Unit-2

SQL Queries and Constraints

SQL Queries and Constraints

Structured Query Language Fundamentals

Commands, Queries, Operators, Joins, Views

PL/SQL, Cursors, Triggers & Active Databases

Contents Overview

1. **SQL Basics**
   Commands, Queries & Operators
2. **Advanced Queries**
   Aggregates, Sets & Subqueries
3. **Keys & Constraints**
   Integrity & Key Types
4. **Joins & Views**
   Table Combinations & Virtual Tables
5. **PL/SQL & Triggers**
   Procedural SQL & Active DBs

This unit covers the fundamentals of SQL from basic query syntax to advanced procedural programming with PL/SQL, including data integrity mechanisms and active database concepts through triggers.

PART 01

SQL Commands, Query Structure & Operators

SQL Basics

Understanding the five types of SQL commands, the structure of SELECT statements, and the operators used to filter and manipulate data

** This chapter covers the foundational elements of SQL that every database developer must master*

Types of SQL Commands

Type	Full Form	Key Commands	Purpose
DDL	Data Definition Language	CREATE, ALTER, DROP, TRUNCATE	Define & modify database structure
DML	Data Manipulation Language	SELECT, INSERT, UPDATE, DELETE	Manipulate data within tables
DCL	Data Control Language	GRANT, REVOKE	Control access to data
TCL	Transaction Control Language	COMMIT, ROLLBACK, SAVEPOINT	Manage transactions
DQL	Data Query Language	SELECT	Query data from database

Key Insight: DDL changes structure, DML changes data, DCL controls access, TCL manages transactions, and DQL retrieves data.

Quick Reference

CREATE TABLE I ALTER TABLE I DROP TABLE I SELECT * FROM I INSERT INTO I UPDATE ... SET I DELETE FROM I GRANT ... TO I REVOKE ... FROM I COMMIT I ROLLBACK

Form of Basic SQL Query

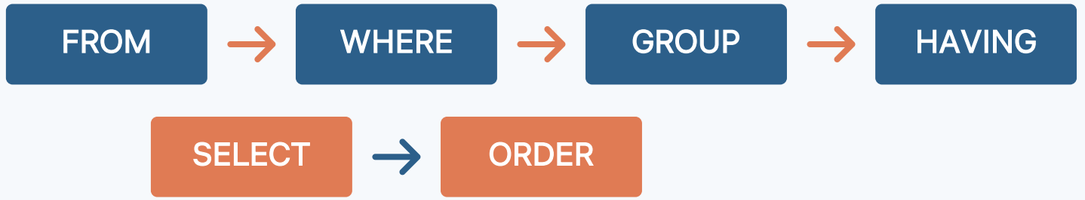
SELECT Statement Syntax

```
SELECT column_list FROM table_name WHERE condition GROUP BY column HAVING condition ORDER BY column [ASCIDESC];
```

Clause Functions

- **SELECT** — Specifies columns to retrieve
- **FROM** — Identifies table(s) to query
- **WHERE** — Filters rows by condition
- **GROUP BY** — Groups rows for aggregation
- **HAVING** — Filters groups after aggregation
- **ORDER BY** — Sorts the result set

Execution Order



Example Query

```
SELECT dept, AVG(salary) FROM employees WHERE salary > 30000 GROUP BY dept HAVING COUNT(*) > 5 ORDER BY AVG(salary) DESC;
```

Important Notes

- Only **SELECT** and **FROM** are mandatory; all other clauses are optional
- **WHERE** filters rows before grouping; **HAVING** filters groups after aggregation
- Use **DISTINCT** after SELECT to eliminate duplicate rows

SQL Operators

Arithmetic Operators

+ Addition

- Subtraction

* Multiplication

/ Division

% Modulo (remainder)

Comparison Operators

= Equal to

!=, <> Not equal to

>, < Greater / Less than

>=, <= Greater/Less or equal

Logical Operators

AND — Both conditions true

OR — At least one true

NOT — Negates condition

Combine multiple WHERE conditions

Special Operators

BETWEEN — Range selection

IN — Match list of values

LIKE — Pattern matching (% _)

IS NULL — Test for NULL

EXISTS — Test row existence

PART 02

Aggregates, Set Operators, Nested Queries & NULL Values

Advanced Query Techniques

Mastering aggregate functions, set operations, subqueries, and handling NULL values in SQL

** These techniques form the foundation for complex data analysis and reporting queries*

Aggregate Operators

COUNT()

Returns the number of rows matching criteria

COUNT(*) counts all rows; COUNT(col) excludes NULLs

SUM()

Returns the total sum of a numeric column

Ignores NULL values in calculation

AVG()

Returns the average of a numeric column

Automatically excludes NULL values

MAX()

Returns the maximum value in a column

Works with numeric, string, and date types

MIN()

Returns the minimum value in a column

Works with numeric, string, and date types

Usage: Combine with **GROUP BY** for per-group summaries. Use **HAVING** (not WHERE) to filter aggregated results.

Set Operators

Operator	Description	Result
UNION	Combines results, removes duplicates	All unique rows from both queries
UNION ALL	Combines results, keeps duplicates	All rows including duplicates
INTERSECT	Returns rows common to both queries	Only matching rows from both
MINUS / EXCEPT	Returns rows in first but not second	Difference between two result sets

Rules for Set Operations

- All SELECTs must have same number of columns
- Corresponding columns must have compatible types
- Column names come from the first SELECT
- ORDER BY can only appear at the end

Syntax Example

```
-- UNION example
SELECT emp_name FROM managers
UNION
SELECT emp_name FROM engineers;
-- INTERSECT example
SELECT city FROM customers
INTERSECT
SELECT city FROM suppliers;
```

Nested Queries (Subqueries)

A **subquery** is a query nested inside another query. The inner query executes first and its result is used by the outer query.

Types of Subqueries

- **Single-row:** Returns one row; uses =, >, <
- **Multi-row:** Returns multiple rows; uses IN, ANY, ALL
- **Correlated:** Depends on outer query; runs per row
- **Placement:** Can appear in SELECT, FROM, WHERE, HAVING

Code Examples

```
-- Single-row subquery
SELECT * FROM emp WHERE salary >
(SELECT AVG(salary) FROM emp);
-- Multi-row with IN
SELECT * FROM emp WHERE dept_id IN
(SELECT dept_id FROM dept WHERE loc='NY');
```

EXISTS Operator

Tests for existence of rows returned by subquery. Returns TRUE if subquery returns at least one row.

Example: SELECT * FROM customers c WHERE EXISTS (SELECT 1 FROM orders o WHERE o.cust_id = c.cust_id);

Performance Tip

Correlated subqueries can be slow on large datasets. Use JOINs or EXISTS instead of IN with large subquery results. Always test with realistic data.

NULL Values in SQL

NULL represents missing, unknown, or inapplicable data. It is **not** the same as zero, empty string, or false.

Key Concepts

- **IS NULL / IS NOT NULL** — Only way to test for NULL
- **Three-valued logic:** TRUE, FALSE, UNKNOWN
- **Arithmetic:** Any operation with NULL = NULL
- **Aggregation:** COUNT(*) includes NULL; COUNT(col) excludes

NULL Handling Functions

COALESCE(val1, val2, ...)

Returns first non-NULL value from list

NVL(val, replacement)

Oracle: replaces NULL with specified value

IFNULL(val, replacement)

MySQL: replaces NULL with specified value

Condition A	Condition B	A AND B	A OR B
TRUE	NULL	UNKNOWN	TRUE
FALSE	NULL	FALSE	UNKNOWN
NULL	NULL	UNKNOWN	UNKNOWN

Three-valued logic truth table for AND/OR operations with NULL

PART 03

Understanding Keys and Enforcing Data Integrity

Keys & Integrity Constraints

Exploring the types of keys in relational databases and mechanisms to ensure data accuracy and consistency

** Keys and constraints form the backbone of relational database design and data quality*

Keys in Relational Databases

Super Key

A set of attributes that can **uniquely identify** a tuple in a relation.

May contain extra (non-essential) attributes beyond what's needed for uniqueness.

Candidate Key

A **minimal super key** — no proper subset is a super key.

A relation can have multiple candidate keys. Each is a minimal set that uniquely identifies tuples.

Primary Key

The chosen candidate key. **Cannot be NULL.** Uniquely identifies each tuple.

Only one per table. Indexed for fast lookups.

Foreign Key

References the **primary key** of another relation. Establishes relationships between tables.

Can be NULL. Enforces referential integrity.

Composite Key

A key with **two or more** attributes that together uniquely identify a tuple.

Used when no single attribute is unique.

Integrity Constraints Over Relations

Constraint	Description	Example
NOT NULL	Column cannot contain NULL values	emp_name VARCHAR(50) NOT NULL
UNIQUE	All values in column must be distinct	email VARCHAR(100) UNIQUE
PRIMARY KEY	NOT NULL + UNIQUE; identifies each row	emp_id INT PRIMARY KEY
FOREIGN KEY	References PK of another table	dept_id INT REFERENCES dept(id)
CHECK	Values must satisfy a condition	salary DECIMAL CHECK (salary > 0)
DEFAULT	Assigns default value if none given	status VARCHAR(20) DEFAULT 'Active'

Types of Integrity

Entity Integrity

No PK attribute can be NULL

Referential Integrity

FK must match PK or be NULL

Domain Integrity

Values must match defined type/range

Referential Actions

ON DELETE CASCADE — auto-deletes child rows | **ON UPDATE CASCADE** — updates FK | **SET NULL** — sets FK to NULL | **RESTRICT** — rejects operation

PART 04

Combining Data from Multiple Tables and Virtual Table Concepts

Joins & Views

Learning to combine data across tables using joins and create virtual tables with views for simplified querying

** Joins and views are essential skills for real-world database development and reporting*

Join Type	Description	Result
INNER JOIN	Returns only matching rows from both tables	Intersection of both tables
LEFT JOIN	All rows from left + matching from right	Left table fully preserved
RIGHT JOIN	All rows from right + matching from left	Right table fully preserved
FULL JOIN	All rows when match in either table	Union of both tables
CROSS JOIN	Cartesian product of both tables	All possible combinations
SELF JOIN	Joins a table with itself using aliases	Compare rows within same table
NATURAL JOIN	Auto-joins on same-named columns	Implicit column matching

Syntax Examples

```
-- INNER JOIN
SELECT e.name, d.dept_name
FROM employees e
INNER JOIN departments d
ON e.dept_id = d.dept_id;
```

```
-- LEFT JOIN
SELECT c.name, o.order_id
FROM customers c
LEFT JOIN orders o
ON c.cust_id = o.cust_id;
```

Introduction to Views

A **view** is a virtual table based on a SELECT query. It does not store data physically — only the query definition is stored.

Advantages of Views

- **Simplicity:** Hides complex query logic
- **Security:** Restricts access to sensitive columns
- **Data Independence:** Shield apps from schema changes
- **Consistency:** Same query logic across applications

Creating a View

```
CREATE VIEW emp_dept_view AS
SELECT e.name, d.dept_name
FROM employees e
JOIN departments d
ON e.dept_id = d.dept_id;
```

Types of Views

Simple/Updatable Views

Support INSERT, UPDATE, DELETE on base table

Complex/Read-only Views

Contain joins, aggregates, DISTINCT — not updatable

Materialized Views

Store results physically; require periodic refresh

WITH CHECK OPTION

Prevents modifications through the view that would cause rows to disappear from the view. Ensures data modified through a view still satisfies the view's WHERE condition.

PART 05

Procedural SQL Programming and Active Database Concepts

PL/SQL, Cursors & Triggers

Extending SQL with procedural constructs, row-by-row processing, and automated database event handling

** These advanced features transform databases from passive storage into active, rule-enforcing systems*

Introduction to PL/SQL

PL/SQL (Procedural Language/SQL) extends SQL with procedural programming: variables, loops, conditions, and exception handling.

Block Structure

```
DECLARE
v_salary NUMBER;
v_name VARCHAR(50);
BEGIN
SELECT name, salary INTO v_name, v_salary
FROM emp WHERE emp_id = 101;
DBMS_OUTPUT.PUT_LINE(v_name);
EXCEPTION
WHEN NO_DATA_FOUND THEN ...
END;
```

Key Features

- **Variables & Constants** — Declared in DECLARE with data types
- **Control Structures** — IF-THEN-ELSE, CASE, LOOP, WHILE, FOR
- **Exception Handling** — Predefined and user-defined exceptions
- **Stored Procedures** — Named blocks stored for reuse
- **Functions** — Return a single value; used in SQL expressions

Benefits: Improved performance (reduced network traffic), modular code, tight integration with Oracle database, enhanced security through encapsulation.

Control Structure Example

```
IF v_salary > 50000 THEN grade := 'A'; ELSIF v_salary > 30000 THEN grade := 'B'; ELSE grade := 'C'; END IF;
FOR rec IN (SELECT * FROM emp) LOOP DBMS_OUTPUT.PUT_LINE(rec.name); END LOOP;
```

Cursors in PL/SQL

A **cursor** is a pointer to the memory area (context area) that holds query results. It enables row-by-row processing of query results.

Implicit Cursors

Auto-created for DML statements (INSERT, UPDATE, DELETE).

SQL%ROWCOUNT — rows affected

SQL%FOUND — TRUE if rows affected

SQL%NOTFOUND — TRUE if no rows

Explicit Cursors

Programmer-controlled for multi-row SELECT queries.

DECLARE — define with SELECT

OPEN — execute, populate result

FETCH — retrieve one row

CLOSE — release resources

Cursor Attributes

Attribute	Description
%ISOPEN	TRUE if cursor is open
%FOUND	TRUE if last fetch returned a row
%NOTFOUND / %ROWCOUNT	No row fetched / Rows fetched so far

Cursor FOR Loop

```
FOR emp_rec IN c_employees LOOP
DBMS_OUTPUT.PUT_LINE(
emp_rec.name II ' - ' II
emp_rec.salary);
END LOOP; -- auto OPEN, FETCH, CLOSE
```

Advanced Cursor Types

Parameterized Cursors

Accept parameters to make queries dynamic at runtime

REF CURSOR

Cursor variable associated with different queries at runtime

Triggers & Active Databases

A trigger is a stored program that automatically executes in response to specific database events (INSERT, UPDATE, DELETE).

Trigger Components

Event → **Condition** → **Action**

- **Event**: INSERT / UPDATE / DELETE
- **Condition**: WHEN clause (optional)
- **Action**: PL/SQL block to execute

Timing & Granularity

Timing	Description
BEFORE	Trigger fires before the event executes
AFTER	Trigger fires after the event completes
INSTEAD OF	Replaces the event (used with views)

Row-level: Fires once per affected row (**FOR EACH ROW**)

Statement-level: Fires once per SQL statement

Trigger Types & Uses

- **DML Triggers** — On table INSERT/UPDATE/DELETE
- **DDL Triggers** — On CREATE, ALTER, DROP
- **System Triggers** — On database logon/logoff

Common Applications

- Audit logging (track data changes)
- Automatic calculation of derived fields
- Enforce complex business rules
- Data validation and integrity checks

Basic Trigger Syntax

```
CREATE TRIGGER trg_name BEFORE INSERT ON table_name FOR EACH ROW BEGIN ... END;
```

Thank You

UNIT II: SQL Queries and Constraints

SQL Commands | Queries | Operators | Aggregates | Joins | Views | PL/SQL | Triggers

Questions & Discussion

Master SQL fundamentals to build robust, efficient database applications