DBMS DBMS is software used to manage, manipulate, and retrieve data stored in databases.

Relational Database Management System (RDBMS) RDBMS is a type of DBMS that stores and retrieves data in a tabular form with relationships defined between tables.

SQL (Structured Query Language) SQL is a standard language for interacting with RDBMS. It is used for querying, updating, and managing relational databases.

```
SELECT * FROM table_name WHERE condition;
```

DDL (Data Definition Language)

DDL is a subset of SQL used to define database structure and schema.

CREATE TABLE CREATE TABLE defines a new table and its columns and data types.

```
CREATE TABLE Students (ID INT, Name VARCHAR(100), Age INT);
```

ALTER TABLE ALTER TABLE modifies the structure of an existing table.

```
ALTER TABLE Students ADD COLUMN Email VARCHAR(255);
```

DROP TABLE DROP TABLE deletes an existing table and its data from the database.

```
DROP TABLE Students;
```

DML (Data Manipulation Language) DML is used to manipulate data in the database. This includes inserting, updating, and deleting data.

Example:

- **INSERT:** Adds new records to a table.

```
INSERT INTO Students (StudentID, StudentName) VALUES (1, 'Alice');
```

- **UPDATE:** Modifies existing records in a table.

```
UPDATE Students SET StudentName = 'Alicia' WHERE StudentID = 1;
```

- **DELETE:** Removes records from a table.

```
DELETE FROM Students WHERE StudentID = 1;
```

DQL (Data Query Language) DQL is a subset of SQL used to retrieve data from one or more tables.

```
SELECT column1, column2 FROM table_name WHERE condition;
```

DCL (Data Control Language) DCL is a subset of SQL used to control access to data stored in the database. **Example:**

- **GRANT:** Gives a user specific privileges.

```
GRANT SELECT, INSERT ON Students TO user_name;
```

- **REVOKE:** Removes specific privileges from a user.

```
REVOKE SELECT, INSERT ON Students FROM user_name;
```

TCL (Transaction Control Language) TCL is a subset of SQL used to manage transactions within the database. **Example:**

- **COMMIT:** Saves all changes made during the current transaction.

```
COMMIT;
```

- **ROLLBACK:** Undoes all changes made during the current transaction.

```
ROLLBACK;
```

- **SAVEPOINT:** Sets a point within a transaction to which you can later roll back.

```
SAVEPOINT savepoint1;
```

- **ROLLBACK TO SAVEPOINT:** Rolls back the transaction to a specified savepoint.

```
ROLLBACK TO SAVEPOINT savepoint1;
```

Structure of Relational Databases

Table A table is a collection of related data entries and it consists of columns and rows.

```
CREATE TABLE Students (StudentID int, StudentName varchar(255), Age int);
```

Column A column in a table represents a set of data values of a particular type, one for each row.

```
ALTER TABLE Students ADD COLUMN Email varchar(255);
```

Row A row is a single record in a table, containing data in each column field.

```
INSERT INTO Students (StudentID, StudentName, Age) VALUES (1, 'John Doe', 22);
```

Schema A schema is the structure of a database defined by a set of formulas (sentences) called integrity constraints.

```
CREATE SCHEMA IF NOT EXISTS School; // Creates a new schema named School.
```

Relation A relation, or table, is a set of tuples, or rows, sharing the same attributes, or columns.

```
SELECT * FROM Students; // Retrieves the relation of all students.
```

Attribute An attribute is a property or characteristic of an entity, often corresponding to a column in a table.

```
SELECT StudentName FROM Students; // Retrieves the 'StudentName' attribute.
```

Tuple A tuple is a single entry in a table, which may contain a value for each column, synonymous with a row.

```
SELECT * FROM Students WHERE StudentID = 1; // Retrieves a single tuple.
```

Domain A domain is the set of allowable values that a column can contain, defined by its data type.

```
CREATE DOMAIN AgeDomain AS INT CHECK (VALUE > 0 AND VALUE < 130);
```

Integrity Constraint Integrity constraints are rules that ensure the accuracy and consistency of data within a relational database.

```
ALTER TABLE Students ADD CONSTRAINT AgeConstraint CHECK (Age >= 18);
```

Database Users and Administrators

Database Users Database Users are individuals who interact with the database through applications or direct queries.

```
SELECT * FROM Employees; // A query a database user might run.
```

Database Administrators Database Administrators (DBAs) are responsible for the overall management of the database environment.

```
CREATE USER 'dbadmin' IDENTIFIED BY 'password'; // DBA creating a new user.
```

Casual Users Casual Users access the database occasionally and may need different information each time.

```
SELECT OrderDate, CustomerName FROM Orders WHERE OrderID = 10248; // Casual user query.
```

Naive or Parametric Users Naive or Parametric Users interact with the database by using pre-defined operations.

```
EXECUTE GetCustomerOrders @CustomerID = 1; // Parametric user running a stored procedure.
```

Sophisticated Users Sophisticated Users utilize complex queries and may use tools like report generators.

```
WITH MonthlySales AS (SELECT * FROM Sales WHERE DatePart(month, SaleDate) = 5) SELECT * FROM MonthlySales; // Sophisticated user query.
```

Application Programmers Application Programmers write application programs that interact with the database using APIs or query languages.

```
db.Query("SELECT * FROM Products WHERE Price > 100"); // Application programmer code.
```

Database Schema

Logical Schema A logical schema defines the structure of the database as perceived by the end user.

```
CREATE TABLE Students (ID INT, Name VARCHAR(100), Age INT);
```

Physical Schema A physical schema describes the physical storage structure of the database on the storage media.

```
ALTER TABLE Students ADD COLUMN Gender CHAR(1);
```

Schema Object Schema objects are logical structures created by users to contain data or to reference data.

```
CREATE INDEX idx_student_name ON Students (Name);
```

Database Instance A database instance is a set of memory structures and background processes that access a database's files.

When a database server starts, it creates an instance of the database.

Schema Evolution Schema evolution refers to the ability to change the schema without affecting the existing data and applications.

```
ALTER TABLE Students RENAME COLUMN Age TO BirthYear;
```

Schema Migration Schema migration involves moving the schema from one database environment to another.

INSERT INTO NewDB.Students SELECT * FROM OldDB.Students;

The Entity-Relationship Model

Entity An Entity represents a real-world object or concept with distinct existence in the domain.

CREATE TABLE Student (StudentID INT, Name VARCHAR(100), Age INT); // SQL table for 'Student' entity.

Attribute Attributes are properties or characteristics of an entity, describing various aspects.

ALTER TABLE Student ADD COLUMN Email VARCHAR(255); // Adding 'Email' attribute to 'Student' entity.

Relationship A Relationship represents an association among two or more entities within the system.

CREATE TABLE Enrollments (StudentID INT, CourseID INT); // 'Enrollments' table represents a relationship.

ER Diagram An ER Diagram visually represents the entities, attributes, and relationships of a database schema.

Cardinality Cardinality defines the numerical relationship between two entities, such as one-to-one or many-to-many.

Participation Constraint Participation Constraint specifies if the existence of an entity depends on its being related to another entity.

ER Diagram ER Model is used to model the logical view of the system from a data perspective .

1. **Entities** are depicted as **rectangles** in an ER diagram, indicating objects or concepts such as a "Student".
2. **Attributes** are illustrated using **ellipses**, representing properties or details about an entity like "Name" or "Date of Birth".
3. **Relationships** between entities are shown with **diamonds**, highlighting how entities like "Student" and "Course" are connected.
4. **Lines** serve to link attributes to their entities and entities to their relationships, demonstrating the connections between them.
5. **Multi-Valued Attributes** are drawn with **double ellipses**, signifying attributes that can hold multiple values, such as a list of phone numbers for a student.
6. **Weak Entities** are represented with **double rectangles**, indicating entities that need an additional key from another entity to be uniquely identified, like a "Dependent" needing an "Employee" reference.

Relational Algebra

Selection Selection is a unary operation that filters tuples based on a condition.

σ_{age > 30}(Employees) // Selects employees older than 30.

Projection Projection is a unary operation that extracts specific columns from a relation.

Π_{name, salary}(Employees) // Projects name and salary columns.

Union Union is a binary operation that combines tuples from two relations without duplicates.

R ∪ S // Union of relations R and S.

Intersection Intersection is a binary operation that returns tuples common to both relations.

R ∩ S // Intersection of relations R and S.

Difference Difference is a binary operation that returns tuples in one relation but not in the other.

R - S // Tuples in R but not in S.

Cartesian Product Cartesian Product is a binary operation that returns all possible pairs of tuples from two relations.

R × S // Cartesian product of R and S.

Join Join is a binary operation that combines related tuples from two relations based on a condition.

R ⋈_{R.id = S.id} S // Join R and S where R.id equals S.id.

Theta Join Theta Join is a join operation that uses a generic condition (θ) for matching tuples.

R ⋈_{R.age > S.age} S // Theta join on age comparison.

Natural Join Natural Join is a binary operation that joins two relations on common attributes.

R ⋈ S // Natural join of R and S on common columns.

Division Division is a binary operation that returns tuples from one relation that match all tuples in another.

R ÷ S // Division of R by S.

Rename Rename is a unary operation that changes the name of a relation or its attributes.

ρ_{x ← R}(R) // Renames relation R to x.

Basic Structure of SQL Queries

SELECT Statement The SELECT statement is used to select data from a database.

```
SELECT column1, column2 FROM table_name;
```

WHERE Clause The WHERE clause is used to filter records based on specified conditions.

```
SELECT * FROM table_name WHERE condition;
```

FROM Clause The FROM clause specifies the table to select or delete data from.

```
SELECT column_name FROM table_name;
```

JOIN Clause JOIN clause is used to combine rows from two or more tables, based on a related column.

```
SELECT columns FROM table1 JOIN table2 ON table1.column = table2.column;
```

GROUP BY Clause GROUP BY clause groups rows that have the same values in specified columns into summary rows.

```
SELECT column_name, COUNT(*) FROM table_name GROUP BY column_name;
```

ORDER BY Clause ORDER BY clause is used to sort the result set in ascending or descending order.

```
SELECT column1, column2 FROM table_name ORDER BY column1 ASC, column2 DESC;
```

INSERT INTO Statement INSERT INTO statement is used to insert new records into a table.

```
INSERT INTO table_name (column1, column2) VALUES (value1, value2);
```

UPDATE Statement UPDATE statement is used to modify existing records in a table.

```
UPDATE table_name SET column1 = value1 WHERE condition;
```

DELETE Statement DELETE statement is used to delete existing records from a table.

```
DELETE FROM table_name WHERE condition;
```

Normal Forms

Normalization Normalization is the process of organizing data to minimize redundancy and improve data integrity.

Denormalization Denormalization is the process of adding redundancy to a database to improve read performance.

Let's explain all the normal forms with an example table to illustrate the concepts. We'll start with a simple table and progressively normalize it to satisfy higher normal forms.

Example Table: Student Courses

Consider a table storing information about students and the courses they are enrolled in, along with the instructor for each course:

StudentID	**StudentName**	**CourseID**	**CourseName**	**InstructorName**	**InstructorPhone**
1	Alice	C101	Database	Dr. Smith	555-1234
2	Bob	C101	Database	Dr. Smith	555-1234
1	Alice	C102	Networks	Dr. Johnson	555-5678

1NF (First Normal Form) Condition: A table is in 1NF if all columns contain only atomic, indivisible values, and each column contains values of a single type.

1NF Example: Our table already satisfies 1NF because each cell contains a single value, and there are no repeating groups.

2NF (Second Normal Form) Condition: A table is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the primary key.

In our case, the primary key is a composite key of StudentID and CourseID.

Problems in 1NF:

- StudentName depends only on StudentID.
- CourseName, InstructorName, and InstructorPhone depend only on CourseID.

2NF Example: To achieve 2NF, we need to remove partial dependencies by splitting the table into two tables:

Students Table:

StudentID	StudentName
1	Alice
2	Bob

Courses Table:

CourseID	CourseName	InstructorName	InstructorPhone
C101	Database	Dr. Smith	555-1234
C102	Networks	Dr. Johnson	555-5678

Enrollments Table:

StudentID	CourseID
1	C101
2	C101
1	C102

3NF (Third Normal Form) Condition: A table is in 3NF if it is in 2NF and there are no transitive dependencies (a non-key attribute should not depend on another non-key attribute).

Problems in 2NF:

- InstructorPhone depends on InstructorName, not directly on CourseID.

3NF Example: To achieve 3NF, we need to remove transitive dependencies by further splitting the tables:

Instructors Table:

InstructorName	InstructorPhone

Dr. Smith	555-1234
Dr. Johnson	555-5678

Courses Table (Updated):

CourseID	**CourseName**	**InstructorName**
C101	Database	Dr. Smith
C102	Networks	Dr. Johnson

Other tables remain the same:

- Students Table
- Enrollments Table

BCNF (Boyce-Codd Normal Form) Condition: A table is in BCNF if it is in 3NF and every determinant is a candidate key. **Problems in 3NF**:

- If there were any overlapping candidate keys, we would need to ensure that every determinant (attribute determining another attribute) is a candidate key.

BCNF Example: Our tables from 3NF already satisfy BCNF, as there are no overlapping candidate keys.

4NF (Fourth Normal Form) Condition: A table is in 4NF if it is in BCNF and there are no multi-valued dependencies (an attribute should not have a one-to-many relationship with two independent attributes).

4NF Example: Our current structure already satisfies 4NF because there are no multi-valued dependencies.

5NF (Fifth Normal Form) Condition: A table is in 5NF if it is in 4NF and every join dependency in the table is implied by the candidate keys.

5NF Example: Our current structure already satisfies 5NF because there are no join dependencies that are not implied by the candidate keys.

6NF (Sixth Normal Form)

Condition: A table is in 6NF if it is in 5NF and there are no non-trivial join dependencies (focused on temporal databases, ensuring minimal redundancy).

Summary Diagram Representation

1NF: Atomicity, No Repeating Groups

└─ Students Table, Courses Table, Enrollments Table

2NF: 1NF + Full Functional Dependency

└─ Split based on partial dependencies (StudentName, CourseName)

3NF: 2NF + No Transitive Dependency

└─ Split based on transitive dependencies (InstructorPhone)

BCNF: 3NF + Every Determinant is a Candidate Key

└─ Ensure all determinants are candidate keys

4NF: BCNF + No Multi-Valued Dependency

└─ Ensure no multi-valued dependencies

5NF: 4NF + No Join Dependency

└─ Ensure join dependencies are implied by candidate keys

6NF: 5NF + No Non-Trivial Join Dependency

└─ Focus on temporal data

This structure illustrates how the table evolves through each normal form by addressing specific types of dependencies and ensuring data integrity and minimal redundancy.

Set Operations

UNION The UNION operation combines the result sets of two or more SELECT statements without duplicates.

SELECT column_name FROM table1

UNION

SELECT column_name FROM table2;

UNION ALL UNION ALL combines the result sets of two SELECT statements including duplicates.

SELECT column_name FROM table1

UNION ALL

SELECT column_name FROM table2;

INTERSECT INTERSECT returns the intersection of two SELECT statement result sets.

SELECT column_name FROM table1

INTERSECT

```
SELECT column_name FROM table2;
```

EXCEPT EXCEPT returns the difference between the first SELECT statement and the second.

```
SELECT column_name FROM table1

EXCEPT

SELECT column_name FROM table2;
```

NULL

Definition of Null A Null value represents missing or unknown data in a database.\nIt is different from zero or an empty string.

```
SELECT * FROM table WHERE column IS NULL;
```

Aggregate Functions

SUM SUM function calculates the total sum of a numeric column.

```
SELECT SUM(Salary) FROM Employees;
```

AVG AVG function returns the average value of a numeric column.

```
SELECT AVG(Salary) FROM Employees;
```

MIN MIN function returns the smallest value of the selected column.

```
SELECT MIN(Salary) FROM Employees;
```

MAX MAX function returns the largest value of the selected column.

```
SELECT MAX(Salary) FROM Employees;
```

Nested Subqueries

Nested Subqueries Nested Subqueries are SQL queries with another query embedded within the WHERE or HAVING clause.

```
SELECT * FROM Employees WHERE salary > (SELECT AVG(salary) FROM Employees);
```

Correlated Subqueries A Correlated Subquery is a subquery that references columns from the outer query.

```
SELECT e1.name FROM Employees e1 WHERE EXISTS (SELECT 1 FROM Employees e2 WHERE e1.manager_id = e2.id);
```

Subquery in SELECT Subqueries in SELECT clause are used to return a single value used in column projection.

```
SELECT name, (SELECT COUNT(*) FROM Orders WHERE employee_id = e.id) AS order_count FROM Employees e;
```

Subquery in FROM Subqueries in FROM clause create a derived table that the outer query can select from.

```
SELECT avg_salary FROM (SELECT AVG(salary) AS avg_salary FROM Employees) AS salary_info;
```

Subquery in WHERE Subqueries in WHERE clause are used to filter results based on a condition evaluated by the inner query.

```
SELECT * FROM Products WHERE price < (SELECT AVG(price) FROM Products);
```

Subquery in HAVING Subqueries in HAVING clause filter groups created by GROUP BY based on a condition in the subquery.

```
SELECT category_id, COUNT(*) FROM Products GROUP BY category_id HAVING COUNT(*) > (SELECT COUNT(*) / 10 FROM Products);
```

Subquery with IN Subqueries with IN operator return a list of values for the outer query to check membership against.

```
SELECT * FROM Customers WHERE id IN (SELECT customer_id FROM Orders WHERE total > 500);
```

Subquery with EXISTS Subqueries with EXISTS operator check for the existence of rows returned by the subquery.

```
SELECT * FROM Suppliers s WHERE EXISTS (SELECT 1 FROM Products p WHERE p.supplier_id = s.id);
```

Subquery with ANY/SOME Subqueries with ANY or SOME compare a value to each value in a list returned by the subquery.

```
SELECT * FROM Employees WHERE salary > ANY (SELECT salary FROM Employees WHERE department_id = 2);
```

Subquery with ALL Subqueries with ALL operator compare a value to all values in a list returned by the subquery.

```
SELECT * FROM Employees WHERE salary > ALL (SELECT salary FROM Employees WHERE department_id = 3);
```

Integrity Constraints

PRIMARY KEY Constraint A PRIMARY KEY constraint uniquely identifies each record in a table.

```
CREATE TABLE Students (ID int PRIMARY KEY, name varchar(255));
```

FOREIGN KEY Constraint A FOREIGN KEY constraint is a key used to link two tables together.

```
CREATE TABLE Enrollments (student_id int, FOREIGN KEY (student_id) REFERENCES Students(ID));
```

CHECK Constraint The CHECK constraint ensures that all values in a column satisfy a specific condition.

```
CREATE TABLE Students (ID int, age int, CHECK (age>=18));
```

DEFAULT Constraint The DEFAULT constraint provides a default value for a column when no value is specified.

```
CREATE TABLE Students (ID int, name varchar(255) DEFAULT 'Unknown');
```

Transactions

ACID Properties ACID properties ensure reliable processing of database transactions, maintaining data integrity.

ACID is an acronym in database systems that stands for:

- **Atomicity**: Ensures that each transaction is treated as a single unit, which either completes in its entirety or not at all.
- **Consistency**: Ensures that a transaction brings the database from one valid state to another, maintaining database invariants.
- **Isolation**: Ensures that concurrent transactions execute independently without interfering with each other.
- **Durability**: Ensures that once a transaction has been committed, it will remain so, even in the event of a system failure.

```
BEGIN TRANSACTION; // Start of a transaction block

COMMIT; // Commit changes
```

Transaction Management Transaction management ensures the control and integrity of transactions in a database system.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; // Set transaction isolation level
```

Commit Commit finalizes all changes made during the current transaction.

```
UPDATE accounts SET balance = balance - 100 WHERE id = 1;

COMMIT; // Commit the transaction
```

Rollback Rollback undoes all changes made in the current transaction.

```
DELETE FROM orders WHERE order_id = 10;

ROLLBACK; // Undo the delete operation
```

Concurrency Control Concurrency control manages simultaneous operations without conflicting data integrity.

```
SELECT * FROM users WITH (NOLOCK); // Read uncommitted data
```

Isolation Levels Isolation levels define the degree to which a transaction must be isolated from others.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; // Set isolation level
```

Deadlocks Deadlocks occur when two transactions block each other, waiting for resources.

```
SELECT * FROM table1;

SELECT * FROM table2; // Potential deadlock scenario
```

Savepoint Savepoints allow partial rollbacks within a transaction.

```
SAVEPOINT sp1; // Create a savepoint

ROLLBACK TO sp1; // Rollback to savepoint
```

Two-phase Commit Two-phase commit is a protocol to ensure all or nothing transaction commit across multiple databases.

```
PREPARE TRANSACTION 'txn_id'; // First phase
```

```
COMMIT PREPARED 'txn_id'; // Second phase
```

Distributed Transactions Distributed transactions span across multiple databases or network nodes.

```
BEGIN DISTRIBUTED TRANSACTION; // Start a distributed transaction
```

Normal Forms

Normalization Normalization is the process of organizing data to minimize redundancy and improve data integrity.

Denormalization Denormalization is the process of adding redundancy to a database to improve read performance.

Let's explain all the normal forms with an example table to illustrate the concepts. We'll start with a simple table and progressively normalize it to satisfy higher normal forms.

Example Table: Student Courses

Consider a table storing information about students and the courses they are enrolled in, along with the instructor for each course:

StudentID	**StudentName**	**CourseID**	**CourseName**	**InstructorName**	**InstructorPhone**
1	Alice	C101	Database	Dr. Smith	555-1234
2	Bob	C101	Database	Dr. Smith	555-1234
1	Alice	C102	Networks	Dr. Johnson	555-5678

1NF (First Normal Form) Condition: A table is in 1NF if all columns contain only atomic, indivisible values, and each column contains values of a single type.

1NF Example: Our table already satisfies 1NF because each cell contains a single value, and there are no repeating groups.

2NF (Second Normal Form) Condition: A table is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the primary key.

In our case, the primary key is a composite key of StudentID and CourseID.

Problems in 1NF:

- StudentName depends only on StudentID.
- CourseName, InstructorName, and InstructorPhone depend only on CourseID.

2NF Example: To achieve 2NF, we need to remove partial dependencies by splitting the table into two tables:

Students Table:

StudentID	StudentName
1	Alice
2	Bob

Courses Table:

CourseID	CourseName	InstructorName	InstructorPhone
C101	Database	Dr. Smith	555-1234
C102	Networks	Dr. Johnson	555-5678

Enrollments Table:

StudentID	CourseID
1	C101
2	C101
1	C102

3NF (Third Normal Form) Condition: A table is in 3NF if it is in 2NF and there are no transitive dependencies (a non-key attribute should not depend on another non-key attribute).

Problems in 2NF:

- InstructorPhone depends on InstructorName, not directly on CourseID.

3NF Example: To achieve 3NF, we need to remove transitive dependencies by further splitting the tables:

Instructors Table:

InstructorName	InstructorPhone

Dr. Smith	555-1234
Dr. Johnson	555-5678

Courses Table (Updated):

CourseID	CourseName	InstructorName
C101	Database	Dr. Smith
C102	Networks	Dr. Johnson

Other tables remain the same:

- Students Table
- Enrollments Table

BCNF (Boyce-Codd Normal Form) Condition: A table is in BCNF if it is in 3NF and every determinant is a candidate key. **Problems in 3NF**:

- If there were any overlapping candidate keys, we would need to ensure that every determinant (attribute determining another attribute) is a candidate key.

BCNF Example: Our tables from 3NF already satisfy BCNF, as there are no overlapping candidate keys.

4NF (Fourth Normal Form) Condition: A table is in 4NF if it is in BCNF and there are no multi-valued dependencies (an attribute should not have a one-to-many relationship with two independent attributes).

4NF Example: Our current structure already satisfies 4NF because there are no multi-valued dependencies.
5NF (Fifth Normal Form) Condition: A table is in 5NF if it is in 4NF and every join dependency in the table is implied by the candidate keys.

5NF Example: Our current structure already satisfies 5NF because there are no join dependencies that are not implied by the candidate keys.

6NF (Sixth Normal Form)

Condition: A table is in 6NF if it is in 5NF and there are no non-trivial join dependencies (focused on temporal databases, ensuring minimal redundancy).

Summary Diagram Representation

1NF: Atomicity, No Repeating Groups

└─ Students Table, Courses Table, Enrollments Table

2NF: 1NF + Full Functional Dependency

└─ Split based on partial dependencies (StudentName, CourseName)

3NF: 2NF + No Transitive Dependency

└─ Split based on transitive dependencies (InstructorPhone)

BCNF: 3NF + Every Determinant is a Candidate Key

└─ Ensure all determinants are candidate keys

4NF: BCNF + No Multi-Valued Dependency

└─ Ensure no multi-valued dependencies

5NF: 4NF + No Join Dependency

└─ Ensure join dependencies are implied by candidate keys

6NF: 5NF + No Non-Trivial Join Dependency

└─ Focus on temporal data

This structure illustrates how the table evolves through each normal form by addressing specific types of dependencies and ensuring data integrity and minimal redundancy.

Set Operations

UNION The UNION operation combines the result sets of two or more SELECT statements without duplicates.

```
SELECT column_name FROM table1

UNION

SELECT column_name FROM table2;
```

UNION ALL UNION ALL combines the result sets of two SELECT statements including duplicates.

```
SELECT column_name FROM table1

UNION ALL

SELECT column_name FROM table2;
```

INTERSECT INTERSECT returns the intersection of two SELECT statement result sets.

```
SELECT column_name FROM table1

INTERSECT
```

SELECT column_name FROM table2;

EXCEPT EXCEPT returns the difference between the first SELECT statement and the second.

SELECT column_name FROM table1

EXCEPT

SELECT column_name FROM table2;

NULL

Definition of Null A Null value represents missing or unknown data in a database.\nIt is different from zero or an empty string.

SELECT * FROM table WHERE column IS NULL;

Aggregate Functions

SUM SUM function calculates the total sum of a numeric column.

SELECT SUM(Salary) FROM Employees;

AVG AVG function returns the average value of a numeric column.

SELECT AVG(Salary) FROM Employees;

MIN MIN function returns the smallest value of the selected column.

SELECT MIN(Salary) FROM Employees;

MAX MAX function returns the largest value of the selected column.

SELECT MAX(Salary) FROM Employees;

Nested Subqueries

Nested Subqueries Nested Subqueries are SQL queries with another query embedded within the WHERE or HAVING clause.

SELECT * FROM Employees WHERE salary > (SELECT AVG(salary) FROM Employees);

Correlated Subqueries A Correlated Subquery is a subquery that references columns from the outer query.

SELECT e1.name FROM Employees e1 WHERE EXISTS (SELECT 1 FROM Employees e2 WHERE e1.manager_id = e2.id);

Subquery in SELECT Subqueries in SELECT clause are used to return a single value used in column projection.

SELECT name, (SELECT COUNT(*) FROM Orders WHERE employee_id = e.id) AS order_count FROM Employees e;

Subquery in FROM Subqueries in FROM clause create a derived table that the outer query can select from.

SELECT avg_salary FROM (SELECT AVG(salary) AS avg_salary FROM Employees) AS salary_info;

Subquery in WHERE Subqueries in WHERE clause are used to filter results based on a condition evaluated by the inner query.

```
SELECT * FROM Products WHERE price < (SELECT AVG(price) FROM Products);
```

Subquery in HAVING Subqueries in HAVING clause filter groups created by GROUP BY based on a condition in the subquery.

```
SELECT category_id, COUNT(*) FROM Products GROUP BY category_id HAVING COUNT(*) > (SELECT COUNT(*) / 10 FROM Products);
```

Subquery with IN Subqueries with IN operator return a list of values for the outer query to check membership against.

```
SELECT * FROM Customers WHERE id IN (SELECT customer_id FROM Orders WHERE total > 500);
```

Subquery with EXISTS Subqueries with EXISTS operator check for the existence of rows returned by the subquery.

```
SELECT * FROM Suppliers s WHERE EXISTS (SELECT 1 FROM Products p WHERE p.supplier_id = s.id);
```

Subquery with ANY/SOME Subqueries with ANY or SOME compare a value to each value in a list returned by the subquery.

```
SELECT * FROM Employees WHERE salary > ANY (SELECT salary FROM Employees WHERE department_id = 2);
```

Subquery with ALL Subqueries with ALL operator compare a value to all values in a list returned by the subquery.

```
SELECT * FROM Employees WHERE salary > ALL (SELECT salary FROM Employees WHERE department_id = 3);
```

Integrity Constraints

PRIMARY KEY Constraint A PRIMARY KEY constraint uniquely identifies each record in a table.

```
CREATE TABLE Students (ID int PRIMARY KEY, name varchar(255));
```

FOREIGN KEY Constraint A FOREIGN KEY constraint is a key used to link two tables together.

```
CREATE TABLE Enrollments (student_id int, FOREIGN KEY (student_id) REFERENCES Students(ID));
```

CHECK Constraint The CHECK constraint ensures that all values in a column satisfy a specific condition.

```
CREATE TABLE Students (ID int, age int, CHECK (age>=18));
```

DEFAULT Constraint The DEFAULT constraint provides a default value for a column when no value is specified.

```
CREATE TABLE Students (ID int, name varchar(255) DEFAULT 'Unknown');
```

Transactions

ACID Properties ACID properties ensure reliable processing of database transactions, maintaining data integrity.

ACID is an acronym in database systems that stands for:

- **Atomicity**: Ensures that each transaction is treated as a single unit, which either completes in its entirety or not at all.
- **Consistency**: Ensures that a transaction brings the database from one valid state to another, maintaining database invariants.
- **Isolation**: Ensures that concurrent transactions execute independently without interfering with each other.
- **Durability**: Ensures that once a transaction has been committed, it will remain so, even in the event of a system failure.

```
BEGIN TRANSACTION; // Start of a transaction block
```

```
COMMIT; // Commit changes
```

Transaction Management Transaction management ensures the control and integrity of transactions in a database system.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; // Set transaction isolation level
```

Commit Commit finalizes all changes made during the current transaction.

```
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
```

```
COMMIT; // Commit the transaction
```

Rollback Rollback undoes all changes made in the current transaction.

```
DELETE FROM orders WHERE order_id = 10;
```

```
ROLLBACK; // Undo the delete operation
```

Concurrency Control Concurrency control manages simultaneous operations without conflicting data integrity.

```
SELECT * FROM users WITH (NOLOCK); // Read uncommitted data
```

Isolation Levels Isolation levels define the degree to which a transaction must be isolated from others.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; // Set isolation level
```

Deadlocks Deadlocks occur when two transactions block each other, waiting for resources.

```
SELECT * FROM table1;
```

```
SELECT * FROM table2; // Potential deadlock scenario
```

Savepoint Savepoints allow partial rollbacks within a transaction.

```
SAVEPOINT sp1; // Create a savepoint
```

```
ROLLBACK TO sp1; // Rollback to savepoint
```

Two-phase Commit Two-phase commit is a protocol to ensure all or nothing transaction commit across multiple databases.

```
PREPARE TRANSACTION 'txn_id'; // First phase
```

```
COMMIT PREPARED 'txn_id'; // Second phase
```

Distributed Transactions Distributed transactions span across multiple databases or network nodes.

```
BEGIN DISTRIBUTED TRANSACTION; // Start a distributed transaction
```