1. SQL Aggregate Functions with Examples

AIM

To study and demonstrate the various SQL Aggregate Functions such as COUNT, SUM, AVG, MAX, and MIN with practical examples on a relational database table.

DESCRIPTION

SQL Aggregate Functions are built-in functions that perform a calculation on a set of values and return a single summary value. They are commonly used with the GROUP BY clause to group result sets by one or more columns. These functions are fundamental to data analysis, reporting, and business intelligence queries.

The main SQL aggregate functions are:

1. COUNT() – Returns the number of rows that match a specified condition.
2. SUM() – Returns the total sum of a numeric column.
3. AVG() – Returns the average value of a numeric column.
4. MAX() – Returns the largest value in a selected column.
5. MIN() – Returns the smallest value in a selected column.

Real-world relevance: Aggregate functions are used in banking (total balance per customer), e-commerce (total sales per product), HR (average salary per department), and academic systems (student grade analysis).

ALGORITHM

1. Start the SQL environment and connect to the database.
2. Create a table named EMPLOYEES with columns: EMP_ID, EMP_NAME, DEPARTMENT, SALARY.
3. Insert sample records into the EMPLOYEES table.
4. Use COUNT(*) to count the total number of employees.
5. Use SUM(SALARY) to compute the total salary expenditure.
6. Use AVG(SALARY) to calculate the average salary across all employees.
7. Use MAX(SALARY) to identify the highest salary.
8. Use MIN(SALARY) to identify the lowest salary.
9. Use GROUP BY clause with aggregate functions to get department-wise statistics.
10. Display and verify the results.

CONCLUSION

SQL Aggregate Functions (COUNT, SUM, AVG, MAX, MIN) were successfully demonstrated. These functions are essential for summarizing and analyzing data in relational databases. When combined with GROUP BY and ORDER BY clauses, they provide powerful departmental and categorical statistics that form the backbone of database reporting.

2. PL/SQL Program for Electricity Bill Generation using Cursor

AIM

To develop a PL/SQL program that generates electricity bills for all customers using an Explicit Cursor and applies multiple slab-based billing conditions to compute the final bill amount.

DESCRIPTION

Electricity billing systems use tiered or slab-based pricing, where the cost per unit increases as consumption increases. A PL/SQL cursor is used to process each customer record one by one from the database. For each customer, the consumed units are read and the bill amount is computed based on the applicable slab rates.

Slab Rate Structure Used:

Units 1–100 : Rs. 1.50 per unit (Basic slab)

Units 101–200 : Rs. 2.50 per unit (Medium slab)

Units 201–300 : Rs. 4.00 per unit (High slab)

Units > 300 : Rs. 6.00 per unit (Premium slab)

A fixed service charge of Rs. 50 is added to every bill. An Explicit Cursor fetches all consumer records and the IF-ELSIF-ELSE construct applies the slab logic per consumer.

ALGORITHM

1. Create the ELECTRICITY table with customer details and unit consumption.
2. Insert sample customer records into the table.
3. Declare variables: v_cid, v_name, v_units, v_bill in the DECLARE section.
4. Declare an EXPLICIT CURSOR that selects all rows from ELECTRICITY table.
5. OPEN the cursor to start fetching.
6. Enter a LOOP and FETCH the next record into declared variables.
7. EXIT WHEN the cursor has no more rows (%NOTFOUND).
8. Apply IF-ELSIF conditions to compute bill based on consumed units.
9. Add fixed service charge of Rs. 50 to the computed bill.
10. Update the ELECTRICITY table with the computed bill amount.
11. Display the customer ID, name, units, and total bill using DBMS_OUTPUT.
12. CLOSE the cursor and COMMIT the transaction.

CONCLUSION

A PL/SQL program for electricity bill generation was successfully implemented using an explicit cursor and multiple IF-ELSIF-ELSE conditions. The program correctly applied slab-based billing rates for each customer, added the fixed service charge, updated the database, and displayed a formatted bill report. This demonstrates the practical use of PL/SQL cursors in real-world billing systems.

3. PL/SQL Program to Generate ICET Rank Card Report with Grade Assignment

AIM

To develop a PL/SQL program using a Cursor and IF-ELSE conditions to generate an ICET Rank Card Report that fetches student details and assigns grades based on their scores.

DESCRIPTION

ICET (Integrated Common Entrance Test) is a state-level entrance examination for admission into MBA and MCA programs. The rank card report displays the candidate's hall ticket number, name, subject-wise marks, total score, and assigned grade.

Grade Criteria Used:

Score >= 90 : Grade O (Outstanding)

Score >= 75 : Grade A+ (Excellent)

Score >= 60 : Grade A (Very Good)

Score >= 50 : Grade B (Good)

Score >= 40 : Grade C (Average)

Score < 40 : Grade F (Fail)

An explicit cursor iterates through all student records, computes the total score, assigns a grade via IF-ELSIF-ELSE logic, and prints the rank card.

ALGORITHM

1. Create the ICET_STUDENTS table with hall ticket number, name, and subject scores.

2. Insert sample student records with marks for three subjects.

3. Declare cursor c_icet to select all students and their scores.

4. Declare variables for each field and computed values (total, grade).

5. OPEN the cursor and begin the LOOP.

6. FETCH each student record into the declared variables.

7. EXIT WHEN cursor %NOTFOUND.

8. Compute total = maths + reasoning + communication.

9. Apply IF-ELSIF-ELSE to assign grade based on total score.

10. Print the formatted rank card entry for each student.

11. CLOSE the cursor after all records are processed.

CONCLUSION

The PL/SQL program for ICET Rank Card generation was successfully implemented. The program used an explicit cursor to iterate through all student records, computed the total score, and applied multi-level IF-ELSIF-ELSE conditions to assign appropriate grades. The formatted output simulates a real-world rank card report suitable for academic systems.

4. Database Trigger to Track Bank Transactions

AIM

To implement a database trigger in Oracle PL/SQL that automatically records every deposit and withdrawal transaction into an audit/log table whenever the account balance is updated.

DESCRIPTION

A database trigger is a stored PL/SQL block that is automatically executed (fired) in response to specified DML events (INSERT, UPDATE, DELETE) on a table. Triggers are used for auditing, enforcing business rules, maintaining referential integrity, and generating derived data.

In a banking system, it is critical to maintain a complete audit trail of all balance changes. An AFTER UPDATE trigger on the BANK_ACCOUNTS table captures every balance modification and logs it into the TRANSACTION_LOG table with transaction type, amount, and timestamp.

Types of Triggers: BEFORE/AFTER, ROW-LEVEL (:NEW, :OLD), STATEMENT-LEVEL. This program uses an AFTER UPDATE, FOR EACH ROW trigger.

ALGORITHM

1. Create the BANK_ACCOUNTS table with account number, holder name, and balance.

2. Create the TRANSACTION_LOG table with log ID, account number, transaction type, amount, and timestamp.

3. Create a sequence TXN_SEQ to generate unique log IDs.

4. Insert sample account records.

5. Create an AFTER UPDATE FOR EACH ROW trigger on BANK_ACCOUNTS.

6. Inside the trigger, compare :NEW.BALANCE with :OLD.BALANCE.

7. If :NEW.BALANCE > :OLD.BALANCE, record 'DEPOSIT' with difference amount.

8. If :NEW.BALANCE < :OLD.BALANCE, record 'WITHDRAWAL' with difference amount.

9. Insert the transaction details into TRANSACTION_LOG.

10. Perform test UPDATEs on BANK_ACCOUNTS to fire the trigger.

11. Query TRANSACTION_LOG to verify trigger fired correctly.

CONCLUSION

A database trigger was successfully created and tested to track all bank account balance changes. The AFTER UPDATE FOR EACH ROW trigger automatically classified each change as DEPOSIT or WITHDRAWAL and inserted a timestamped record into the TRANSACTION_LOG table. This demonstrates how Oracle triggers provide automatic audit trails in financial systems without requiring explicit application-level logging.

5. Exception Handling in PL/SQL with Examples

AIM

To understand and demonstrate Exception Handling in PL/SQL using predefined exceptions, user-defined exceptions, and the OTHERS exception handler with practical examples.

DESCRIPTION

Exception handling is a critical feature of PL/SQL that allows programmers to gracefully manage runtime errors without abruptly terminating the program. An exception is an error or warning condition that disrupts the normal execution flow of a PL/SQL block.

PL/SQL Exception Categories:

1. Predefined Exceptions – Built-in Oracle exceptions (e.g., NO_DATA_FOUND, TOO_MANY_ROWS, ZERO_DIVIDE, VALUE_ERROR).

2. Non-predefined Exceptions – Oracle errors without names; declared using PRAGMA EXCEPTION_INIT.

3. User-defined Exceptions – Custom exceptions declared and raised by the programmer using RAISE.

Structure of Exception Handling Block:

DECLARE ... BEGIN ... EXCEPTION WHEN <exception> THEN ... END;

ALGORITHM

1. Create a demo table STUDENTS with ID, name, and marks.
2. Insert a few sample records.
3. Write Example 1: Handle NO_DATA_FOUND when SELECT returns no row.
4. Write Example 2: Handle TOO_MANY_ROWS when SELECT INTO fetches multiple rows.
5. Write Example 3: Handle ZERO_DIVIDE exception.
6. Write Example 4: Declare and raise a user-defined exception.
7. Write Example 5: Use WHEN OTHERS to catch unexpected exceptions.
8. Use SQLCODE and SQLERRM in OTHERS handler for error diagnostics.
9. Execute each block and observe the exception messages.

CONCLUSION

Exception handling in PL/SQL was successfully demonstrated using five practical examples covering predefined exceptions (NO_DATA_FOUND, TOO_MANY_ROWS, ZERO_DIVIDE), user-defined exceptions, and the WHEN OTHERS generic handler. Proper exception handling ensures robust, fault-tolerant PL/SQL programs by gracefully managing runtime errors and providing meaningful diagnostic messages.

6. SQL Commands using Master-Child Tables

AIM

To demonstrate SQL commands (DDL and DML) using Master-Child (Parent-Child) table relationships, including creation of Foreign Key constraints, Cascading operations, and JOIN queries.

DESCRIPTION

In relational databases, a Master-Child relationship exists when one table (Master/Parent) contains primary key data that is referenced by another table (Child) via a Foreign Key. This enforces referential integrity, ensuring that child records always have a corresponding parent record.

Example: A DEPARTMENT table (Master) and an EMPLOYEE table (Child). Each employee belongs to a department; the department must exist before an employee can be assigned to it.

Key SQL Concepts Demonstrated: PRIMARY KEY, FOREIGN KEY, ON DELETE CASCADE, INNER JOIN, LEFT JOIN, Cascading deletes, DML operations on related tables.

ALGORITHM

1. Create the DEPARTMENTS master table with DEPT_ID as Primary Key.
2. Create the EMPLOYEES child table with DEPT_ID as Foreign Key referencing DEPARTMENTS.
3. Insert records into DEPARTMENTS first (master first rule).
4. Insert records into EMPLOYEES referencing valid DEPT_ID values.
5. Demonstrate INSERT violation: try inserting an employee with invalid DEPT_ID.
6. Demonstrate DELETE restriction: try deleting a department that has employees.
7. Perform INNER JOIN to display employees with department names.
8. Perform LEFT JOIN to show departments even without employees.
9. Apply ON DELETE CASCADE: recreate with cascade and verify.
10. Display final results of all queries.

CONCLUSION

Master-Child table relationships were successfully demonstrated using DEPARTMENTS (master) and EMP_DETAIL (child) tables. SQL Foreign Key constraints enforced referential integrity. INNER JOIN and LEFT JOIN queries retrieved combined data across related tables. ON DELETE CASCADE automatically removed child records when the parent was deleted, illustrating real-world cascading referential integrity.

7. PL/SQL Program to Find Factorial of a Number

AIM

To write a PL/SQL program that accepts a number from the user and computes its factorial using a loop construct, demonstrating basic PL/SQL programming with variables, loops, and output.

DESCRIPTION

The factorial of a non-negative integer n (denoted as n!) is the product of all positive integers from 1 to n. By definition, 0! = 1.

Formula: n! = n × (n-1) × (n-2) × ... × 2 × 1

Examples: 5! = 5 × 4 × 3 × 2 × 1 = 120 | 0! = 1 | 1! = 1

In PL/SQL, factorial is computed using a FOR loop or WHILE loop. The result variable is initialized to 1 and multiplied iteratively. The program also handles the special case of 0 and validates that negative numbers do not have a factorial.

ALGORITHM

1. Start the PL/SQL block.

2. Declare variables: v_n (input number), v_fact (result, initialized to 1), v_i (loop counter).

3. Assign the input value to v_n (e.g., 6).

4. Check if v_n is negative. If so, print error and exit.

5. If v_n = 0, factorial is 1 (special case).

6. Use a FOR loop from 1 to v_n.

7. Inside the loop, multiply v_fact by the loop counter: v_fact := v_fact * i.

8. After the loop ends, print: 'Factorial of v_n is v_fact'.

9. End the PL/SQL block.

CONCLUSION

The PL/SQL program to compute the factorial of a number was successfully implemented using a FOR loop. The program handles all edge cases including 0! = 1 and negative number validation. Factorial computation is a fundamental mathematical operation used in combinatorics, permutations, probability, and algorithm analysis.

8. PL/SQL Program to Check Whether a Number is Palindrome

AIM

To write a PL/SQL program that checks whether a given number is a palindrome — i.e., reads the same forwards and backwards.

DESCRIPTION

A palindrome number is a number that remains the same when its digits are reversed. For example: 121, 1331, 12321 are palindromes; while 123, 456 are not.

Logic: Extract the last digit of the number using MOD(n, 10). Build the reversed number by multiplying the current reversed value by 10 and adding the digit. Divide the original by 10. Repeat until n = 0. Compare the reversed number with the original.

Additional: The program also uses the built-in TO_CHAR and REVERSE functions as an alternative string-based approach for verification.

ALGORITHM

1. Declare variables: v_num (original), v_temp (copy for processing), v_rev (reversed number), v_digit (current digit).

2. Initialize v_rev to 0 and v_temp to v_num.

3. Enter a WHILE loop: continue while v_temp > 0.

4. Inside the loop: extract last digit with MOD(v_temp, 10).

5. Build reversed number: v_rev := v_rev * 10 + v_digit.

6. Remove last digit: v_temp := TRUNC(v_temp / 10).

7. Exit loop when v_temp = 0.

8. Compare v_rev with v_num.

9. If equal, print 'Palindrome'; otherwise print 'Not a Palindrome'.

CONCLUSION

The PL/SQL program to check palindrome numbers was successfully implemented using a WHILE loop and digit extraction with MOD and TRUNC functions. The program correctly identified 12321 as a palindrome and 12345 as non-palindrome. Palindrome checking has applications in data validation, encryption algorithms, and string processing utilities.

9. Demonstration of Predefined Exceptions in PL/SQL

AIM

To demonstrate the various Predefined (Built-in) Exceptions available in Oracle PL/SQL by writing separate examples for each commonly used exception type.

DESCRIPTION

Oracle PL/SQL has a set of predefined exceptions that are automatically raised by the Oracle engine when specific runtime errors occur. These exceptions are pre-named and do not need to be declared; they can be directly referenced in the EXCEPTION section.

Common Predefined Exceptions:

NO_DATA_FOUND – SELECT INTO returns no rows (ORA-01403)
TOO_MANY_ROWS – SELECT INTO returns more than one row (ORA-01422)
ZERO_DIVIDE – Division by zero attempted (ORA-01476)
VALUE_ERROR – Arithmetic, conversion, or truncation error (ORA-06502)
INVALID_CURSOR – Illegal cursor operation (ORA-01001)
CURSOR_ALREADY_OPEN– Attempt to open an already open cursor (ORA-06511)
DUP_VAL_ON_INDEX – Duplicate value in unique index (ORA-00001)
LOGIN_DENIED – Invalid username/password (ORA-01017)

ALGORITHM

1. Create a test table SAMPLE_DATA with a unique constraint.
2. Insert test records.
3. Example 1: Trigger NO_DATA_FOUND with a SELECT for a missing row.
4. Example 2: Trigger TOO_MANY_ROWS with SELECT INTO returning multiple rows.
5. Example 3: Trigger ZERO_DIVIDE with an integer division by zero.
6. Example 4: Trigger DUP_VAL_ON_INDEX with a duplicate INSERT.
7. Example 5: Trigger VALUE_ERROR by assigning a large value to a small variable.
8. In each example, catch the exception and display a descriptive message.

CONCLUSION

All five commonly used predefined PL/SQL exceptions were successfully demonstrated with clear, executable examples. These built-in exceptions (NO_DATA_FOUND, TOO_MANY_ROWS, ZERO_DIVIDE, DUP_VAL_ON_INDEX, VALUE_ERROR) allow PL/SQL programs to handle common runtime errors gracefully, preventing abnormal termination and providing meaningful error feedback to the user.

10. PL/SQL Program using User-Defined Exception for Divide Operation

AIM

To write a PL/SQL program that performs a division operation and uses a user-defined exception to handle division by zero along with additional business-logic validations.

DESCRIPTION

User-defined exceptions allow PL/SQL programmers to define and raise their own application-specific error conditions. Unlike predefined exceptions that are raised automatically by Oracle, user-defined exceptions must be: (1) declared in the DECLARE section, (2) explicitly raised using the RAISE statement, and (3) handled in the EXCEPTION section.

This program demonstrates three user-defined exceptions for a division operation:

e_divide_by_zero – Raised when divisor is 0.

e_negative_input – Raised when any operand is negative.

e_overflow – Raised when result exceeds a defined limit.

ALGORITHM

1. Start the DECLARE section.

2. Declare v_numerator and v_denominator as NUMBER input variables.

3. Declare v_result as NUMBER to store the output.

4. Declare three user-defined exceptions: e_divide_by_zero, e_negative_input, e_overflow.

5. In the BEGIN section, check if the denominator is 0; if yes, RAISE e_divide_by_zero.

6. Check if either operand is negative; if yes, RAISE e_negative_input.

7. Perform the division: v_result := v_numerator / v_denominator.

8. Check if result exceeds limit (e.g., 10000); if yes, RAISE e_overflow.

9. Print the result.

10. In the EXCEPTION section, handle each user-defined exception with WHEN ... THEN.

CONCLUSION

A PL/SQL program using user-defined exceptions for a division operation was successfully implemented. The program declared three custom exceptions (e_divide_by_zero, e_negative_input, e_overflow), raised them explicitly using RAISE when corresponding conditions were met, and handled them in the EXCEPTION section with meaningful messages. User-defined exceptions are essential for implementing application-specific business rule validations in PL/SQL.

11. PL/SQL Program for Library Book Issue System with Exception Handling

AIM

To develop a PL/SQL program simulating a Library Book Issue System that uses cursors to process book issue requests and incorporates comprehensive exception handling for cases like book unavailability, member not found, and already issued books.

DESCRIPTION

A Library Management System is a classic real-world application of database programming. The book issue process involves: verifying that the member exists, checking that the book is available, recording the issue transaction, and updating the book's availability status.

This program uses: EXPLICIT CURSOR to process pending issue requests, predefined exceptions (NO_DATA_FOUND), and user-defined exceptions (e_book_not_available, e_already_issued, e_max_limit).

ALGORITHM

1. Create LIBRARY_MEMBERS table (member ID, name, issued count).
2. Create LIBRARY_BOOKS table (book ID, title, author, available copies).
3. Create BOOK_ISSUES table (issue ID, member ID, book ID, issue date, return date).
4. Insert sample members and books.
5. Create a PL/SQL block with a cursor on pending issue requests.
6. For each issue request, use SELECT INTO to fetch member and book details.
7. Handle NO_DATA_FOUND for non-existent member or book.
8. Check available copies: if 0, raise e_book_not_available.
9. Check if member already has the book issued: raise e_already_issued.
10. Check member issue limit (max 3): raise e_max_limit if exceeded.
11. If all checks pass: INSERT into BOOK_ISSUES and UPDATE book stock.
12. Print success or error message for each transaction.

CONCLUSION

A comprehensive Library Book Issue System was successfully implemented in PL/SQL. The program used an explicit cursor to process multiple issue requests, combined predefined (NO_DATA_FOUND) and user-defined exceptions (e_book_not_available, e_max_limit) for robust error handling, and performed transactional updates including book stock and member issue count. This demonstrates real-world application of PL/SQL in library management systems.

12. Practical Usage of Predefined Oracle Packages in PL/SQL

AIM

To demonstrate the practical usage of commonly used predefined Oracle PL/SQL packages: DBMS_OUTPUT, DBMS_DATE, UTL_FILE, and DBMS_RANDOM with examples.

DESCRIPTION

Oracle provides a rich library of built-in (predefined) packages that extend PL/SQL's functionality. These packages contain procedures, functions, and variables that can be called from any PL/SQL block.

Key Predefined Packages Covered:

DBMS_OUTPUT – Displays output from PL/SQL blocks to screen.

DBMS_RANDOM – Generates random numbers and strings.

DBMS_UTILITY – Provides miscellaneous utility procedures.

UTL_FILE – Enables file I/O operations (read/write text files).

TO_CHAR/TO_DATE – Date/number formatting functions (built-in).

ALGORITHM

1. Demonstrate DBMS_OUTPUT package: PUT, PUT_LINE, NEW_LINE, ENABLE.

2. Demonstrate DBMS_RANDOM: VALUE (float), VALUE(low,high) (integer range), STRING.

3. Demonstrate date functions with TO_CHAR, SYSDATE, ADD_MONTHS, MONTHS_BETWEEN.

4. Demonstrate DBMS_UTILITY.FORMAT_CALL_STACK to view execution context.

5. Show DBMS_UTILITY.GET_TIME for performance measurement.

6. Execute each package example and display results.

CONCLUSION

Predefined Oracle PL/SQL packages were successfully demonstrated. DBMS_OUTPUT provides screen I/O capabilities; DBMS_RANDOM generates random values for testing and simulations; Date built-in functions support comprehensive date arithmetic; DBMS_UTILITY offers performance measurement and diagnostic utilities. These packages significantly extend PL/SQL capabilities beyond basic SQL functionality.

13. Create and Manage Object Types, Tables, and Methods using Oracle SQL

AIM

To create and manage Oracle Object Types (user-defined types), Object Tables, and Member Methods in Oracle SQL to demonstrate the Object-Relational features of Oracle Database.

DESCRIPTION

Oracle supports Object-Relational programming through user-defined data types called Object Types. An Object Type is a schema object that encapsulates a data structure (attributes) and behavior (methods/functions) into a single unit, similar to classes in object-oriented programming.

Key Concepts:

Object Type – Template defining attributes and methods (similar to a class).

Object Table – A table where each row is an instance (object) of an Object Type.

Member Methods– Functions/procedures defined inside an Object Type body.

Constructors – Automatically generated by Oracle to create object instances.

SELF – Keyword inside methods referring to the current object instance.

ALGORITHM

1. Create an Object Type STUDENT_TYPE with attributes (roll_no, name, marks).

2. Define a member function GET_GRADE in the type specification.

3. Create the Object Type Body to implement the GET_GRADE function logic.

4. Create an Object Table STUDENT_OBJ_TABLE of type STUDENT_TYPE.

5. Insert student object instances using the constructor STUDENT_TYPE().

6. Query the object table to retrieve all attribute values.

7. Call the member method GET_GRADE using dot notation.

8. Create a second type ADDRESS_TYPE for demonstrating type composition.

9. Create an EMPLOYEE_OBJ type containing an ADDRESS_TYPE attribute (nested type).

10. Insert and query nested object instances.

CONCLUSION

Oracle Object Types were successfully created, implemented, and queried. The STUDENT_TYPE object type encapsulated attributes (ROLL_NO, STU_NAME, MARKS) and methods (GET_GRADE, GET_DETAILS) in a single unit. Object Tables stored typed object instances. Nested Object Types (ADDRESS_TYPE inside EMP_OBJ) demonstrated type composition. This experiment illustrates Oracle's Object-Relational Model capabilities for modelling complex real-world entities.

14. Selection, Projection, and Sorting Queries on a Simple Table (Case Study-1)

AIM

To perform and demonstrate fundamental SQL operations — Selection (filtering rows), Projection (choosing columns), and Sorting (ordering results) — on a simple relational table representing a Student database.

DESCRIPTION

Three fundamental relational algebra operations are directly mapped to SQL:

1. SELECTION – Retrieves specific rows that satisfy a condition (implemented using WHERE clause). Reduces the number of rows in the result.

2. PROJECTION – Retrieves specific columns from a table (implemented by specifying column names in SELECT). Reduces the number of columns in the result.

3. SORTING – Arranges result rows in ascending or descending order (implemented using ORDER BY clause).

These operations are the building blocks of all SQL queries and form the foundation of relational algebra applied to database management.

Case Study: A STUDENT table is created with Roll No, Name, Branch, City, and CGPA. Various combinations of Selection, Projection, and Sorting are performed.

ALGORITHM

1. Create the STUDENT table with appropriate columns and data types.
2. Insert at least 10 student records covering different branches and cities.
3. Perform PROJECTION: select only specific columns (Roll No, Name, CGPA).
4. Perform SELECTION: select rows where BRANCH = 'CSE' (filter by condition).
5. Combine Selection and Projection: select Name and CGPA where CGPA > 8.0.
6. Perform SORTING: ORDER BY CGPA ASC and CGPA DESC.
7. Combine all three: project specific columns, apply WHERE filter, and sort.
8. Demonstrate DISTINCT projection to eliminate duplicate values.
9. Use BETWEEN, IN, LIKE, IS NULL in selection conditions.
10. Display and verify all results.

CONCLUSION

The three fundamental SQL operations — Selection (WHERE), Projection (SELECT column list), and Sorting (ORDER BY) — were successfully demonstrated on the STUDENT table. These operations correspond directly to Relational Algebra operations and form the core building blocks of all SQL queries. Additional clauses like DISTINCT, BETWEEN, IN, and LIKE extended the querying capability.

15. ER Diagram for a Company Database (Case Study-2)

AIM

To design an Entity-Relationship (ER) Diagram for a Company Database representing Employees, Departments, and Projects, and to implement the schema in SQL with appropriate constraints.

DESCRIPTION

An Entity-Relationship (ER) Diagram is a graphical representation of the logical structure of a database. It is used in the database design phase to model real-world entities, their attributes, and the relationships between them before implementation.

ER Diagram Symbols:

Rectangle – Entity (e.g., EMPLOYEE, DEPARTMENT)

Ellipse – Attribute (e.g., EMP_ID, EMP_NAME)

Double Ellipse – Multivalued Attribute (e.g., PHONE_NUMBERS)

Dashed Ellipse – Derived Attribute (e.g., AGE derived from DOB)

Diamond – Relationship (e.g., WORKS_IN, MANAGES)

Double Rectangle– Weak Entity

Line – Connecting entities and relationships

Company Database ER Entities:

EMPLOYEE – EMP_ID (PK), EMP_NAME, DOB, SALARY, DEPT_NO (FK)

DEPARTMENT – DEPT_NO (PK), DEPT_NAME, LOCATION, MGR_EMP_ID (FK)

PROJECT – PROJ_ID (PK), PROJ_NAME, BUDGET, DEPT_NO (FK)

WORKS_ON – EMP_ID + PROJ_ID (Composite PK), HOURS (relationship attribute)

DEPENDENT – Weak entity with EMP_ID (partial key), DEP_NAME, RELATION

ER Relationships:

EMPLOYEE WORKS_IN DEPARTMENT (M:1 – Many employees in one department)

DEPARTMENT MANAGES (has) EMPLOYEE (1:1 – One manager per department)

EMPLOYEE WORKS_ON PROJECT (M:N – Many employees on many projects)

EMPLOYEE HAS DEPENDENT (1:N – One employee has many dependents)

DEPARTMENT CONTROLS PROJECT (1:N – One department controls many projects)

ALGORITHM

1. Identify entities: EMPLOYEE, DEPARTMENT, PROJECT, DEPENDENT.

2. Identify attributes for each entity and mark primary keys.

3. Identify relationships: WORKS_IN, MANAGES, WORKS_ON, CONTROLS, HAS.

4. Determine cardinality (1:1, 1:N, M:N) for each relationship.

5. Identify weak entities (DEPENDENT) and their identifying relationship.

6. Translate ER Diagram to relational schema (table definitions).

7. Create SQL tables with appropriate Primary Keys and Foreign Keys.

8. Create WORKS_ON junction table to resolve M:N relationship.

9. Insert sample data and verify with JOIN queries.

10. Write a query to display each employee's department and projects.

CONCLUSION

An Entity-Relationship Diagram for a Company Database was designed and successfully translated into a relational SQL schema. The ER model captured entities (EMPLOYEE, DEPARTMENT, PROJECT, DEPENDENT), their attributes, and five different relationships with appropriate cardinalities. The M:N relationship between EMPLOYEE and PROJECT was resolved using the WORKS_ON junction table. All constraints including Primary Key, Foreign Key, and NOT NULL were implemented, demonstrating the complete ER-to-Relational mapping process.