INFS1200 — Week 6 Notes

Table of contents

Basic SQL Syntax: Data Definition and Data Manipulation Language

Module 3, part 1. Worked examples use two running schemas — a Company database and a Movie database — introduced below.

Where we are

Module 1 covered conceptual modelling (the ER diagram); Module 2 covered the relational model and ER-to-relational mapping. Module 3 is about **expressing queries against a relational schema** using SQL. Relational algebra (Codd's formal query language) is the "logic engine" behind databases — precise, composable, and optimisable — but SQL is the **declarative** language we actually use to talk to a DBMS: you say *what* you want, not *how* to compute it. SQL is (almost) universal across Postgres, MySQL, Oracle, SQL Server, etc., and is the query interface behind tools like Power BI, Tableau, and most data-science workflows.

Example schemas used this module

```
Department [dNumber, dName, mgrSSN, mgrStartDate]
Employee   [ssn, name, dob, address, sex, salary, mgrSSN, dNum]
Sale       [itemID, custID, timestamp, price, salesPerson]
Item       [itemID, category, colour]
Customer   [custID, cname, gender, dob]


Movie     [movieID, title, year]
StarsIn   [movieID, starID, role]
MovieStar [starID, name, gender]
```

Foreign keys: `Employee.mgrSSN → Employee.ssn` (self-referencing — recursive relationship), `Employee.dNum → Department.dNumber`, `Department.mgrSSN → Employee.ssn`, `Sale.itemID → Item.itemID`, `Sale.custID → Customer.custID`, `Sale.salesPerson → Employee.ssn`, `StarsIn.movieID → Movie.movieID`, `StarsIn.starID → MovieStar.starID`.

The three (main) types of SQL statement

- **DDL** (Data Definition Language) — statements that define/change the database *schema* (`CREATE`, `ALTER`, `DROP`).
- **DML** (Data Manipulation Language) — statements that manipulate *data* (`INSERT`, `UPDATE`, `DELETE`, `SELECT`).

- **DCL** (Data Control Language) — transaction control, semantic integrity (triggers/assertions), authorisation/privilege management, physical storage parameters (file structures, indexes), role-based security controls.

A relational query language should support **INSERT** (new tuples), **DELETE** (remove tuples), **UPDATE** (change attribute values), and **SELECT** (retrieve attributes/tuples/relations).

Data Definition Language (DDL)

DROP TABLE

```
DROP TABLE <table name> [CASCADE];
```

Drops all constraints defined on the table (including constraints *in other tables* that reference it), deletes all tuples, and removes the table definition from the system catalog.

CREATE TABLE

```
CREATE TABLE <table name>
   (<column name> <column type> [<attribute constraint>]
   {, <column name> <column type> [<attribute constraint>]}
   [<table constraint> {, <table constraint>}])
```

Notation used throughout this module: `KEYWORD`, `<argument>`, `[optional]`, `{repeatable}`, `…|choice|…`. Key/entity/referential integrity constraints are specified after the attributes are declared; domain constraints are specified per-attribute (directly, or via a `CREATE DOMAIN`).

Entity table example — `Item [itemID, category, colour]`:

```
CREATE TABLE Item (
    itemID   INTEGER,
    category ENUM('food', 'clothing', 'furniture'),
    colour   CHAR(3),
    PRIMARY KEY (itemID));
```

Relationship table example — `Sale [itemID, custID, timestamp, price, salesPerson]`, a ternary-degree relation between `Item`, `Customer` and `Employee`:

```
CREATE TABLE Sale (
    itemID      INTEGER,
    custID      INTEGER,
    timestamp   TIMESTAMP,
    price       DOUBLE(8, 2),
    salesPerson CHAR(9),
    PRIMARY KEY (itemID, custID, timestamp),
    FOREIGN KEY (itemID) REFERENCES Item(itemID),
    FOREIGN KEY (custID) REFERENCES Customer(custID),
    FOREIGN KEY (salesPerson) REFERENCES Employee(ssn));
```

Constraints

Constraints are rules that limit what data can go into a table:

- **PRIMARY KEY** — attribute value is unique and not null.
- **FOREIGN KEY** — attribute value must exist in the referenced (parent) table.
- **CHECK** — attribute value(s) must satisfy a predefined condition.
- **UNIQUE** — attribute value is unique *or null* (unlike a primary key).

`CHECK` constraints are semantic constraints over a single table, evaluated whenever tuples are inserted or modified:

```
CREATE TABLE Sale (
    itemID      INTEGER,
    custID      INTEGER,
    timestamp   TIMESTAMP,
    price       DOUBLE(8, 2),
    salesPerson CHAR(9),
    PRIMARY KEY (itemID, custID, timestamp),
    FOREIGN KEY (itemID) REFERENCES Item(itemID),
    FOREIGN KEY (custID) REFERENCES Customer(custID),
    FOREIGN KEY (salesPerson) REFERENCES Employee(ssn),
    CHECK (price >= 8.50 AND price < 150000));
```

Naming constraints

```
CREATE TABLE Item (
    itemID   INTEGER,
    category ENUM('food', 'clothing', 'furniture'),
    colour   CHAR(3),
    CONSTRAINT item_pk PRIMARY KEY (itemID));
```

Giving a constraint a name has two benefits: (1) clearer error messages — a violation names the constraint; (2) the constraint can be modified/removed later, e.g. `ALTER TABLE Item DROP CONSTRAINT item_pk;`.

Question 1 — DDL constraints

Which SQL statement correctly implements `Student [id, firstName, lastName]` where `{firstName, lastName}` is unique?

```
-- A
CREATE TABLE Student (
    id INT NOT NULL, firstName VARCHAR(50), lastName VARCHAR(50),
    PRIMARY KEY (id));

-- B
CREATE TABLE Student (
    id INT NOT NULL, firstName VARCHAR(50), lastName VARCHAR(50),
    PRIMARY KEY (firstName, lastName));

-- C
CREATE TABLE Student (
    id INT NOT NULL, firstName VARCHAR(50), lastName VARCHAR(50),
    PRIMARY KEY (id), PRIMARY KEY (firstName, lastName));

-- D
CREATE TABLE Student (
    id INT NOT NULL, firstName VARCHAR(50), lastName VARCHAR(50),
    PRIMARY KEY (id), UNIQUE(firstName, lastName));
```

Solution

D. A has no uniqueness constraint on `{firstName, lastName}` at all. B makes `{firstName, lastName}` the primary key instead of `id` (wrong key). C is invalid SQL — a table cannot declare two `PRIMARY KEY` clauses. D correctly keeps `id` as the primary key *and* adds a `UNIQUE` constraint over `{firstName, lastName}`.

Referential integrity: handling broken links

If a row that other rows depend on (e.g. a department's manager) is deleted, what should happen to the dependent rows?

Action	Effect
`ON DELETE RESTRICT` (default)	Disallows the deletion
`ON DELETE CASCADE`	Deletes dependent rows too
`ON DELETE SET NULL`	Breaks the link, but doesn't delete other rows
`ON DELETE SET DEFAULT`	Uses a predefined fallback value

The same options apply to updates (`ON UPDATE ...`). These options are attached to the `FOREIGN KEY` clause, and apply **in the direction the foreign key points** — from the referencing (child) table's row being orphaned when the referenced (parent) table's row is removed, not the reverse.

```
CREATE TABLE Sale (
    itemID      INTEGER,
    custID      INTEGER,
    timestamp   TIMESTAMP,
    price       DOUBLE(8, 2),
    salesPerson CHAR(9),
    PRIMARY KEY (itemID, custID, timestamp),
    CONSTRAINT itemID_fk FOREIGN KEY (itemID) REFERENCES Item(itemID)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT custID_fk FOREIGN KEY (custID) REFERENCES Customer(custID)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT ssn_fk FOREIGN KEY (salesPerson) REFERENCES Employee(ssn)
        ON DELETE RESTRICT ON UPDATE CASCADE);
```

Question 2 — Designing with referential integrity

Given `Student [studentID, name, advisorID]`, `Professor [professorID, name]`, `Student.advisorID references Professor.professorID` — *which statement best explains the design rationale for using* `ON DELETE SET NULL` *on* `advisorID`*?*

A. It automatically reassigns students to a new advisor. B. It prevents the professor's deletion unless all advisees are manually updated. C. It allows deletion of a professor without losing student records, marking that they no longer have an advisor. D. It deletes all students advised by the professor.

i Solution

C. `ON DELETE SET NULL` preserves student records and referential integrity by clearing the advisor reference — it does *not* reassign (A), does not block deletion (B, that's `RESTRICT`), and does not delete students (D, that's `CASCADE`).

Question 3 — Implementing referential integrity

```
CREATE TABLE ParkingPermit (
    pID     INTEGER,
    staffID INTEGER,
    PRIMARY KEY (pID),
    FOREIGN KEY (staffID) REFERENCES Staff(staffID) ON DELETE CASCADE);
```

Given a row `pID = 1000, staffID = 5678` — which is correct? A. Deleting `pID = 1000` cascades to delete `staffID = 5678` in `Staff`. B. Deleting `staffID = 5678` in `Staff` cascades to delete matching rows in `ParkingPermit`. C. Both A and B. D. None of the above.

> **i** Solution
>
> **B.** `ON DELETE CASCADE` only propagates **from the referenced (parent) table to the referencing (child) table** — deleting the `Staff` row cascades to `ParkingPermit`. It never works in reverse (deleting a `ParkingPermit` row has no defined effect on `Staff`).

ALTER TABLE

```
ALTER TABLE <table name>
    ADD <column name> <column type> [<attribute constraint>]
        {, <column name> <column type> [<attribute constraint>]}
  | DROP <column name> [CASCADE]
  | MODIFY <column name> <column-options>
  | ADD <constraint name> <constraint-options>
  | DROP <constraint name> [CASCADE];
```

Used for schema evolution — adding/dropping columns, changing a column definition, adding/dropping constraints. To *alter* a constraint, it must be dropped and re-added (commercial products vary in exact syntax).

```
-- Add an attribute (existing rows get NULL, so NOT NULL can't be used)
ALTER TABLE Employee ADD job VARCHAR(12);

-- Drop an attribute (CASCADE if other tables' FKs reference it)
ALTER TABLE Employee DROP address;

-- Add a constraint
ALTER TABLE Sale ADD CONSTRAINT ChkAge CHECK (price BETWEEN 10 AND 10000);
```

```
-- Drop a constraint (must have been named)
ALTER TABLE Sale DROP CONSTRAINT itemID_fk;
```

Cyclical foreign key dependencies

`Department.mgrSSN` → `Employee.ssn` and `Employee.dNum` → `Department.dNumber` form a cycle — you can't create either table first with its foreign keys in place, since the other table doesn't exist yet. Solution: create both tables *without* the foreign keys, insert the data, then `ALTER TABLE` to add the foreign keys afterwards.

```
CREATE Department (without FKs)
CREATE Employee (without FKs)
INSERT data into Department
INSERT data into Employee
ALTER TABLE Department ADD FOREIGN KEY (mgrSSN) REFERENCES Employee(ssn)
ALTER TABLE Employee ADD FOREIGN KEY (dNum) REFERENCES Department(dNumber)
```

Data Manipulation Language (DML)

Four statements: `INSERT` (add tuples), `UPDATE` (modify existing data), `DELETE` (remove tuples), `SELECT` (retrieve data).

INSERT

```
INSERT INTO <table name>
    [(<column name> {, <column name>})]
    (VALUES (<constant value> {, <constant value>}) | <select statement>);
```

A single-tuple insert lists values in the same order as the columns were declared (or the explicit column list, if given). A multi-tuple insert either comma-separates several value-lists, or loads the result of a query.

```
-- Insert from values
INSERT INTO Customer VALUES
    ('653298653', 'Ronald West', 'Male', '1995-12-30');

-- Insert from a query: create a customer account for every
-- employee in department 1
INSERT INTO Customer (custID, cname, gender, dob)
```

```
    SELECT ssn, name, sex, dob
    FROM   Employee
    WHERE  dNum = 1;
```

DELETE

```
DELETE FROM <table name>
    [WHERE <select condition>];
```

A single `DELETE` can remove zero, one, several, or all tuples from one table. Deletion may **propagate** to other tables if `ON DELETE` referential-triggered actions were declared.

```
DELETE FROM Employee
    WHERE name = 'Ramesh';
```

UPDATE

```
UPDATE <table name>
    SET <column name> = <value expression> {, <column name> = <value expression>}
    [WHERE <select condition>];
```

Tuples are selected for update from a single table; updating a primary key value may propagate to other tables (referencing foreign keys).

```
UPDATE Employee
    SET salary = salary * 1.1
    WHERE name = 'Joyce';
```

Question 4 — Correct use of DELETE

*Given a `Student [id, fName, lName, degree]` table, which query deletes exactly the `CompSci` students **except** `id = 4`?*

A. `DELETE FROM Student WHERE id = 3 AND id = 5 AND id = 12` B. `DELETE FROM Student WHERE fName = 'Diluen' OR fName = 'Peter' OR fName = 'Jason'` C. `DELETE FROM Student WHERE degree = 'CompSci'` D. `DELETE FROM Student WHERE degree = 'CompSci' AND NOT id = 4`

Solution

D. A is a contradiction (`id` can never equal three different values at once — nothing is deleted). B also incidentally deletes a non-CompSci student (Peter Park, `id=10`, is not CompSci). C deletes `id = 4` too, which we want to keep. D correctly restricts to `degree = 'CompSci' AND NOT id = 4`.

Basic SELECT

```
SELECT <attribute list>
FROM   <table list>
[WHERE <condition>];
```

`SELECT` is declarative — you specify what the result should look like, and the DBMS decides the execution plan. The result of any SQL query is itself a table (relation).

Projection

```
SELECT [DISTINCT] (<attribute list> | *)
FROM   <table list>
[WHERE <condition>];
```

SQL relations are **bags/multisets**, not sets — duplicates are *not* eliminated by default. `DISTINCT` eliminates duplicates and enforces set semantics; `*` is a wildcard for "all columns".

```
-- Find the titles of movies
SELECT title
FROM   Movie;

-- Find all the years a movie was produced (with vs without duplicates)
SELECT year FROM Movie;
SELECT DISTINCT year FROM Movie;
```

Question 5 — SQL projection

Given `Scores [team1, team2, score1, score2]` with rows `(Dragons, Tigers, 5, 3)`, `(Carp, Swallows, 4, 6)`, `(Bay Stars, Giants, 2, 1)`, `(Marines, Hawks, 5, 3)`, `(Ham`

`Fighters, Buffaloes, 1, 6)`, `(Lions, Golden Eagles, 8, 12)` — for `SELECT score1, score2 FROM Scores`, which tuple is in the result?

A. `(1,2)` B. `(5,3)` C. `(8,6)` D. All are in the answer E. None are in the answer

> **Solution**
>
> **B.** `(5, 3)` appears twice in the source table (Dragons vs Tigers, Marines vs Hawks) — projection just doesn't eliminate the duplicate by default.

Question 6 — SQL projection with DISTINCT

Same table as Question 5. For `SELECT DISTINCT score1, score2 FROM Scores`, how many tuples are in the output?

A. 6 B. 5 C. 4 D. 3

> **Solution**
>
> **B.** Six rows exist, but `(5, 3)` is duplicated — `DISTINCT` removes one copy, leaving 5 unique tuples.

Projection with expressions

Expressions can use standard arithmetic operators (`+ - * /`) on numeric attributes, and can be given an alias with `AS`.

```
-- Names, salaries, and salaries with a 17% loading, for dept 6
SELECT name, salary, 1.17 * salary AS 'includingSuper'
FROM   Employee
WHERE  dNum = 6;
```

Selection (WHERE clause)

```
SELECT <attribute list>
FROM   <table list>
[WHERE search condition];
```

```sql
-- Find all the male stars
SELECT *
FROM   MovieStar
WHERE  gender = 'Male';

-- Names of employees in dept 4 earning > 25000, or dept 5 earning > 30000
SELECT name
FROM   Employee
WHERE  (dNum = 4 AND salary > 25000) OR (dNum = 5 AND salary > 30000);
```

Question 7 — Selection

Given a `Scores [team, opponent, runsFor, runsAgainst]` table, for

```sql
SELECT *
FROM   Scores
WHERE  (runsFor >= 6 AND runsAgainst <= 4) OR (runsFor < 3 AND opponent = 'Giants');
```

which rows are in the result? A. Swallows vs Carp, 6-4 B. Buffaloes vs Ham Fighters, 6-1 C. Lions vs Golden Eagles, 8-12 D. A and B

> **Solution**
>
> **D.** Swallows vs Carp (`runsFor=6, runsAgainst=4`) satisfies the first clause (`6>=6 AND 4<=4`). Buffaloes vs Ham Fighters (`runsFor=6, runsAgainst=1`) also satisfies the first clause (`6>=6 AND 1<=4`). Lions vs Golden Eagles (`8, 12`) fails both clauses (`12<=4` is false, and `8<3` is false) — it's the archetypal "bigger numbers, so it must match" trap.

Complex WHERE conditions

- `LIKE` — string matching: `%` = zero-or-more arbitrary characters, `_` = any one character. `WHERE title LIKE '%sin%'` finds titles containing "sin" anywhere.
- `IN` — membership in a list: `WHERE lastName IN ('Jones', 'Wong', 'Harrison')`.
- `IS` — null-checking (`=` doesn't work with `NULL`): `WHERE dNum IS NULL`.
- Arithmetic/date functions, `BETWEEN`: `WHERE salary BETWEEN 10000 AND 30000`.

```sql
-- Names of employees in a "Research" department earning 40-60K
SELECT name
FROM   Employee
JOIN   Department ON dNum = dNumber
WHERE  dName LIKE '%Research%' AND salary BETWEEN 40000 AND 60000;
```

(Multi-relation `JOIN` queries like this are covered properly next lecture.)

Sorting (ORDER BY)

```
SELECT [DISTINCT] <target list>
FROM   <table list>
[WHERE search condition]
[ORDER BY column [ASC|DESC] {, column [ASC|DESC]}];
```

The target list can be a column name, expression, or `*`; `column` can be a name or a position in the target list; sorting can use multiple columns.

```
-- Employee names/salaries, ordered by salary descending
SELECT name, salary
FROM   Employee
ORDER BY salary DESC;

-- Employee names/depts/salaries, by dept ascending then salary descending
SELECT dNum, name, salary
FROM   Employee
ORDER BY dNum ASC, salary DESC;
```

Question 8 — Sorting

For `SELECT a, b, c FROM R ORDER BY c DESC, b ASC`, which tuple `t` necessarily precedes `(5, 5, 5)`?

A. `(3, 6, 3)` B. `(1, 5, 5)` C. `(5, 5, 6)` D. All of the above

> **Solution**
>
> **C.** Sorting is primarily by `c DESC`. `(5,5,6)` has `c=6 > 5`, so it sorts strictly before `(5,5,5)` regardless of `a`/`b`. `(3,6,3)` has `c=3 < 5`, so it comes *after*. `(1,5,5)` ties on `c=5` with `(5,5,5)`, so the tie-break (`b ASC`) applies — both have `b=5`, so it's still a tie, and the ordering between them is unspecified (not "necessarily precedes").

Summary

You should now be able to create, alter and drop relations, enforce integrity constraints, and perform basic insert/update/delete/select operations in SQL. Next lecture: aggregation,

grouping, and querying across multiple relations. See week6-tutorial-applied-class-5-basic-sql-ddl-and-dml[1] and week6-tutorial-case-study-4-easydrive-insurance[2] for practice.

Applied Class 5: Basic SQL syntax, DDL and DML

Practice for 2026-03-30-basic-sql-ddl-and-dml[3].

Schema — BestTechLtd

BestTechLtd have built a simplistic authorisation management system for secure access control within their organisational network. When a new employee joins, their personal details are stored in `Employee`. Administrative employees (a specific type of employee, recorded additionally in `AdministrativeEmployee`) can grant roles to employees through a role-granting process. Each `Role` comes with specific `Permissions` that determine which company websites/resources an employee holding that role can access.

```
Employee             [EmployeeID, FirstName, LastName, DOB, PasswordHash, PasswordSalt]
AdministrativeEmployee [EmployeeID, Level, Type]
Role                  [RoleID, Name, Description]
RoleGranting          [EmployeeID, RoleID, AdministrationID, Timestamp]
Permission            [WebsiteURI, RoleID, GrantType, Description]

AdministrativeEmployee.EmployeeID references Employee.EmployeeID
RoleGranting.EmployeeID references Employee.EmployeeID
RoleGranting.RoleID references Role.RoleID
RoleGranting.AdministrationID references AdministrativeEmployee.EmployeeID
Permission.RoleID references Role.RoleID
```

Section 1 — Basic SQL

Question 1

Return all information (all columns) relating to the Roles recorded in the database.

[1] courses/infs1200/week6-tutorial-applied-class-5-basic-sql-ddl-and-dml.pdf
[2] courses/infs1200/week6-tutorial-case-study-4-easydrive-insurance.pdf
[3] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf

Working

```
SELECT * FROM Role;
```

Question 2

Return the distinct first name and last name of all employees, in descending alphabetical order of their last name.

Working

```
SELECT DISTINCT FirstName, LastName
FROM   Employee
ORDER BY LastName DESC;
```

Question 3

Return all the Permission `websiteURI`s which grant edit permissions to commercial websites (websites ending in `.com`).

Working

```
SELECT WebsiteURI
FROM   Permission
WHERE  GrantType = 'Edit'
  AND  WebsiteURI LIKE '%.com';
```

Question 4 (Challenge)

Create an Employee code, which is the combination of the employee's first name, last name, and DOB year with syntax `[FirstName]-[Lastname]-[yyyy]`. Create the code only if a first name, last name, and Date of Birth is present.

Working

```
SELECT CONCAT(FirstName, '-', LastName, '-', YEAR(DOB))
FROM   Employee
WHERE  1 = 1
  AND  FirstName IS NOT NULL
  AND  LastName IS NOT NULL
  AND  DOB IS NOT NULL;
```

Section 2 — Data Definition Language (DDL)

Question 5

Before BestTechLtd created their authorisation system, employees were assigned only a single role, plus a unique access token granting access for that role. Create a new relation ***LegacyEmployee****, a specialised type of employee (stored in a separate relation), with:*

- ***LegacyEmployeeID****: a unique reference to the* ***EmployeeID*** *stored in the authorisation database.*
- ***GrantAccessToken****: a string of exactly 64 characters granting access to old websites.*
- ***RoleID****: a reference to the legacy employee's original role. This reference cannot be null.*

Working

```
CREATE TABLE LegacyEmployee (
    LegacyEmployeeID VARCHAR(8) PRIMARY KEY,
    GrantAccessToken CHAR(64) NOT NULL,
    RoleID           INT NOT NULL,
    FOREIGN KEY (LegacyEmployeeID) REFERENCES Employee(EmployeeID),
    FOREIGN KEY (RoleID) REFERENCES Role(RoleID)
);
```

`LegacyEmployee` is a specialisation of `Employee` (recall relational-mapping-notation[4]'s Rule 8b: a `1:1` relation with the subclass's primary key *also* being a foreign key referencing the superclass).

[4] courses/infs1200/relational-mapping-notation.pdf

Question 6 (Challenge)

BestTechLtd want to migrate password credentials from `Employee` *into a separate* `Credential` *relation. Create the new table (fields in any order), assuming there is no existing data, and remove the migrated fields from* `Employee`*.*

i Working

```
CREATE TABLE Credential (
    CredentialID VARCHAR(9) PRIMARY KEY,
    EmployeeID   VARCHAR(8) NOT NULL,
    PasswordHash VARCHAR(128) NOT NULL,
    PasswordSalt VARCHAR(64) NOT NULL,
    Timestamp    DATETIME,
    FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)
);

ALTER TABLE Employee
    DROP COLUMN PasswordHash,
    DROP COLUMN PasswordSalt;
```

Section 3 — Data Manipulation Language (DML)

Question 7

Revoke all roles granted to EmployeeID `E0008`*.*

i Working

```
DELETE FROM RoleGranting
WHERE  EmployeeID = 'E0008';
```

Question 8

Revoke the role(s) granted to John Stevens at any time during 22 July 2024. Assume such a role exists and only one John Stevens exists.

Working

```
DELETE FROM RoleGranting
WHERE  EmployeeID = (
    SELECT EmployeeID FROM Employee
    WHERE  FirstName = 'John' AND LastName = 'Stevens'
)
AND    Timestamp BETWEEN '2024-07-22 00:00:00' AND '2024-07-22 23:59:59';
```

Question 9 (Challenge)

A new employee `E0024`, James Moran, born 21 November 2001, with a given password hash and salt, is granted all the roles that employee Sofia Gonzalez has, granted by admin `E0001` today. Insert the new records (assume only one Sofia Gonzalez exists).

Working

```
INSERT INTO Employee (EmployeeID, FirstName, LastName, DOB, PasswordHash,
↪  PasswordSalt)
VALUES ('E0024', 'James', 'Moran', '2001-11-21',
        'e7cf3ef8d8aac2c1c93963e7a58b7b62ade24d0d0ba2c8ae0f7fb6c8b0aa0332',
        'gmAlpLW8cUj');

INSERT INTO RoleGranting
SELECT 'E0024', RoleID, 'E0001', Timestamp
FROM   RoleGranting
WHERE  EmployeeID = (
    SELECT EmployeeID
    FROM   Employee
    WHERE  FirstName = 'Sofia' AND LastName = 'Gonzalez'
);
```

Case Study 4: EasyDrive Insurance (DDL & DML)

Practice for 2026-03-30-basic-sql-ddl-and-dml[5]. This case study builds practical experience implementing a database in phpMyAdmin (a GUI for MySQL) — running real DDL/DML, and learning to break-test a schema and check edge cases.

[5]courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf

Correspondence 1

Sarah Chen, Head of Technology Operations at EasyDrive Insurance (a direct-to-consumer car insurer), emails asking for help migrating their schema to MySQL/phpMyAdmin. When a customer signs up they create a profile (`Customer`, `Address`), then may purchase a `Policy` for a `Vehicle`, insuring it over consecutive years.

Relational schema:

```
Customer    [CustomerID, Name, DateOfBirth, Email, Occupation, AddressID]
Address     [AddressID, StreetName, Number, Suburb, Postcode, State, Country]
Vehicle     [VehicleID, VehicleCode, VehiclePurpose, EstYearlyKm]
VehicleType [VehicleCode, Make, Model, Year, MarketValue]
Policy      [PolicyID, CustomerID, VehicleID, PolicyStartYear, PolicyPurchaseDate, Excess, Pr

Customer.AddressID references Address.AddressID
Policy.VehicleID references Vehicle.VehicleID
Policy.CustomerID references Customer.CustomerID
Vehicle.VehicleCode references VehicleType.VehicleCode
```

Data types (as specified by Sarah):

- `Address`: `AddressID` (4 chars), `StreetName` (255), `Number` (10, to allow e.g. `"4/12"`), `Suburb` (255), `Postcode` (10), `State` (10), `Country` (50).
- `Customer`: `CustomerID` (integer), `Name` (255), `DateOfBirth` (date), `Email` (255), `Occupation` (255), `AddressID` (4).
- `VehicleType`: a reference table mapping a `VehicleCode` (50) to `Make`/`Model`/`Year` — the combination of `Make`, `Model`, `Year` must be unique, since each distinct vehicle configuration maps to exactly one code (used to determine `MarketValue`, the accident payout).
- `Vehicle`: `VehicleID` (6), `VehicleCode` (references `VehicleType`), `VehiclePurpose` (only `'Private'` or `'Business'`), `EstYearlyKm` (integer).
- `Policy`: `PolicyID` (6), `CustomerID` (integer), `VehicleID` (6), `PolicyStartYear` (integer), `PolicyPurchaseDate` (date), `Excess` (integer), `Premium` (decimal, 2 dp, 10 digits total).

Section A — Data Definition Language

Using phpMyAdmin, create a new database ***`EasyDrive`*** *and implement the five tables above, with all data types/keys/constraints as described.*

Working

```sql
CREATE DATABASE EasyDriveInsurance;
USE EasyDriveInsurance;

CREATE TABLE Address (
    AddressID  VARCHAR(4),
    StreetName VARCHAR(255),
    Number     VARCHAR(10),
    Suburb     VARCHAR(255),
    Postcode   VARCHAR(10),
    State      VARCHAR(10),
    Country    VARCHAR(50),
    PRIMARY KEY (AddressID)
);

CREATE TABLE Customer (
    CustomerID  INT,
    Name        VARCHAR(255),
    DateOfBirth DATE,
    Email       VARCHAR(255),
    Occupation  VARCHAR(255),
    AddressID   VARCHAR(4),
    PRIMARY KEY (CustomerID),
    FOREIGN KEY (AddressID) REFERENCES Address(AddressID)
);

CREATE TABLE VehicleType (
    VehicleCode VARCHAR(50),
    Make        VARCHAR(50),
    Model       VARCHAR(50),
    Year        INT,
    PRIMARY KEY (VehicleCode),
    UNIQUE (Make, Model, Year)
);

CREATE TABLE Vehicle (
    VehicleID      VARCHAR(6),
    VehicleCode    VARCHAR(50),
    VehiclePurpose ENUM('Private', 'Business'),
    EstYearlyKm    INT,
    PRIMARY KEY (VehicleID),
    FOREIGN KEY (VehicleCode) REFERENCES VehicleType(VehicleCode)
);

CREATE TABLE Policy (
    PolicyID           VARCHAR(6),
    CustomerID         INT,
    VehicleID          VARCHAR(6),
    PolicyStartYear    INT,
    PolicyPurchaseDate DATE,
    Excess             INT,
    Premium            DECIMAL(10, 2),
    PRIMARY KEY (PolicyID),
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID),
    FOREIGN KEY (VehicleID) REFERENCES Vehicle(VehicleID)
);
```

> Note: modern AI tools can draft `CREATE TABLE` statements from a schema description, but the draft must be checked carefully against the description before running it — AI-generated SQL can contain errors or omissions (see 2026-04-13-nested-queries-views-and-generative-ai[6]'s "Generative AI & SQL" section).

Section B — Analysis of Data Manipulation Language

Sarah asks the students to run five `INSERT` operations, in order, and record for each: whether it succeeded/failed, the error message (if any), and which integrity constraint was violated and why.

```
-- Operation 1
INSERT INTO Address (AddressID, StreetName, Number, Suburb, Postcode, State, Country)
VALUES ('A001', 'Main Street', '12', 'Brisbane', '4000', 'QLD', 'Australia');

-- Operation 2
INSERT INTO Customer (CustomerID, Name, DateOfBirth, Email, Occupation, AddressID)
VALUES (1001, 'Allan Smith', '2000-01-01', 'allan@uq.edu.au', 'Student', 'A999');

-- Operation 3
INSERT INTO Customer (CustomerID, Name, DateOfBirth, Email, Occupation, AddressID)
VALUES (1001, 'Rebecca Johnson', '1995-06-15', 'rebecca@gmail.com', 'Engineer',
 ↪  'A001');

-- Operation 4
INSERT INTO VehicleType (VehicleCode, Make, Model, Year)
VALUES ('VT001', 'Toyota', 'Corolla', 2020);
INSERT INTO Vehicle (VehicleID, VehicleCode, VehiclePurpose, EstYearlyKm)
VALUES ('V00001', 'VT001', 'Personal', 15000);

-- Operation 5
INSERT INTO Policy (PolicyID, CustomerID, VehicleID, PolicyStartYear,
 ↪  PolicyPurchaseDate, Excess, Premium)
VALUES ('P00001', 1001, 'V00001', 2024, '2024-03-01', 500, 1200.00);
```

[6] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf

Working

Op	Result	Error / constraint
1	Succeeded	—
2	Failed	`#1452 - Cannot add or update a child row: a foreign key constraint fails`. **Referential integrity** — `AddressID = 'A999'` doesn't exist in `Address`.
3	Succeeded	— (a *different* row, correctly referencing `A001`; note it reuses `CustomerID = 1001` from the failed Op 2, which is fine since Op 2 never committed)
4	Failed	`#1265 - Data truncated for column 'VehiclePurpose' at row 1`. **Domain constraint** — `'Personal'` is not a valid `VehiclePurpose` value (only `'Private'`/`'Business'`).
5	Failed	`#1452 - ... FOREIGN KEY (VehicleID) REFERENCES Vehicle(VehicleID)`. **Referential integrity** — since Op 4 failed, `VehicleID = 'V00001'` was never inserted into `Vehicle`.

This is a good illustration of *cascading* failure: a single domain violation in Op 4 also causes Op 5 to fail, even though Op 5's own values look superficially fine.

Correspondence 2 (Challenge)

Sarah's team requests further changes now the core schema is in place:

1. A new `InsuranceClaim` table: an auto-incrementing claim ID; a reference to the `Policy` the claim is against (claims should be automatically deleted if their policy is deleted); the incident date/time; the claimed amount (2 dp decimal); a status restricted to `'Pending'`, `'Approved'`, or `'Rejected'`; and a description (255 chars).
2. Remove the `Occupation` column from `Customer` (no longer collected).
3. Add two constraints to `Customer`: a named `UniqueEmail` uniqueness constraint on `Email`, and a check constraint requiring customers to be at least 18 years old (based on `DateOfBirth`).

Task 4 — Creating InsuranceClaim

Working

```
CREATE TABLE InsuranceClaim (
    ClaimID          INT AUTO_INCREMENT,
    PolicyID         VARCHAR(6),
    ClaimDate        DATE,
    ClaimAmount      DECIMAL(10, 2),
    ClaimStatus      ENUM('Pending', 'Approved', 'Rejected'),
    ClaimDescription VARCHAR(255),
    PRIMARY KEY (ClaimID),
    FOREIGN KEY (PolicyID) REFERENCES Policy(PolicyID) ON DELETE CASCADE
);
```

Task 5 — Removing Occupation

Working

```
ALTER TABLE Customer DROP COLUMN Occupation;
```

Task 6 — Fraud prevention constraints

Working

```
ALTER TABLE Customer
    ADD CONSTRAINT UniqueEmail UNIQUE (Email);

ALTER TABLE Customer
    ADD CONSTRAINT AgeOver CHECK (
        TIMESTAMPDIFF(YEAR, DateOfBirth, CURDATE()) >= 18
    );
```

Reference material

Relational Mapping Notation

Reference legend + course style guide for writing relational schemas — introduced across 2026-03-09-the-relational-model-and-integrity-constraints[7] and 2026-03-16-er-to-relational-mapping[8].

Schema notation

- `Relation [attr1, attr2, ...]` — a relation schema; the (primary) key attribute(s) are underlined.
- A composite primary key gets a single continuous underline spanning all of its attributes, e.g. `Enrolment [studentId, courseCode, sem, year]`.
- `Relation.fk references OtherRelation.pk` — a foreign key constraint, listed directly under the relation it belongs to.
- `Relation.{fk1, fk2} references OtherRelation.{pk1, pk2}` — a composite foreign key referencing a composite primary key.

Naming convention (INFS1200/7900 style guide)

- **Table names**: UpperCamelCase (first letter of each word capitalised, no spaces) — e.g. `Flight`, not `FLIGHT`.
- **Attribute names**: lowerCamelCase (first letter of each word from the *second* word onwards capitalised) — e.g. `departureTime`, not `Departure Time`. Acronym attribute names stay entirely lowercase (e.g. `eta`, not `ETA`).

[7] courses/infs1200/2026-03-09-the-relational-model-and-integrity-constraints.pdf
[8] courses/infs1200/2026-03-16-er-to-relational-mapping.pdf

- A space separates the table name from the opening bracket: `Flight [planeNumber, ...]`, not `Flight[planeNumber, ...]`.

Layout convention

A table's foreign key constraint lines go directly underneath that table's own definition, with a blank line before the next table starts — not all grouped together under a separate "Foreign Keys:" heading at the end:

```
Employee [ssn, firstName, lastName, dob, manager]
Employee.manager references Employee.ssn

Dependant [ssn, name, dob]
Dependant.ssn references Employee.ssn

DependantPhoneNumber [ssn, name, phoneNumber]
DependantPhoneNumber.{ssn, name} references Dependant.{ssn, name}
```