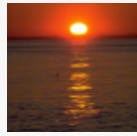
- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

NOTE: The purpose of creating an index on a particular table in a database is to make it faster to search through the table and find the row(s) that we want in lesser time.



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 decimal number, with user-specified precision of *p* digits, with *n* 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.



Bank Database Schema

branch (*branch_name*, *branch_city*, *assets*)

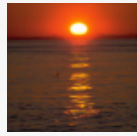
customer (*customer_name*, *customer_street*, *customer_city*)

loan (*loan_number*, *branch_name*, *amount*)

borrower (*customer_name*, *loan_number*)

account (*account_number*, *branch_name*, *balance*)

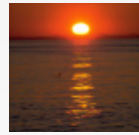
depositor (*customer_name*, *account_number*)



SQL CREATE TABLE Construct

- SQL CREATE TABLE Statement is used to create a new table in a database.
- Users can define the table structure by specifying the column's name and data type in the CREATE TABLE command.
- This statement also allows to **create table with constraints,** that define the rules for the table.
- Users can create tables in SQL and insert data at the time of table creation.
- **Syntax:** To create a table in SQL, use the **CREATE TABLE syntax**:

```
CREATE table table_name
 (
 Column1 datatype (size),
 column2 datatype (size),
 .
 .
 columnN datatype(size)
 );
```



CREATE TABLE Construct with Integrity Constraints

- An SQL relation is defined using the **CREATE TABLE** command:

  CREATE TABLE table_name
  (column1 data_type(size) constraint_name,
  column2 data_type(size) constraint_name,
  column3 data_type(size) constraint_name,
  ....);

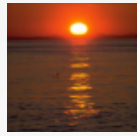
  table_name: Name of the table to be created.

  data_type: Type of data that can be stored in the field.

  constraint_name: Name of the constraint. For example - NOT NULL, UNIQUE, PRIMARY KEY, etc.

- **Example:** Create table "Employee" with EmployeeID as its primary key.

```
CREATE TABLE Employee (
EmployeeID INT PRIMARY KEY,
FirstName VARCHAR(50),
LastName VARCHAR(50),
Department VARCHAR(50),
Salary DECIMAL(10, 2)
);
```

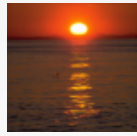


Drop and Alter Table Constructs

- SQL provides command to DROP an existing table completely from a database. The **DROP TABLE** command deletes all information about the dropped relation from the database.
- **Syntax:** The basic syntax of this DROP TABLE statement is as follows – **DROP TABLE table_name;**
- The **ALTER TABLE** command is used to add attributes to an existing relation:
- **Syntax: 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.*
- The **ALTER TABLE** command can also be used to drop attributes of a relation:
- **Syntax: ALTER TABLE** *r* **drop** *A;*

  where, *A* is the name of an attribute of relation *r.*

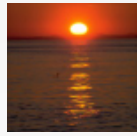


Truncate Table Command

- The SQL truncate table command is used to empty a table.
- We can also use drop table command to delete a table but it will remove the complete table structure from the database and it is needed to re-create this table once again if we require the table again.
- The truncate table command deletes the data inside the table, but not the table itself.
- **Syntax:** The basic syntax of a truncate table command is as follows.

  truncate table table_name;
- **Example:** The following SQL statement truncates the table "Categories":

  truncate table Categories;



Rename Operation using AS Clause

- The SQL allows renaming attributes using the **as** clause:

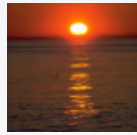
  old-name as new-name

- The AS command is used to rename a column or table with an alias.
- An alias only exists for the duration of the query.

Alias for Columns

- **Example:** Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id.*

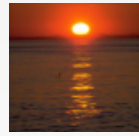
select *customer_name, borrower.loan_number* **as** *loan_id, amount*
from *borrower, loan*
where *borrower.loan_number = loan.loan_number;*



Rename Operation using AS Clause

- **Alias for Tables**
- The following SQL statement selects all the orders from the customer with CustomerName = “Ram”.
- We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively.

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName = “Ram" AND c.CustomerID =
o.CustomerID;
```

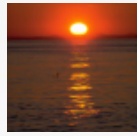


Rename Table Command

- **Syntax of ALTER TABLE statement in SQL**

  ALTER TABLE old_table_name RENAME TO new_table_name;
- In the Syntax, we have to specify the RENAME TO keyword after the old name of the table.
- **Examples of ALTER TABLE statement in SQL**

  ALTER TABLE Bikes RENAME TO Bikes_Details;



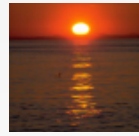
Rename Table Operation using Alter Table Command

- **Syntax of ALTER TABLE statement in SQL**

  ALTER TABLE old_table_name RENAME TO new_table_name;
- In the Syntax, we have to specify the RENAME TO keyword after the old name of the table.

- **Examples of ALTER TABLE statement in SQL**

  ALTER TABLE Bikes RENAME TO Bikes_Details;



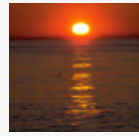
Rename Column Operation using Alter Table Command

- Columns can also be given a new name with the use of ALTER TABLE.
- **Syntax of ALTER TABLE statement in SQL**

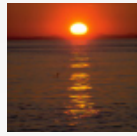
  ALTER TABLE table_name RENAME old_col_name TO new_col_name;

- To change the column name of the existing table you have to use Column keyword before writing the existing column name to change it.
- **Example:** To change the name of column name to FIRST_NAME in table Student we have to write the following SQL statement:

  ALTER TABLE Student RENAME COLUMN Column_NAME TO FIRST_NAME;



Data Query Language (DQL)

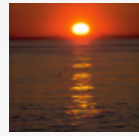


Basic Select Query Structure

- 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.
- This query is equivalent to the a combination of cross product, selection and projection operations in relational algebra.
- The result of an SQL query is a relation.

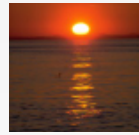


The select Clause

- The **select** clause list the attributes desired in the result of a query.
  - It corresponds to the projection operation of the relational algebra.
- Example: find the names of all branches in the *loan* relation:

  select *branch_name*
  from *loan;*
- In the relational algebra, the query would be:

  Π_{branch_name} (*loan*)
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*
  - Some people use upper case wherever we use bold font.



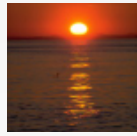
The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- Find the names of all branches in the *loan* relations, and remove duplicates.

```
select distinct branch_name
from loan;
```

- The keyword **all** specifies that duplicates not be removed.

```
select all branch_name
from loan;
```



The select Clause (Cont.)

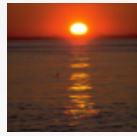
- An asterisk in the select clause denotes “all attributes”

  select *
  from *loan;*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.
- The query:

  select *loan_number, branch_name, amount* * 100
  from *loan;*

  would return a relation that is the same as the *loan* relation, except that the value of the attribute *amount* is multiplied by 100.

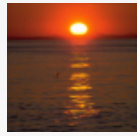


The where Clause

- The **where** clause specifies conditions that the result must satisfy.
  - Corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

```
select loan_number
from loan
where branch_name = 'Perryridge'  and amount > 1200;
```

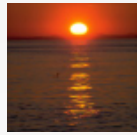
- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.



The where Clause (Cont.)

- SQL can includes a **between** comparison operator in the **where clause**.
- Example: Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)

```
select loan_number
from loan
where amount between 90000 and 100000;
```



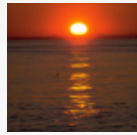
The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower* × *loan*. It is equivalent to the following SQL statement:

```
select *
from borrower, loan;
```

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

```
select customer_name, borrower.loan_number, amount
    from borrower, loan
    where   borrower.loan_number = loan.loan_number  and
            branch_name = 'Perryridge';
```



Tuple Variables

- Tuple variables are defined in the **from** clause via the use of the **as** clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

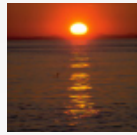
select *customer_name, T.loan_number, S.amount*
from *borrower* **as** *T, loan* **as** *S*
where *T.loan_number = S.loan_number;*

- Find the names of all branches that have greater assets than some branch located in Brooklyn.

select distinct *T.branch_name*
from *branch* **as** *T, branch* **as** *S*
where *T.assets > S.assets* **and** *S.branch_city* = 'Brooklyn' *;*

- Keyword **as** is optional and may be omitted

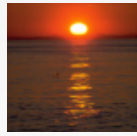
borrower **as** *T* ≡ *borrower T*



String Operations

- SQL includes a string-matching operator for comparisons on character strings.
- We can use the **like operator** of SQL to search sub-strings. The **like operator** is used with the where clause to search a pattern in a string of columns.
- The operator “like” uses patterns that are described using two special characters:
  - **Percentage sign(%):** It represents zero, one, or multiple characters of variable length.
  - **Underscore (_):** It represents one, single character of fixed length.
- The Syntax of a LIKE Clause is
  - **select column1, column2 from table_name where column_name like [expression];**
- Find the names of all customers whose street includes the substring “Main”.

```
select customer_name
from customer
where customer_street like '%Main%';
```

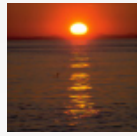


Ordering the Display of Tuples

- List in alphabetic order the names of all customers having a loan in Perryridge branch.

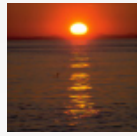
```
select distinct customer_name
from    borrower, loan
where borrower loan_number = loan.loan_number and
        branch_name = 'Perryridge'
order by customer_name;
```

- We may specify **desc** for descending order or **asc** for ascending order. For each attribute; ascending order is the default order.
  - Example: **order by** *customer_name* **desc;**



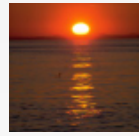
Set Operations

- The set operations **UNION**, **INTERSECT**, and **EXCEPT** operate on relations and correspond to the relational algebra operations ∪, ∩, –.
- Each of the above operations automatically eliminates duplicates. To retain all duplicates use **UNION ALL**, **INTERSECT ALL** and **EXCEPT ALL**.
- There are certain rules which must be followed to perform operations using SET operators in SQL. The rules are as follows:
  - The number and order of columns must be the same.
  - Data types must be compatible.



Set Operations

- **UNION:**
  - UNION will be used to combine the result of two select statements.
  - Duplicate rows will be eliminated from the results obtained after performing the UNION operation.
- **INTERSECT:**
  - It is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements.
- **EXCEPT (MINUS):**
  - It displays the rows which are present in the first query but absent in the second query.



Examples of Set Operations

- Find all customers who have a loan, an account, or both:

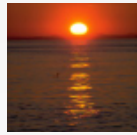
  (select *customer_name* **from** *depositor*)
  union
  (select *customer_name* **from** *borrower);*

- Find all customers who have both a loan and an account.

  (**select** *customer_name* **from** *depositor*)
  intersect
  (select *customer_name* **from** *borrower);*

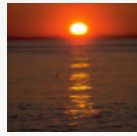
- Find all customers who have an account but no loan.

  (**select** *customer_name* **from** *depositor*)
  except
  (select *customer_name* **from** *borrower);*



Aggregate Functions

- An aggregate function is a function that performs a calculation on a set of values, and returns a single value.
- Aggregate functions are often used with the GROUP BY clause of the SELECT statement to summarize the grouped data.
- The GROUP BY clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.
- Aggregate function operates on non-NULL values only (except COUNT).
- The most commonly used SQL aggregate functions are:
  - MIN() - returns the smallest value within the selected column
  - MAX() - returns the largest value within the selected column
  - COUNT() - returns the number of rows in a set
  - SUM() - returns the total sum of a numerical column
  - AVG() - returns the average value of a numerical column



Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

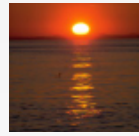
```
select avg (balance)
from account
where branch_name = 'Perryridge';
```

- Find the number of tuples in the *customer* relation.

```
select count (*)
from customer;
```

- Find the number of depositors in the bank.

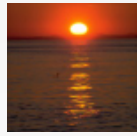
```
select count (distinct customer_name)
from depositor;
```



Aggregate Functions – Group By

- Find the number of depositors for each branch.

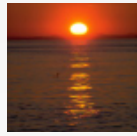
select *branch_name,* **count (distinct** *customer_name)*
from *depositor, account*
where *depositor.account_number = account.account_number*
group by *branch_name;*



Having Clause

- The **HAVING clause** is similar to the WHERE clause; both are used to filter rows in a table based on specified criteria. However, the HAVING clause is used to filter grouped rows instead of single rows.
- The rows are grouped together by the GROUP BY clause and the HAVING clause must always be followed by the GROUP BY clause.
- Moreover, the HAVING clause can be used with aggregate functions such as COUNT(), SUM(), AVG(), etc., whereas the WHERE clause cannot be used with them.
- **Syntax:**

```
SELECT column1, aggregate_function(column)
FROM table_name
GROUP BY column1
HAVING condition;
```



Having Clause

- SELECT COUNT(customer_id), country FROM Customers GROUP BY country HAVING COUNT(customer_id) > 1;

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT COUNT(customer_id), country
FROM Customers
GROUP BY country
HAVING COUNT(customer_id) > 1;
```

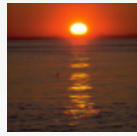
COUNT(customer_id)	country
2	UK
2	USA

Having Clause

- Find the names of all branches where the average account balance is more than $1,200.

```
select branch_name, avg(balance)
    from account
    group by branch_name
    having avg(balance) > 1200
```

Note: Predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups.



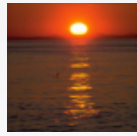
Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes.
- **SQL IS NULL** is a logical operator that checks for NULL values in a column.
- **IS NULL Operator in SQL** is used to test for empty or missing values in a column.
- It checks if the specified expression evaluates to NULL, if it evaluates to NULL, it returns TRUE; otherwise, it returns FALSE.
- **Syntax:** The SQL IS NULL syntax is:

  SELECT * **FROM** table_name **WHERE** column_name **IS NULL**;

- **Example:** Find all loan number which appear in the *loan* relation with null values for *amount.*

  select *loan_number*
  from *loan*
  where *amount* **is null**;

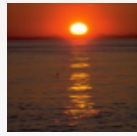


Null Values and Aggregates

- Total of all loan amounts

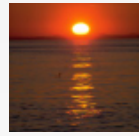
  select sum(*amount*)
  from *loan;*

  - The above statement ignores null amounts
  - The result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.



Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A subquery is a SELECT statement that is nested within another SELECT statement and which return intermediate results.
- SQL executes innermost subquery first, then goes to the next level.
- Nesting of queries is a way to perform complex queries by embedding one query within another.
- The outer query can apply some conditions on the results of the inner query.
- A common use of subqueries is to perform tests for set membership, set comparisons, etc.



SQL IN Operator

- The **SQL IN operator** filters data based on a list of specific values. In general, we can only use one condition in the **WHERE clause**, but the IN operator allows us to specify multiple values.
- **IN Operator Syntax**

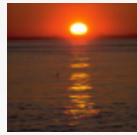
  The syntax of the IN operator is as follows:

  SELECT *column_name(s)*
  FROM *table_name*
  WHERE *column_name* IN (*value1*, *value2*, ...);

  Here, we select the column *column_name* from the table *table_name* where the condition is checked in all the values passed with the IN operator.
- **Example 1:** Write a SQL Query to get Fname and Lname of employees who have address in Delhi and Himachal.

  Query:

  SELECT Fname, Lname FROM employee **WHERE** Address **IN** ('Delhi', 'Himachal');

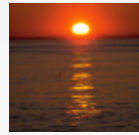


SQL NOT IN Operator

- By using the NOT keyword in front of the IN operator, it returns all records that are NOT in any of the values in the list.
- **Example:**

  Return all customers that are NOT from 'Germany', 'France', or 'UK':
- SQL Query:

```
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');
```



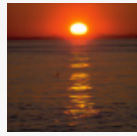
Example Query

- Find all customers who have both an account and a loan at the bank.

```
select distinct customer_name
    from borrower
    where customer_name in (select customer_name
                            from depositor);
```

- Find all customers who have a loan at the bank but do not have an account at the bank.

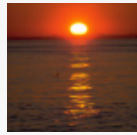
```
select distinct customer_name
    from borrower
    where customer_name not in (select customer_name
                                from depositor);
```



Set Comparison

- Find all branches that have greater assets than some branch located in Brooklyn.

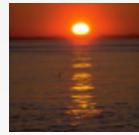
```
select distinct  T.branch_name
      from branch as T, branch as S
      where  T.assets > S.assets and
                  S.branch_city = 'Brooklyn';
```



Example Query

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name
    from branch
    where assets > all
        (select assets
        from branch
        where branch_city = 'Brooklyn');
```

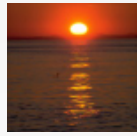


Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

  (**select** *customer_name, borrower.loan_number, branch_name*
  from *borrower, loan*
  where *borrower.loan_number = loan.loan_number*);

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

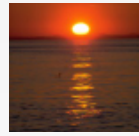


View Definition

- In SQL, a view is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table.
- A view is defined using the **create view** statement which has the form

  create view *v* **as** <query expression>;

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.



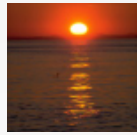
Example Queries

- A view consisting of branches and their customers.

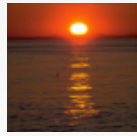
```
create view all_customer as
        (select branch_name, customer_name
         from depositor, account
        where depositor.account_number =
              account.account_number )
        union
        (select branch_name, customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number );
```

- Find all customers of the Perryridge branch

```
select customer_name
      from all_customer
      where branch_name = 'Perryridge‘;
```

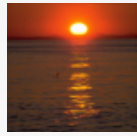


Data Manipulation Language (DML)



Data Manipulation Language

- DML is an abbreviation of Data Manipulation Language.
- The DML commands in Structured Query Language change the data present in a database.
- We can easily insert, update and delete the existing records from the database using DML commands.
- **Following are the three main DML commands in SQL:**
  1. INSERT Command
  2. UPDATE Command
  3. DELETE Command



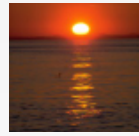
DELETE DML Command

- DELETE is a DML command which allows SQL users to remove single or multiple existing records from the database tables.

- This command of Data Manipulation Language does not delete the stored data permanently from the database. We use the WHERE clause with the DELETE command to select specific rows from the table.

- Syntax of DELETE Command

DELETE FROM Table_Name WHERE condition;



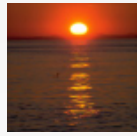
Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

```
delete from account
where branch_name = 'Perryridge'
```

- Delete all accounts at every branch located in the city ‘Needham’.

```
delete from account
where branch_name in (select branch_name
                      from branch
                      where branch_city = 'Needham')
```

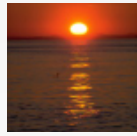


Example Query

- Delete the record of all accounts with balances below the average at the bank.

delete from *account*
where *balance* < (**select avg** (*balance*)
from *account*)

  - Problem: as we delete tuples from deposit, the average balance changes
  - Solution used in SQL:
    1. First, compute **avg** balance and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

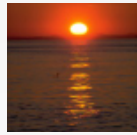


INSERT DML Command

- INSERT is an important data manipulation command in Structured Query Language, which allows users to insert data in database tables.

- **Syntax of INSERT Command:**

INSERT INTO TABLE_NAME (column_Name1 , column_Name2 , column_Name3 , ... column_NameN) VALUES (value_1, value_2, value_3, ... value_N);



Modification of the Database – Insertion

- Add a new tuple to *account*

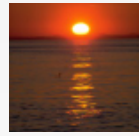
  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  or equivalently

  insert into *account* (*branch_name, balance, account_number*)
  values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

  insert into *account*
  values ('A-777','Perryridge', *null*)

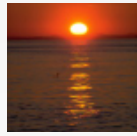


Modification of the Database – Insertion

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.

```
insert into account
  select loan_number, branch_name,  200
  from loan
  where branch_name = 'Perryridge'

insert into depositor
  select customer_name, loan_number
  from loan, borrower
  where branch_name = 'Perryridge'
        and loan.account_number = borrower.account_number
```

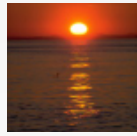


UPDATE DML Command

- UPDATE is another most important data manipulation command in Structured Query Language, which allows users to update or modify the existing data in database tables.

- Syntax of UPDATE Command:

UPDATE Table_name SET [column_name1= value_1, ….., column_nameN = value_N] WHERE CONDITION;

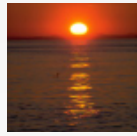


Modification of the Database – Updates

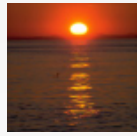
- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

update *account*
set *balance* = *balance* * 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* * 1.05
where *balance* ≤ 10000

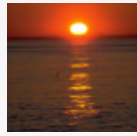


Data Control Language (DCL)



Data Control Language (DCL)

- DCL stands for Data Control Language.
- It is a component of SQL (Structured Query Language) that is used to implement security on database objects.
- DCL includes the following two commands which allow permissions to be added or removed from database users respectively:
  1. GRANT Command
  2. REVOKE Command

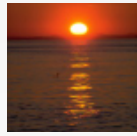


Data Control Language (DCL)

- **GRANT Command:**
  - This command allows the administrator to provide particular privileges or permissions over a database object, such as a table, view, or procedure.
  - It can provide user access to perform certain database or component operations.
  - **Syntax:**

    GRANT privilege_type [(column_list)] ON [object_type] object_name TO user;

    Example:
  - **GRANT SELECT ON** student **TO** Aman;
  - This command will allow Aman to implement the SELECT queries on the student table.

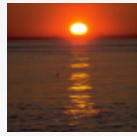


Data Control Language (DCL)

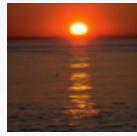
- **Revoke Command:**
  - The REVOKE command enables the database administrator to remove the previously provided privileges or permissions from a user over a database or database object, such as a table, view, or procedure.
  - The REVOKE commands prevent the user from accessing or performing a specific operation on an element in the database.
  - **Syntax:**

    REVOKE [GRANT OPTION FOR] privilege_type [(column_list)] ON [object_type] object_name FROM user;

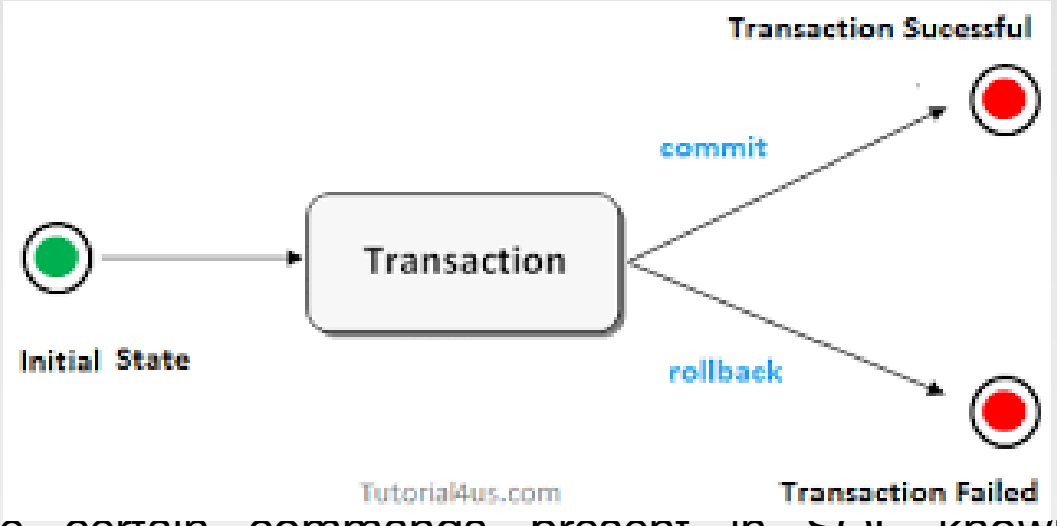
    Example:
  - **REVOKE SELECT ON** student **FROM** Aman;
  - This will stop the user Aman from implementing the SELECT query on the student table.



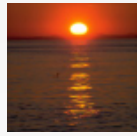
Transaction Control Language (TCL)



- Transactions group a set of tasks into a single execution unit.
- Each transaction begins with a specific task and ends when all the tasks in the group are successfully completed.
- If any of the tasks fail, the transaction fails.
- Therefore, a transaction has only two results: success or failure.



- There are certain commands present in SQL known as TCL commands that help the user manage the transactions that take place in a database.
- COMMIT, ROLLBACK and SAVEPOINT are the most commonly used TCL commands in SQL.



List of TCL Commands

Command	Description	Syntax
BEGIN TRANSACTION	Starts a new transaction	BEGIN TRANSACTION [transaction_name];
COMMIT	Saves all changes made during the transaction	COMMIT;
ROLLBACK	Undoes all changes made during the transaction	ROLLBACK;
SAVEPOINT	Creates a savepoint within the current transaction	SAVEPOINT savepoint_name;

The SAVEPOINT command is a feature in SQL that allows us to set a specific point in a transaction that we can roll back to, without having to roll back the entire transaction.