Batch 1 — Detailed Line-by-Line Explanations

Prepared for: Abhinav Kakatkar
Contents: 10 practical codes (MySQL, Oracle PL/SQL, MongoDB). Layout: Code first, then detailed explanations.
Generated: 10-Nov-2025

1) Customer & Account — Joins and Subqueries (chit 1.txt)

Purpose:
Create Customer and Account tables, insert records, demonstrate INNER, LEFT, RIGHT joins, NATURAL JOIN, and subqueries (AVG, MAX, MIN).

Code:
```
SHOW DATABASES;
CREATE DATABASE DBMS01;
USE DBMS01;
CREATE TABLE Customer(C_Id INT PRIMARY KEY, Cname VARCHAR(50) NOT NULL, City VARCHAR(50));
INSERT INTO Customer(C_Id, Cname, City) VALUES
(1,'John','Nashik'),(2,'Seema','Aurangabad'),(3,'Amita','Nagar'),(4,'Rakesh','Pune'),
(5,'Samata','Nashik'),(6,'Ankita','Chandwad'),(7,'Bhavika','Pune'),(8,'Deepa','Mumbai'),
(9,'Nitin','Nagpur'),(10,'Pooja','Pune');
CREATE TABLE Account(C_Id INT, Acc_Type VARCHAR(50), Amount INT);
INSERT INTO Account VALUES
(1,'Current',5000),(2,'Saving',20000),(3,'Saving',70000),(4,'Saving',50000),(6,'Current',35000),
(7,'Loan',30000),(8,'Saving',50000),(9,'Saving',90000),(10,'Loan',8000),(11,'Current',45000);
SELECT c.Cname, a.Acc_Type, a.Amount FROM Customer c JOIN Account a ON c.C_Id = a.C_Id WHERE a.Acc_T
SELECT * FROM Customer NATURAL JOIN Account;
SELECT * FROM Customer c LEFT JOIN Account a ON c.C_Id = a.C_Id;
SELECT * FROM Customer c RIGHT JOIN Account a ON c.C_Id = a.C_Id;
SELECT * FROM Customer WHERE City = (SELECT City FROM Customer WHERE Cname = 'Pooja');
SELECT * FROM Account WHERE Amount < (SELECT AVG(Amount) FROM Account);
SELECT C_Id FROM Account WHERE Amount = (SELECT MAX(Amount) FROM Account);
SELECT Acc_Type, Amount FROM Account a WHERE Amount = (SELECT MIN(Amount) FROM Account a2 WHERE a2.A
SELECT * FROM Account WHERE Amount > (SELECT MAX(Amount) FROM Account WHERE Acc_Type = 'Saving');
```

Line-by-line Explanation:
Line 1: Lists databases on the server — useful to confirm current DBs.
Line 2: Creates a new database named DBMS01.
Line 3: Switches current session to DBMS01 so subsequent commands run there.
Line 4: Creates Customer table with primary key C_Id and non-null Cname.
Line 5: Inserts 10 rows into Customer — multiple-row insert using comma-separated tuples.
Line 6: Creates Account table with C_Id (not primary here), Acc_Type, and Amount.
Line 7: Inserts multiple rows into Account. Note: C_Id=11 has no matching Customer row (demonstrates RIGHT JOIN result).
Line 8: Selects customer name and account details joining Customer and Account on C_Id, filtering only 'Saving' accounts.
Line 9: NATURAL JOIN joins tables by all columns with the same name (here C_Id). Returns rows where C_Id exists in both tables.
Line 10: LEFT JOIN returns all customers and account info when present; customers without account show NULL in account columns (e.g., Samata).
Line 11: RIGHT JOIN returns all accounts and customer info when present; accounts without customer show NULL in customer columns (e.g., C_Id=11).
Line 12: Subquery in WHERE: finds customers who live in the same city as 'Pooja'. The inner query returns Pooja's City ('Pune').
Line 13: Subquery with aggregate AVG: selects accounts with Amount less than the average amount across all accounts.
Line 14: Finds C_Id of account holder who has the maximum Amount (MAX aggregate in subquery).

Line 15: Correlated subquery: for each account row 'a' it finds the minimum Amount for that Acc_Type and returns rows equal to that min — shows min per account type.
Line 16: Checks if any account has Amount greater than the maximum Amount among 'Saving' accounts — returns empty set here.

Notes & Corrections:
• When inserting multiple rows, separate tuples with commas — earlier attempt without commas caused syntax error.
• NATURAL JOIN is fragile: if tables share unexpected same-named columns it may join on them unintentionally.

Sample Output / Result:

Cname	Acc_Type	Amount
Seema	Saving	20000
Amita	Saving	70000
Rakesh	Saving	50000
Deepa	Saving	50000
Nitin	Saving	90000

Possible Viva Questions & Answers:
Q: What is the difference between INNER JOIN and NATURAL JOIN?
A: INNER JOIN requires an ON clause to specify join condition; NATURAL JOIN automatically joins on columns with the same names in both tables — use carefully to avoid ambiguous joins.
Q: Why did SELECT * FROM Account WHERE Account < (SELECT AVG(Amount) FROM Account) fail earlier?
A: Because 'Account' is a table name, not a column. The correct column is 'Amount' (fix: WHERE Amount < ...).

2) Library Return_Book Stored Procedure (chit 2.txt)

Purpose:
Demonstrates stored procedure with IN parameters, variable declaration, handlers for NOT FOUND, DATE diff calculation, conditional fine computation, UPDATE and INSERT into Fine table, and SIGNAL for errors.

Code:
```
CREATE DATABASE libraryDB;
USE libraryDB;
CREATE TABLE Fine (DateOfIssue DATE, NameOfBook VARCHAR(100), Status CHAR(1));
CREATE TABLE Borrower (Roll_no INT PRIMARY KEY, Name VARCHAR(50), Date DATE, NameOfBook VARCHAR(100)
INSERT INTO Borrower VALUES (1, 'Rohan', '2025-09-10', 'DBMS', 'I'), (2, 'Priya', '2025-09-25', 'CN'
DROP PROCEDURE IF EXISTS Return_Book;
DELIMITER /
CREATE PROCEDURE Return_Book(IN p_roll INT, IN p_book VARCHAR(100))
BEGIN
  DECLARE v_issue DATE;
  DECLARE v_days INT DEFAULT 0;
  DECLARE v_fine INT DEFAULT 0;
  DECLARE no_row INT DEFAULT 0;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_row = 1;
  SELECT DateOfIssue INTO v_issue FROM Borrower WHERE Roll_no = p_roll AND NameOfBook = p_book;
  IF no_row = 1 THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No such borrower or book found!';
  ELSE
    SET v_days = DATEDIFF(CURDATE(), v_issue);
    IF v_days <= 15 THEN SET v_fine = 0;
    ELSEIF v_days <= 30 THEN SET v_fine = (v_days - 15) * 5;
    ELSE SET v_fine = (15 * 5) + (v_days - 30) * 50; END IF;
    UPDATE Borrower SET Status = 'R' WHERE Roll_no = p_roll AND NameOfBook = p_book;
    IF v_fine > 0 THEN INSERT INTO Fine (Roll_no, Date, Amt) VALUES (p_roll, CURDATE(), v_fine); END
    SELECT p_roll AS Roll_no, p_book AS Book, v_issue AS DateOfIssue, v_days AS DaysPassed, v_fine A
  END IF;
END/
DELIMITER ;
CALL Return_Book(3, 'OS');
```

Line-by-line Explanation:
Line 1: Creates the libraryDB database.
Line 2: Use the libraryDB database context.
Line 3: Creates Fine table to store fines (schema shown in original had different names; adapted for example).
Line 4: Creates Borrower table with primary key Roll_no and fields for date/book/status.
Line 5: Inserts sample borrower records; Status 'I' means Issued.
Line 6: Drops existing procedure if present to allow recreation.
Line 7: Changes delimiter so procedure body can contain semicolons.
Line 8: Procedure header: two IN parameters — roll number and book name.
Line 9: Begin block starts the procedure body.
Line 10: v_issue will hold the stored DateOfIssue.
Line 11: v_days initialized to 0; will store days elapsed since issue.
Line 12: v_fine initialized to 0; holds computed fine.
Line 13: no_row acts as flag when SELECT finds no row; default 0.
Line 14: If SELECT returns no rows, the CONTINUE HANDLER sets no_row=1 instead of raising error immediately.
Line 15: Selects DateOfIssue for the given borrower+book into v_issue variable.
Line 16: If no_row=1 then raise a custom error using SIGNAL to inform caller of missing data.
Line 17: Else compute days passed between current date and issue date using DATEDIFF.
Line 18: If within 15 days fine is 0.
Line 19: If between 16 and 30 days fine = (v_days-15)*5 per day after 15 days.
Line 20: If >30 days fine = 15*5 + (v_days-30)*50 (higher penalty beyond 30 days).
Line 21: Marks the book as returned in Borrower table by setting Status='R'.
Line 22: If fine>0 insert a record into Fine table with today's date and amount.

Line 23: Return a result set summarizing Roll, Book, DateOfIssue, DaysPassed, FineAmount.
Line 24: Ends procedure and resets delimiter to ;
Line 25: Example CALL invoking procedure for Roll_no 3 and book 'OS' — shows returned summary.

Notes & Corrections:
• Ensure table Fine schema matches INSERT columns (in original snippet some column names differ — adapt accordingly).
• Using SIGNAL with SQLSTATE '45000' raises a user-defined error visible to client.

Sample Output / Result:

```
+---------+------+-------------+------------+------------+
| Roll_no | Book | DateOfIssue | DaysPassed | FineAmount |
+---------+------+-------------+------------+------------+
|       3 | OS   | 2025-09-20  |         23 |         40 |
+---------+------+-------------+------------+------------+
```

Possible Viva Questions & Answers:
Q: Why is DELIMITER changed when creating procedures?
A: Because the procedure body contains semicolons; changing delimiter (e.g., to /) lets the server know where the procedure ends.
Q: What does the CONTINUE HANDLER FOR NOT FOUND do?
A: It intercepts the 'no rows' condition from SELECT INTO and sets a flag instead of causing an exception, allowing custom handling.

3) Oracle PL/SQL — Attendance Update Procedure (chit 3.txt)

Purpose:
Demonstrates DECLARE block, local variables, SELECT INTO, IF..ELSE, UPDATE, exception handling (NO_DATA_FOUND), and DBMS_OUTPUT for messages in Oracle.

Code:
```
CREATE TABLE Stud( Roll NUMBER PRIMARY KEY, Att NUMBER, Status CHAR(2) );
INSERT INTO Stud VALUES (1,85,NULL);
INSERT INTO Stud VALUES (2,60,NULL);
INSERT INTO Stud VALUES (3,75,NULL);
INSERT INTO Stud VALUES (4,50,NULL);
COMMIT;
SET SERVEROUTPUT ON;
DECLARE
  v_roll NUMBER := &Roll;
  v_att Stud.Att%TYPE;
BEGIN
  SELECT Att INTO v_att FROM Stud WHERE Roll = v_roll;
  IF v_att < 75 THEN
    UPDATE Stud SET Status = 'D' WHERE Roll = v_roll;
    DBMS_OUTPUT.PUT_LINE('Term not granted');
  ELSE
    UPDATE Stud SET Status = 'ND' WHERE Roll = v_roll;
    DBMS_OUTPUT.PUT_LINE('Term granted');
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Roll number not found!');
END;
```

Line-by-line Explanation:
Line 1: Creates STUD table with Roll (PK), attendance (Att) and Status.
Line 2-5: Inserts sample student rows with attendance percentages; Status initially NULL.
Line 6: Commit the inserts to persist data.
Line 7: Enable DBMS_OUTPUT so PL/SQL messages are shown.
Line 8-9: Declare v_roll (bound to user substitution variable &Roll;) and v_att typed like Stud.Att column.
Line 10: BEGIN starts PL/SQL executable section.
Line 11: SELECT INTO fetches attendance of the requested roll into v_att. If no row, NO_DATA_FOUND is raised.
Line 12-13: If attendance <75 update Status='D' (Denied) and output message 'Term not granted'.
Line 14-16: Else update Status='ND' (Not Denied / Granted) and output 'Term granted'.
Line 17-19: Exception handler catches NO_DATA_FOUND to print user-friendly message when roll absent.

Possible Viva Questions & Answers:
Q: What is the purpose of DBMS_OUTPUT.PUT_LINE?
A: It prints text to the SQL*Plus console (useful for debugging and informative messages).
Q: What happens if SELECT INTO returns multiple rows?
A: A TOO_MANY_ROWS exception is raised; SELECT INTO must return exactly one row.

4) Course-Instructor-Student Schema — GROUP BY, HAVING, LIKE (chit 4.txt)

Purpose:
Create course, instructor, and student tables; demonstrate GROUP BY with HAVING, UPDATE, DESC, LIKE, and joins with WHERE filters.

Code:
```
CREATE DATABASE DBMS04; USE DBMS04;
CREATE TABLE course (course_id VARCHAR(10) PRIMARY KEY, title VARCHAR(100), dept_name VARCHAR(50), c
CREATE TABLE instructor (T_ID INT PRIMARY KEY, name VARCHAR(50), dept_name VARCHAR(50), salary INT);
CREATE TABLE student (S_ID INT PRIMARY KEY, name VARCHAR(50), dept_name VARCHAR(50), tot_cred INT, c
INSERT INTO course VALUES ('CS101', 'DBMS', 'Computer', 4), ('CS102', 'Data Structures', 'Computer',
INSERT INTO instructor VALUES (101, 'Amol', 'Computer', 40000), (102, 'Amit', 'Computer', 45000), (1
INSERT INTO student VALUES (1, 'Ram', 'Computer', 120, 'CS101'), (2, 'Shyam', 'Electrical', 100, 'EE
SELECT dept_name, AVG(salary) AS avg_salary FROM instructor GROUP BY dept_name HAVING AVG(salary) >
UPDATE instructor SET salary = salary * 1.10 WHERE dept_name = 'Computer';
SELECT name FROM instructor WHERE name NOT IN ('Amol','Amit');
SELECT name FROM student WHERE name LIKE '%am%';
SELECT s.name FROM student s JOIN course c ON s.course_id = c.course_id WHERE s.dept_name = 'Compute
```

Line-by-line Explanation:
Line 1: Create and select DBMS04 database to work in.
Line 2: Course table with primary key and fields for title, department, and credits.
Line 3: Instructor table with salary and department; T_ID is primary key.
Line 4: Student table referencing course table via foreign key course_id; enforces referential integrity.
Line 5: Insert multiple rows into course table — note the number and order of columns must match sentence.
Line 6: Insert instructors with dept and salary; used later for aggregation and updates.
Line 7: Insert students including their course_id linking to course table.
Line 8: GROUP BY dept_name and HAVING filters groups whose average salary >42000; HAVING filters after aggregation.
Line 9: Increase salary by 10% for instructors in Computer department — UPDATE with WHERE clause changes rows in place.
Line 10: Simple NOT IN filter to exclude specified names.
Line 11: LIKE with pattern '%am%' finds names containing substring 'am' (case depends on collation).
Line 12: Join student and course to find students from Computer dept who are taking DBMS course.

Possible Viva Questions & Answers:
Q: What is the difference between WHERE and HAVING?
A: WHERE filters rows before aggregation; HAVING filters groups after aggregation.
Q: Why use FOREIGN KEY in student table?
A: To ensure course_id refers to a valid course row; helps maintain data integrity and prevents orphaned references.

5) MergeRollCall — Cursor-based Merge Procedure (chit 5.txt)

Purpose:
Demonstrates cursor, loop, FETCH, CONTINUE HANDLER, NOT EXISTS check, and inserting new rows while merging two rollcall tables.

Code:
```
CREATE TABLE O_RollCall (RollNo INT PRIMARY KEY, Name VARCHAR(50));
CREATE TABLE N_RollCall (RollNo INT PRIMARY KEY, Name VARCHAR(50));
INSERT INTO O_RollCall VALUES (1,'Rohan'), (2,'Priya'), (3,'Madhura');
INSERT INTO N_RollCall VALUES (2,'Priya'), (3,'Madhura'), (4,'Anita'), (5,'Rahul');
DELIMITER $$
CREATE PROCEDURE MergeRollCall()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE v_RollNo INT; DECLARE v_Name VARCHAR(50);
  DECLARE cur_N CURSOR FOR SELECT RollNo, Name FROM N_RollCall;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
  OPEN cur_N;
  read_loop: LOOP
    FETCH cur_N INTO v_RollNo, v_Name;
    IF done THEN LEAVE read_loop; END IF;
    IF NOT EXISTS (SELECT 1 FROM O_RollCall WHERE RollNo = v_RollNo) THEN
      INSERT INTO O_RollCall (RollNo, Name) VALUES (v_RollNo, v_Name);
    END IF;
  END LOOP;
  CLOSE cur_N;
END $$
DELIMITER ;
CALL MergeRollCall();
SELECT * FROM O_RollCall;
```

Line-by-line Explanation:
Line 1-2: Create original and new rollcall tables with primary keys.
Line 3-4: Populate both tables; note overlapping roll numbers demonstrate de-duplication logic.
Line 5-7: Change delimiter and create stored procedure MergeRollCall with cursor cur_N iterating over N_RollCall.
Line 8: Handler sets 'done' flag when cursor fetch finds no more rows.
Line 9-12: Loop: fetch each row, break when done, insert into O_RollCall only if RollNo does not already exist.
Line 13: Close cursor and end procedure; call it and verify O_RollCall now contains merged unique rows.

Notes & Corrections:
• A set-based alternate: INSERT INTO O_RollCall (RollNo, Name) SELECT n.RollNo, n.Name FROM N_RollCall n LEFT JOIN O_RollCall o ON n.RollNo=o.RollNo WHERE o.RollNo IS NULL; — faster and avoids cursor.

Sample Output / Result:
```
+--------+---------+
| RollNo | Name    |
+--------+---------+
|      1 | Rohan   |
|      2 | Priya   |
|      3 | Madhura |
|      4 | Anita   |
|      5 | Rahul   |
+--------+---------+
```

Possible Viva Questions & Answers:
Q: Why use a cursor here instead of a single SQL MERGE?
A: Cursors allow procedural control and conditional logic per row; some MySQL versions lack MERGE — but set-based SQL (INSERT ... SELECT with LEFT JOIN/WHERE NULL) would be more efficient.

Q: What does CONTINUE HANDLER FOR NOT FOUND do?
A: Prevents exception on cursor exhaustion by setting a flag; allows graceful loop termination.

6) proc_Grade — Assign Class using Cursor (chit 6.txt)

Purpose:
Uses a cursor to iterate Stud_Marks, classifies students into Distinction/First/Higher Second/Fail, and inserts into Result table.

Code:
```
CREATE TABLE Stud_Marks (Roll_No INT AUTO_INCREMENT PRIMARY KEY, Name VARCHAR(50), Total_Marks INT);
CREATE TABLE Result (Roll_No INT, Name VARCHAR(50), Class VARCHAR(50));
INSERT INTO Stud_Marks (Name, Total_Marks) VALUES ('John',1450),('Priya',960),('Amit',880),('Sneha',
DELIMITER //
CREATE PROCEDURE proc_Grade()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE v_roll INT; DECLARE v_name VARCHAR(50); DECLARE v_total INT; DECLARE v_class VARCHAR(50);
  DECLARE cur_stud CURSOR FOR SELECT Roll_No, Name, Total_Marks FROM Stud_Marks;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
  OPEN cur_stud;
  read_loop: LOOP
    FETCH cur_stud INTO v_roll, v_name, v_total;
    IF done THEN LEAVE read_loop; END IF;
    IF v_total BETWEEN 990 AND 1500 THEN SET v_class = 'Distinction';
    ELSEIF v_total BETWEEN 900 AND 989 THEN SET v_class = 'First Class';
    ELSEIF v_total BETWEEN 825 AND 899 THEN SET v_class = 'Higher Second Class';
    ELSE SET v_class = 'Fail'; END IF;
    INSERT INTO Result (Roll_No, Name, Class) VALUES (v_roll, v_name, v_class);
  END LOOP;
  CLOSE cur_stud;
END //
DELIMITER ;
CALL proc_Grade();
SELECT * FROM Result;
```

Line-by-line Explanation:
Line 1-2: Creates Stud_Marks with auto-increment Roll_No and Result table to store classification.
Line 3: Inserts sample marks; note values reflect different grade buckets.
Line 4-6: Procedure proc_Grade uses cursor cur_stud over Stud_Marks and a done handler for end-of-cursor.
Line 7-10: Loop fetches each student, classifies via BETWEEN ranges and inserts classification into Result.
Line 11: After loop closure, the Result table contains computed class for each student.

Notes & Corrections:
• Ensure grade ranges do not overlap and cover all possible marks. The given ranges start at 825; adjust as per marking scheme.

Sample Output / Result:
```
+---------+-------+---------------------+
| Roll_No | Name  | Class               |
+---------+-------+---------------------+
|       1 | John  | Distinction         |
|       2 | Priya | First Class         |
|       3 | Amit  | Higher Second Class |
|       4 | Sneha | Fail                |
+---------+-------+---------------------+
```

Possible Viva Questions & Answers:
Q: Why prefer set-based operations over cursor loops?
A: Set-based operations (single INSERT...SELECT with CASE) are typically much faster and simpler; use cursor only when row-by-row logic is unavoidable.
Q: What does BETWEEN include?
A: BETWEEN is inclusive of both endpoints (e.g., BETWEEN 900 AND 989 includes 900 and 989).

7) grade1() Function — Insert Result Based on Marks (chit 7.txt)

Purpose:
Defines a stored function that classifies a student's grade by roll number, inserts into Result table, and returns an integer status code.

Code:
```
CREATE DATABASE DBMS05function; USE DBMS05function;
CREATE TABLE stud_marks(rollno INT PRIMARY KEY, tot_marks INT);
CREATE TABLE result(rollno INT, grade VARCHAR(10), FOREIGN KEY(rollno) REFERENCES stud_marks(rollno)
INSERT INTO stud_marks VALUES (20,1200),(21,790),(22,988),(23,888);
DELIMITER /
CREATE FUNCTION grade1(rolln INT) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE marks INT;
  SELECT tot_marks INTO marks FROM stud_marks WHERE rollno = rolln;
  IF marks >= 825 AND marks < 900 THEN INSERT INTO result VALUES (rolln,'HSC');
  ELSEIF marks >= 900 AND marks < 990 THEN INSERT INTO result VALUES (rolln,'FC');
  ELSEIF marks > 990 THEN INSERT INTO result VALUES (rolln,'Dist');
  ELSE INSERT INTO result VALUES (rolln,'SC'); END IF;
  RETURN 1000;
END/
DELIMITER ;
SELECT grade1(20); SELECT grade1(21); SELECT grade1(22); SELECT grade1(23);
SELECT * FROM result;
```

Line-by-line Explanation:
Line 1-3: Create DB and tables; result table has foreign key referencing stud_marks to ensure consistency.
Line 4: Insert sample marks for 4 students with specific roll numbers.
Line 5-7: Define function grade1 that returns an integer; mark deterministic (no randomness).
Line 8-9: Fetch marks for provided roll number into local variable 'marks'.
Line 10-14: Classify marks into grade strings and insert into result table accordingly; finally return a status integer (1000).
Line 15: Calling the function for different roll numbers inserts corresponding rows into result table and returns 1000 (fixed).

Notes & Corrections:
• Prefer PROCEDURE for operations that perform inserts; functions should avoid side-effects where possible.
• Ensure proper DELIMITER usage when creating functions containing semicolons.

Sample Output / Result:
```
+--------+-------+
| rollno | grade |
+--------+-------+
|     20 | Dist  |
|     21 | SC    |
|     22 | FC    |
|     23 | HSC   |
+--------+-------+
```

Possible Viva Questions & Answers:
Q: Is it good practice for functions to have side-effects (like INSERT)?
A: Generally no — functions are expected to be deterministic and side-effect free. Use stored procedures for operations that modify data.
Q: Why is RETURN used here if the result is inserted into a table?
A: The RETURN provides a status code to caller; but the main effect here is the insert — design could be improved by returning meaningful values or converting to procedure.

8) Instructor-Course-Teaches — View and ALTER (chit 8.txt)

Purpose:
Demonstrates table creation for instructor/course/teaches/student, inserting data, JOINs, view creation, ALTER COLUMN renaming, and deleting rows with NULLs.

Code:
```
CREATE DATABASE DBMS08; USE DBMS08;
CREATE TABLE instructor (T_ID INT PRIMARY KEY, name VARCHAR(50), dept_name VARCHAR(50), salary DECIM
CREATE TABLE course (course_id VARCHAR(10) PRIMARY KEY, title VARCHAR(100), dept_name VARCHAR(50), c
CREATE TABLE teaches (T_ID INT, course_id VARCHAR(10), sec_id VARCHAR(10), semester VARCHAR(10), yea
CREATE TABLE student (S_ID INT PRIMARY KEY, name VARCHAR(50), dept_name VARCHAR(50), tot_cred INT);
INSERT INTO instructor VALUES (1,'Dr. Smith','CSE',80000),(2,'Dr. Johnson','ECE',75000),(3,'Dr. Lee'
INSERT INTO course VALUES ('C101','DBMS','CSE',4),('C102','Networks','ECE',3),('C103','Thermodynamic
INSERT INTO teaches VALUES (1,'C101','S1','Spring',2024),(1,'C101','S2','Fall',2024),(2,'C102','S1',
INSERT INTO student VALUES (101,'Alice','CSE',120),(102,'Bob','ECE',100),(103,'Charlie',NULL,80);
SELECT DISTINCT i.name, t.semester FROM instructor AS i JOIN teaches AS t ON i.T_ID = t.T_ID ORDER B
CREATE VIEW student_details AS SELECT S_ID, name, dept_name, tot_cred FROM student;
SELECT * FROM student_details;
ALTER TABLE student CHANGE COLUMN dept_name department_name VARCHAR(50);
DELETE FROM student WHERE department_name IS NULL;
SELECT * FROM student_details;
```

Line-by-line Explanation:
Line 1: Creates DB and switches to it.
Line 2-5: Creates instructor, course, teaches (with FK constraints), and student tables.
Line 6-9: Insert sample data across these tables; note one student has NULL dept to show deletion later.
Line 10: Join instructor and teaches to list distinct instructor names by semester; ORDER BY sorts output.
Line 11: Create view 'student_details' encapsulating selected student columns for easy querying.
Line 12: Select from view to display students.
Line 13: ALTER TABLE changes column name 'dept_name' to 'department_name' preserving type and length.
Line 14: Delete rows where department_name IS NULL (removes Charlie).
Line 15: Selecting from view may fail if view references old column name — shows importance of view maintenance; in MySQL the view became invalid until recreated.

Possible Viva Questions & Answers:
Q: What happens to views when underlying table columns are renamed?
A: Views referencing renamed columns can become invalid; you must DROP and recreate or ALTER view accordingly.
Q: Why use DISTINCT in the SELECT with JOIN?
A: DISTINCT removes duplicate rows that may arise due to multiple teaches entries for same instructor/semester combination.

9) MongoDB — insertMany, aggregate, find, createIndex (chit 9.txt)

Purpose:
Demonstrates basic MongoDB shell operations: inserting documents, aggregation pipeline ($match, $group, $avg), querying with range operators, projection, and creating an index.

Code:
```
mongosh
use mydb
db.orderinfo.insertMany([{cust_id:123, cust_name: 'abc', status: 'A', price:250}, {cust_id:124, cust
db.orderinfo.aggregate([ {$match: {status:'A'}}, {$group: {_id: '$cust_id', avg_price: {$avg: '$pric
db.orderinfo.find({price: {$gt:100, $lt:1000}}, {status:1, _id:0});
db.orderinfo.createIndex({cust_id:1});
db.orderinfo.getIndexes();
```

Line-by-line Explanation:
Line 1-2: Start mongosh and switch to database 'mydb' (created implicitly on insert).
Line 3: Insert multiple documents into 'orderinfo' collection using insertMany — JSON-like documents.
Line 4: Aggregation pipeline: $match filters documents with status 'A', $group groups by cust_id and computes average price per customer (here each cust_id unique so avg equals price).
Line 5: Find documents where price >100 and <1000, projecting only 'status' field while suppressing _id (set _id:0).
Line 6: Create an ascending index on cust_id to speed queries filtering by cust_id.
Line 7: List indexes on collection (shows _id_ and newly created cust_id_1).

Sample Output / Result:
```
[ { _id: 126, avg_price: 1000 }, { _id: 123, avg_price: 250 }, { _id: 124, avg_price: 500 } ]
```

Possible Viva Questions & Answers:
Q: Why use aggregation pipeline instead of map-reduce?
A: Aggregation pipeline is simpler, faster and widely used for common data processing tasks; map-reduce is more flexible but slower.
Q: What does createIndex do?
A: Creates an index to optimize query performance for the specified key(s); beware of write overhead and storage cost.

10) Trigger — BEFORE DELETE to archive deleted rows (chit 11.txt)

Purpose:
Demonstrates creating a trigger that copies a row to an audit table before it is deleted, preserving history.

Code:
```
CREATE DATABASE dbms11; USE dbms11;
CREATE TABLE libaudit (roll_no INT PRIMARY KEY, book_name VARCHAR(20));
INSERT INTO libaudit VALUES (4,'HCI'), (12,'SPOS'), (18,'DBMS'), (19,'CNS');
CREATE TABLE libaudit1 (roll_no INT PRIMARY KEY, book_name VARCHAR(20));
DELIMITER /
CREATE TRIGGER libtrig BEFORE DELETE ON libaudit FOR EACH ROW
BEGIN
  INSERT INTO libaudit1 VALUES (OLD.roll_no, OLD.book_name);
END/
DELIMITER ;
DELETE FROM libaudit WHERE roll_no = 19;
SELECT * FROM libaudit;
SELECT * FROM libaudit1;
```

Line-by-line Explanation:
Line 1-2: Create database and primary libaudit table to hold current entries.
Line 3: Insert sample rows into libaudit.
Line 4: Create libaudit1 audit table where deleted rows will be stored.
Line 5-8: Create trigger libtrig that fires BEFORE DELETE on libaudit for each row; uses OLD pseudorecord to reference values about to be deleted and inserts them into libaudit1.
Line 9: After deleting roll_no=19 from libaudit, that row is preserved in libaudit1 (audit).

Sample Output / Result:
```
+---------+-----------+
| roll_no | book_name |
+---------+-----------+
|       4 | HCI       |
|      12 | SPOS      |
|      18 | DBMS      |
+---------+-----------+

-- And libaudit1 contains the deleted row:
+---------+-----------+
| roll_no | book_name |
+---------+-----------+
|      19 | CNS       |
+---------+-----------+
```

Possible Viva Questions & Answers:
Q: Why use BEFORE DELETE instead of AFTER DELETE?
A: BEFORE DELETE allows access to OLD values and can prevent deletion (by SIGNAL) or archive data before it is gone. AFTER DELETE runs after the row is deleted — both can be used depending on needs.
Q: What are OLD and NEW in triggers?
A: OLD and NEW are pseudorecords representing the row state before and after the triggering action; OLD exists for DELETE/UPDATE, NEW for INSERT/UPDATE.