INFS1200 — Week 6 Notes

Table of contents

Basic SQL Syntax: Data Definition and Data Manipulation Language **2**
- Where we are . . . 2
- Example schemas used this module . . . 2
- The three (main) types of SQL statement . . . 3
- Data Definition Language (DDL) . . . 3
- Question 1 — DDL constraints . . . 5
- Question 2 — Designing with referential integrity . . . 6
- Question 3 — Implementing referential integrity . . . 7
- Data Manipulation Language (DML) . . . 8
- Question 4 — Correct use of DELETE . . . 10
- Question 5 — SQL projection . . . 11
- Question 6 — SQL projection with DISTINCT . . . 11
- Question 7 — Selection . . . 12
- Question 8 — Sorting . . . 13
- Summary . . . 14

Applied Class 5: Basic SQL syntax, DDL and DML **14**
- Schema — BestTechLtd . . . 14
- Section 1 — Basic SQL . . . 15
- Section 2 — Data Definition Language (DDL) . . . 16
- Section 3 — Data Manipulation Language (DML) . . . 17

Case Study 4: EasyDrive Insurance (DDL & DML) **18**
- Correspondence 1 . . . 19
- Section A — Data Definition Language . . . 19
- Section B — Analysis of Data Manipulation Language . . . 22
- Correspondence 2 (Challenge) . . . 24

Reference material **25**
- Relational Mapping Notation . . . 25

SQL Data Definition Language (DDL) 26
SQL Data Manipulation: INSERT, UPDATE, DELETE 33
NULLs in SQL 37
The SELECT Statement and Its Clause Order 41
Update Operations and Constraint Violations 46

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));
```

i 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.

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.

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).

```sql
-- 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.

```sql
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

```sql
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.

```sql
-- 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

```sql
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.

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

UPDATE

```sql
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).

```sql
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`

> **i** 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;
```

```sql
-- 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`.

```sql
-- 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];
```

```
-- 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

```
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`.

```
-- 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 `Permission`s 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
```

[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

Section 1 — Basic SQL

Question 1

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

> **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]. These notations are specific to INFS1200/7900 and may differ from textbooks and online resources, so they are what to follow in an assignment or exam. The guide covers *naming and layout only*: it recommends nothing about which relations to produce. The mapping steps themselves are in er-to-relational-mapping[9].

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.

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

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

The guide's compliant and non-compliant pair puts every rule in one line each:

```
Flight [planeNumber, departureTime, duration, eta]

FLIGHT[planeNumber, departure time, Duration, ETA]
```

The second breaks five rules at once — shouted table name, no space before the bracket, a space inside an attribute name, a capitalised attribute, and an upper-case acronym.

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}
```

SQL Data Definition Language (DDL)

DDL is the subset of SQL that defines and changes the database *schema* — `CREATE`, `ALTER`, `DROP` — as opposed to the data held in it. It is where the relational schema produced by er-to-relational-mapping[10] (written in relational-mapping-notation[11]) becomes a real set of

[10] courses/infs1200/er-to-relational-mapping.pdf
[11] courses/infs1200/relational-mapping-notation.pdf

tables, and where the integrity constraints of the relational model are actually declared and enforced. Introduced in 2026-03-30-basic-sql-ddl-and-dml[12]; practised in week6-tutorial-applied-class-5-basic-sql-ddl-and-dml[13] and week6-tutorial-case-study-4-easydrive-insurance[14].

Where DDL sits among SQL statement types

Family	Purpose	Statements
DDL (Data Definition Language)	Define/change the *schema*	`CREATE`, `ALTER`, `DROP`
DML (Data Manipulation Language)	Manipulate *data*	`INSERT`, `UPDATE`, `DELETE`, `SELECT`
DCL (Data Control Language)	Transaction control, semantic integrity (triggers/assertions), authorisation and privileges, role-based security, physical storage parameters	`GRANT` / `REVOKE`, see database-access-control[15]

See sql-dml[16] for the data-modification statements and sql-select-statement[17] for querying.

Syntax notation used in this course

Statement templates throughout Module 3 are written with a fixed convention:

Written	Means
`KEYWORD`	A literal SQL keyword
`<argument>`	A name or value you supply
`[optional]`	May be omitted
`{multiple}`	May be repeated
`…|..choice..|…`	Pick exactly one of the alternatives

[12] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf
[13] courses/infs1200/week6-tutorial-applied-class-5-basic-sql-ddl-and-dml.pdf
[14] courses/infs1200/week6-tutorial-case-study-4-easydrive-insurance.pdf
[15] courses/infs1200/database-access-control.pdf
[16] courses/infs1200/sql-dml.pdf
[17] courses/infs1200/sql-select-statement.pdf

CREATE TABLE

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

Domain constraints are specified **per attribute**; key, entity and referential integrity constraints are specified **after** all the attributes are declared.

An attribute's data type can be given directly on the column, or by declaring a named domain with `CREATE DOMAIN` and using that instead. The course names `CREATE DOMAIN` as the alternative but never gives its syntax or an example, so the only form actually examinable here is the direct one — a type (and optionally a `CHECK`) written on the column.

An entity table — `Item [itemID, category, colour]`:

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

A relationship table — `Sale [itemID, custID, timestamp, price, salesPerson]`, a ternary-degree relation between `Item`, `Customer` and `Employee`, whose primary key is composite and whose foreign keys carry the relationship:

```
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));
```

Column types used in this course

Type	Used for
`INTEGER` / `INT`	Whole numbers (IDs, counts, years)
`CHAR(n)`	Fixed-length strings — e.g. a 64-character access token, a 9-character SSN
`VARCHAR(n)`	Variable-length strings up to `n` characters
`ENUM('a', 'b', …)`	A domain restricted to a fixed list of literal values
`DECIMAL(p, s)` / `DOUBLE(p, s)`	Fixed-precision numbers — e.g. `DECIMAL(10, 2)` for money
`DATE`	A calendar date
`DATETIME` / `TIMESTAMP`	A date *and* time

`AUTO_INCREMENT` (MySQL) on an integer column makes the DBMS generate successive key values, e.g. `ClaimID INT AUTO_INCREMENT`.

Choosing `ENUM` (or a `CHECK`) over a plain string is what turns "should only ever be `'Private'` or `'Business'`" into a constraint the DBMS enforces — inserting `'Personal'` then fails with a *domain* violation rather than silently storing bad data.

Declaring constraints

Constraints are rules limiting what data can go into a table. See integrity-constraints[18] for what each one means in the relational model, and relational-keys[19]/foreign-keys[20] for the key concepts they enforce; this is how each is written in SQL:

Constraint	Meaning
`PRIMARY KEY`	Value is unique **and** not null
`FOREIGN KEY … REFERENCES`	Value must exist in the referenced (parent) table
`UNIQUE`	Value is unique **or null** — unlike a primary key
`CHECK`	Value(s) must satisfy a condition
`NOT NULL`	Value must be supplied

A table may declare only **one** `PRIMARY KEY` clause, but any number of `UNIQUE` clauses — so "`id` is the key, but `{firstName, lastName}` must also be unique" is a primary key plus a separate `UNIQUE` over the pair, not two primary keys:

[18] courses/infs1200/integrity-constraints.pdf
[19] courses/infs1200/relational-keys.pdf
[20] courses/infs1200/foreign-keys.pdf

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

`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

Prefixing a constraint with `CONSTRAINT <name>` has two benefits: error messages name the violated constraint, and the constraint can later be modified or removed by name.

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

ALTER TABLE Item DROP CONSTRAINT item_pk;
```

An unnamed constraint can't be dropped later — so name anything you may plausibly want to evolve.

Referential actions (ON DELETE / ON UPDATE)

Deleting a row other rows point at would leave those rows orphaned. The referential action attached to a `FOREIGN KEY` clause decides what happens instead:

Action	Effect
`ON DELETE RESTRICT` (default)	Disallows the deletion
`ON DELETE CASCADE`	Deletes the 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

These are the SQL spelling of the reject/cascade/set-null policies in update-operations-and-constraint-violations[21].

The same four options apply to updates, written `ON UPDATE …`. The difference is only in what fires them: `ON DELETE` responds to the referenced row being *deleted*, `ON UPDATE` to its referenced key value being *changed*. So on an update, `RESTRICT` refuses to change the parent's key while children still point at the old value, `CASCADE` rewrites the children's foreign key to the new value, `SET NULL` clears it, and `SET DEFAULT` substitutes the column's default. `RESTRICT` on delete with `CASCADE` on update is the common pairing, and the one the course's worked schema uses throughout — a key that is *renumbered* should carry its dependants along, while a parent that is *removed* should not silently destroy them.

```
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);
```

Direction matters, and only one direction exists. The action fires in the direction the foreign key *points*: from the referenced (parent) table to the referencing (child) table. Given

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

[21] courses/infs1200/update-operations-and-constraint-violations.pdf

deleting a `Staff` row cascades into `ParkingPermit`; deleting a `ParkingPermit` row has no defined effect on `Staff` whatsoever.

Choosing between the actions is a *design* decision about what the data should mean. For `Student.advisorID references Professor.professorID`, `ON DELETE SET NULL` says "a professor can leave without their students' records being destroyed — they simply have no advisor for now"; `RESTRICT` would instead block the professor's deletion until every advisee is reassigned, and `CASCADE` would delete the students.

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];
```

`ALTER TABLE` is the schema-evolution statement: add or drop columns, change a column definition, add or drop constraints. A constraint cannot be *altered* in place — drop it and re-add it (exact syntax varies between commercial products).

```
-- 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 named constraints after the fact
ALTER TABLE Sale ADD CONSTRAINT ChkAge CHECK (price BETWEEN 10 AND 10000);
ALTER TABLE Customer ADD CONSTRAINT UniqueEmail UNIQUE (Email);
ALTER TABLE Customer ADD CONSTRAINT AgeOver CHECK (
    TIMESTAMPDIFF(YEAR, DateOfBirth, CURDATE()) >= 18);

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

The `NOT NULL` restriction on a newly added column is worth remembering — see sql-nulls[22].

[22] courses/infs1200/sql-nulls.pdf

DROP TABLE

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

Dropping a table drops every constraint defined on it — *including constraints in other tables that reference it* — deletes all its tuples, and removes the table definition from the system catalog. `RESTRICT` refuses the drop while dependent objects (e.g. views) still exist; `CASCADE` drops them too (see sql-views[23]).

Cyclical foreign key dependencies

`Department.mgrSSN → Employee.ssn` and `Employee.dNum → Department.dNumber` reference each other, so neither table can be created with its foreign keys in place — the other table doesn't exist yet. The fix is to defer the constraints:

```
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)
```

The insertions sit *between* the `CREATE`s and the `ALTER`s for a reason: with both foreign keys already in place, whichever table you populated first would be referencing rows that don't exist yet.

SQL Data Manipulation: INSERT, UPDATE, DELETE

DML is the subset of SQL that manipulates *data* rather than the schema. A relational query language needs four operations — `INSERT` (new tuples), `DELETE` (remove tuples), `UPDATE` (change attribute values) and `SELECT` (retrieve data) — and SQL provides exactly those. This note covers the three that *modify* stored data; `SELECT` is large enough to have its own note, sql-select-statement[24]. Introduced in 2026-03-30-basic-sql-ddl-and-dml[25]; practised in week6-tutorial-applied-class-5-basic-sql-ddl-and-dml[26] and week6-tutorial-case-study-4-easydrive-insurance[27].

[23] courses/infs1200/sql-views.pdf
[24] courses/infs1200/sql-select-statement.pdf
[25] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf
[26] courses/infs1200/week6-tutorial-applied-class-5-basic-sql-ddl-and-dml.pdf
[27] courses/infs1200/week6-tutorial-case-study-4-easydrive-insurance.pdf

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 in the order of the explicit column list if one is given:

```
-- Positional: values in declaration order
INSERT INTO Customer VALUES
    ('653298653', 'Ronald West', 'Male', '1995-12-30');

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

A multi-tuple insert either comma-separates several value lists, or — much more powerfully — takes the result of a query in place of the `VALUES` clause. The query's result columns must line up with the target column list:

```
-- 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;
```

The query may itself be grouped, which turns `INSERT … SELECT` into a way of materialising a summary into a table (sql-aggregation-and-grouping[28]). One row goes in per group, and the target table's existing rows are left alone — the new rows are *appended*, so a `Science` row already sitting in `DegreeStatistics` survives untouched:

```
-- One row per degree, holding that degree's headcount
INSERT INTO DegreeStatistics (degree, num)
    SELECT   degree, COUNT(*)
    FROM     Students
    GROUP BY degree;
```

[28] courses/infs1200/sql-aggregation-and-grouping.pdf

Getting the aggregate wrong here is silent: `SUM(id)` in place of `COUNT(*)` inserts perfectly well-formed rows holding the wrong numbers.

The inserted rows can mix constants with values pulled from the query, which is how "give this new employee the same roles as Sofia Gonzalez" becomes a single statement:

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

DELETE

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

One `DELETE` removes zero, one, several, or *all* tuples from a single table — the `WHERE` clause is what stands between the two extremes, and omitting it empties the table. The condition may itself contain a subquery (see sql-nested-queries[29]):

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

-- Revoke the role(s) granted to John Stevens on 22 July 2024
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';
```

Because a deletion is only as precise as its predicate, "delete all the CompSci students *except* student 4" is

```
DELETE FROM Student WHERE degree = 'CompSci' AND NOT id = 4;
```

[29] courses/infs1200/sql-nested-queries.pdf

and this turns on **operator precedence**: `NOT` binds tighter than `AND`, so the condition reads `degree = 'CompSci' AND (NOT (id = 4))` — CompSci students other than #4 — rather than `NOT (degree = 'CompSci' AND id = 4)`, which would spare nobody but #4 and delete the whole rest of the table. See sql-select-statement[30] for the precedence order.

Two neighbouring formulations both fail. `WHERE id = 3 AND id = 5 AND id = 12` is a contradiction — one column cannot equal three values at once — and quietly deletes nothing at all; `id IN (3, 5, 12)` is the form that was meant. And listing by hand the rows you happened to notice (`fName = 'Diluen' OR fName = 'Peter' OR …`) is unsafe in the other direction: it catches rows that merely share a name without satisfying the condition you actually meant.

Deletion may **propagate** to other tables when `ON DELETE` referential-triggered actions were declared — see sql-ddl[31]'s referential actions section.

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, and the new value can be an expression over the row's existing values:

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

Updating a *primary key* value may propagate to other tables through foreign keys that reference it, according to the declared `ON UPDATE` action.

Modifications are where constraints bite

Every constraint declared in sql-ddl[32] is checked at modification time, so a failed `INSERT` is usually a precise diagnostic about the data rather than the SQL — see update-operations-and-constraint-violations[33] for which operation can break which constraint. Running a batch of inserts against the EasyDrive schema shows the typical failure modes:

[30] courses/infs1200/sql-select-statement.pdf
[31] courses/infs1200/sql-ddl.pdf
[32] courses/infs1200/sql-ddl.pdf
[33] courses/infs1200/update-operations-and-constraint-violations.pdf

Insert	Outcome
`Customer.AddressID = 'A999'` when no such `Address` row exists	Fails — **referential integrity**: `#1452 Cannot add or update a child row: a foreign key constraint fails`
`Vehicle.VehiclePurpose = 'Personal'` against `ENUM('Private','Business')`	Fails — **domain constraint**: `#1265 Data truncated for column 'VehiclePurpose'`
`Policy.VehicleID = 'V00001'` after that `Vehicle` insert failed	Fails — **referential integrity**, purely as a knock-on

Two things are worth taking from that. First, a rejected statement changes nothing, so a key value "used" by a failed insert is still free for the next one. Second, failures cascade through a sequence: the `Policy` row's own values look perfectly fine and it still fails, because the row it depends on was never created. When debugging a batch, fix the first failure before reading anything into the later ones.

See integrity-constraints[34] for the underlying entity/referential integrity rules these errors report.

NULLs in SQL

`NULL` is the special value a relational attribute takes when there is no ordinary value to store — it represents an **inapplicable, unknown or missing** value rather than zero or the empty string. Because a null is not a value in the attribute's domain, it needs its own handling everywhere it appears: in constraints, in `WHERE` conditions, and in the results a query produces. Defined in 2026-02-23-conceptual-database-design-and-the-er-model[35], constrained in 2026-03-09-the-relational-model-and-integrity-constraints[36], and handled in SQL from 2026-03-30-basic-sql-ddl-and-dml[37] onwards.

Which constraints permit a null

Declaration	Nulls allowed?
`PRIMARY KEY`	**No** — a primary key value is unique *and* not null; for a composite key, no part of it may be null (entity integrity)

[34] courses/infs1200/integrity-constraints.pdf
[35] courses/infs1200/2026-02-23-conceptual-database-design-and-the-er-model.pdf
[36] courses/infs1200/2026-03-09-the-relational-model-and-integrity-constraints.pdf
[37] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf

Declaration	Nulls allowed?
`FOREIGN KEY`	**Yes** — a foreign key value must either be null, or match an existing row of the referenced relation (referential integrity)
`UNIQUE`	**Yes** — the value must be unique *or* null; this is the one difference from a primary key
`NOT NULL`	**No** — a value must be supplied

See integrity-constraints[38] for the underlying entity and referential integrity rules, relational-keys[39] and foreign-keys[40] for what the keys themselves are, and sql-ddl[41] for how each is declared.

The nullable foreign key is what makes several mappings work at all: an employee with no manager, or a department with no manager assigned yet, is represented by a null foreign key rather than by a missing row. It is also why er-to-relational-mapping[42] prefers to hang a 1:1 foreign key off the *total*-participation side — the other side would be null for almost every row.

Testing for a null

`NULL` is tested with `IS`, never with `=`. `IS` sits alongside `LIKE`, `IN` and `BETWEEN` in the course's catalogue of things a `WHERE` search condition may contain (sql-select-statement[43]):

```
SELECT * FROM Employee WHERE dNum IS NULL;
SELECT * FROM Employee WHERE dNum IS NOT NULL;
```

This shows up inside subqueries as well as at the top level — here, restricting the result to employees whose department currently has a manager assigned:

```
SELECT name, dNum
FROM   Employee e
WHERE  EXISTS (
    SELECT * FROM Department d
    WHERE  d.dNumber = e.dNum AND d.mgrSSN IS NOT NULL
);
```

[38] courses/infs1200/integrity-constraints.pdf
[39] courses/infs1200/relational-keys.pdf
[40] courses/infs1200/foreign-keys.pdf
[41] courses/infs1200/sql-ddl.pdf
[42] courses/infs1200/er-to-relational-mapping.pdf
[43] courses/infs1200/sql-select-statement.pdf

An explicit `IS NOT NULL` is also the usual guard before grouping or building a derived value out of several columns, so that rows with missing components are excluded rather than silently contributing:

```sql
-- Last names shared by more than one employee, ignoring unknown names
SELECT   DISTINCT LastName FROM Employee
WHERE    LastName IS NOT NULL
GROUP BY LastName
HAVING   COUNT(FirstName) > 1;

-- Build FirstName-LastName-yyyy only where all three parts exist
SELECT CONCAT(FirstName, '-', LastName, '-', YEAR(DOB))
FROM   Employee
WHERE  FirstName IS NOT NULL
  AND  LastName  IS NOT NULL
  AND  DOB       IS NOT NULL;
```

The limits of what the course specifies

Module 3 treats nulls **operationally** rather than logically. It gives the `IS` operator, the constraint rules above, and the places a null can come from — and it stops there. The lectures do not develop a three-valued logic of `TRUE`/`FALSE`/`UNKNOWN`, do not state how an aggregate function treats the nulls inside a group, and do not cover what happens when a `NOT IN` subquery returns a null among its values. Those questions are simply not settled by the material, so don't reason from a rule the course never gave you.

What the course *does* teach is the defensive habit: write the `IS NOT NULL` guard explicitly wherever a null could reach a comparison, a grouping, or a derived value, rather than relying on the DBMS to do something sensible with it. Every example above follows that habit.

Where nulls come from

Not every null is data someone failed to enter — several are created by the schema or the query itself:

Source	Why
A genuinely inapplicable/unknown/missing value	The original meaning of a null

Source	Why
`ALTER TABLE … ADD <column>`	Every existing row gets a null in the new column — which is exactly why the new column cannot be declared `NOT NULL` (sql-ddl[44])
`ON DELETE SET NULL`	The referential action deliberately clears the link instead of deleting the dependent rows (update-operations-and-constraint-violations[45])
An **outer join**	Unmatched rows are padded with nulls in the columns of the table that had no match (sql-joins[46])
Access-control masking	A cell above a user's clearance is presented to them as null rather than withheld (database-access-control[47])

The last two matter when reading results: a null in the output of a `LEFT JOIN` means "no matching row existed", which is information about the *relationship*, not about the row's stored data.

```
-- Departments with no manager still appear, with NULL for the name
SELECT    D.dName, E.name
FROM      Department AS D
LEFT JOIN Employee AS E ON D.mgrSSN = E.ssn;
```

Nulls as a design smell

A schema that produces many nulls is usually telling you something. Reducing the number of null values in tuples is guideline 3 of the informal design guidelines behind normalisation (update-anomalies[48]) — an attribute that is null for most rows of a relation often belongs in a separate relation instead. See 2026-04-27-database-design-guidelines-and-functional-dependencies[49] and normal-forms[50].

[44] courses/infs1200/sql-ddl.pdf
[45] courses/infs1200/update-operations-and-constraint-violations.pdf
[46] courses/infs1200/sql-joins.pdf
[47] courses/infs1200/database-access-control.pdf
[48] courses/infs1200/update-anomalies.pdf
[49] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[50] courses/infs1200/normal-forms.pdf

The SELECT Statement and Its Clause Order

`SELECT` is the retrieval half of SQL, and the reason SQL is called a **declarative** language: 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, which is what makes queries composable — it can be joined (sql-joins[51]), nested inside another query (sql-nested-queries[52]), stacked with another result set (sql-set-operations[53]), or stored as a view (sql-views[54]). Introduced in 2026-03-30-basic-sql-ddl-and-dml[55] and extended across 2026-04-06-aggregation-grouping-and-multiple-relation-queries[56] and 2026-04-13-nested-queries-views-and-generative-ai[57].

The full clause skeleton

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

Only `SELECT` and `FROM` are compulsory; the clauses must appear in this order when present.

Logical evaluation order

The order the clauses are *written* is not the order they take effect. Reading a query as a pipeline — each stage handing a table to the next — explains most of SQL's otherwise arbitrary-looking rules:

#	Clause	Operates on	Produces
1	`FROM`	The named tables (joined, if more than one)	One combined table
2	`WHERE`	Individual **rows**	The rows that satisfy the condition

[51] courses/infs1200/sql-joins.pdf
[52] courses/infs1200/sql-nested-queries.pdf
[53] courses/infs1200/sql-set-operations.pdf
[54] courses/infs1200/sql-views.pdf
[55] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf
[56] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[57] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf

#	Clause	Operates on	Produces
3	`GROUP BY`	The surviving rows	One **group** per distinct combination of grouping attributes
4	`HAVING`	Whole **groups**	The groups that satisfy the condition
5	`SELECT`	Each group (or each row, if no grouping)	The output columns
6	`ORDER BY`	The finished result table	The same rows, sorted

Three of the course's recurring rules fall straight out of this:

- **An aggregate cannot appear in `WHERE`.** At stage 2 no groups exist yet, so there is nothing to aggregate over — `WHERE salary > AVG(salary)` fails with *invalid use of group function*, and the fix is to compute the aggregate in a subquery instead (see sql-nested-queries[58]).
- **`HAVING` may use aggregates, `WHERE` may not**, because `HAVING` runs at stage 4, after the groups exist.
- **`WHERE` filters rows, `HAVING` filters groups.** They are not interchangeable, and a query can sensibly use both — see sql-aggregation-and-grouping[59].

The placement of `SELECT` before `ORDER BY` is not a guess: a sort key may be given as a *column position in the target list* (`ORDER BY 2 DESC`), which is only meaningful if the target list has already been built. Correspondingly, the two clauses that must be able to see raw rows — `WHERE` and `GROUP BY` — can only name attributes of the tables in `FROM`, never an alias invented by `SELECT`.

Projection and selection

The course names the two halves of a simple query after the relational operations they perform:

Clause	Operation	Cuts the table
`SELECT`	**Projection** — choose the attributes	**Vertically**
`WHERE`	**Selection** — choose the tuples	**Horizontally**

Note the terminological trap: the relational operation called *selection* is the `WHERE` clause, **not** the `SELECT` clause.

[58] courses/infs1200/sql-nested-queries.pdf
[59] courses/infs1200/sql-aggregation-and-grouping.pdf

Projection, duplicates and bag semantics

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

SQL relations are **bags** (multisets), not sets: projection does **not** eliminate duplicates by default, unlike the pure relational-model[60]. `DISTINCT` restores set semantics; `*` is the wildcard for "all columns".

```
-- All the years a movie was produced, with and without duplicates
SELECT year FROM Movie;
SELECT DISTINCT year FROM Movie;
```

The consequence to watch for: projecting away the columns that made rows distinct leaves *duplicate* rows behind. Projecting `(score1, score2)` from a table of match results keeps two separate `(5, 3)` rows if two different fixtures both ended 5–3; `SELECT DISTINCT` collapses them into one and the row count drops accordingly.

Expressions and aliases

The target list can contain arithmetic (`+ - * /`) over numeric attributes, string/date functions, and aggregates — each optionally renamed 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;

-- Build an employee code FirstName-LastName-yyyy
SELECT CONCAT(FirstName, '-', LastName, '-', YEAR(DOB))
FROM   Employee;
```

Aliases also disambiguate attributes across tables and enable self-joins — see sql-joins[61].

[60] courses/infs1200/relational-model.pdf
[61] courses/infs1200/sql-joins.pdf

The WHERE clause

The `<search condition>` is a conditional (Boolean) expression identifying the tuples to retrieve. It may contain:

Form	Use
`=`, `<>`, `<`, `<=`, `>`, `>=`	Compare an attribute to a value or another attribute
`AND`, `OR`, `NOT`	Combine conditions — bracket them, the grouping is the whole meaning
`LIKE`	Substring matching: `%` is zero or more arbitrary characters, `_` is exactly one
`IN`	Membership in a list: `lastName IN ('Jones', 'Wong', 'Harrison')`
`IS`	Null-checking — `dNum IS NULL`; `=` does **not** work here, see sql-nulls[62]
`+ - * /`, date/time functions	Arithmetic on the row's own values: `salary * 2 > 50000`, `YEAR(GETDATE() - dob) > 55`
`BETWEEN … AND …`	Inclusive range: `salary BETWEEN 10000 AND 30000`

```sql
-- 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);

-- Permissions to edit commercial (.com) websites
SELECT WebsiteURI
FROM   Permission
WHERE  GrantType = 'Edit' AND WebsiteURI LIKE '%.com';
```

A disjunction of conjunctions is exactly as literal as it looks: a row qualifies only if it satisfies *one whole bracket*. Rows with impressive values in the right columns but the wrong combination — 8 runs scored but 12 conceded, against `(runsFor >= 6 AND runsAgainst <= 4)` — do not qualify.

Dates are compared either as a quoted `'YYYY-MM-DD'` literal or as the bare number in that order; both forms appear in the course and both work, because the ordering is the same either way:

[62] courses/infs1200/sql-nulls.pdf

```
SELECT name, dob FROM Employee WHERE dob < '1965-01-01';
SELECT name, dob FROM Employee WHERE dob < 19650101;
```

Grouping the connectives

The course brackets its compound conditions rather than leaning on precedence, and that is the habit to copy: `(A AND B) OR (C AND D)` written out is unambiguous to you and to the marker. Where precedence still decides the meaning, it is **`NOT` first, then `AND`, then `OR`** — so `NOT` binds to the single comparison next to it, not to the rest of the condition. sql-dml[63] works that case through on a `DELETE`, where misreading it costs the whole table rather than a few rows.

Written without brackets, `A AND B OR C AND D` groups as `(A AND B) OR (C AND D)` — which is usually what was meant, but relying on that is how a condition ends up meaning something you did not intend. Bracket it.

ORDER BY

```
[ORDER BY column [ASC|DESC] {, column [ASC|DESC]}]
```

Sorting is the last stage, applied to the finished result. A sort key is either a **column name** or a **column position in the target list**; multiple keys are applied left to right, each with its own `ASC` (default) or `DESC`.

```
-- By department ascending, then salary descending within each department
SELECT dNum, name, salary
FROM   Employee
ORDER BY dNum ASC, salary DESC;

-- Sorting a grouped result by its count, greatest first: the count is
-- the 2nd column of the target list
SELECT   VehiclePurpose, COUNT(*)
FROM     Vehicle
GROUP BY VehiclePurpose
ORDER BY 2 DESC;
```

The positional form is what makes a computed column sortable at all — the aggregate itself belongs to `SELECT` and `HAVING` (sql-aggregation-and-grouping[64]), and `ORDER BY` reaches it by pointing at the column it produced.

[63] courses/infs1200/sql-dml.pdf
[64] courses/infs1200/sql-aggregation-and-grouping.pdf

Later keys only break ties left by earlier ones, so `ORDER BY c DESC, b ASC` puts *every* row with a larger `c` ahead of `(5, 5, 5)` regardless of its `b`. Where two rows tie on *all* the sort keys, their relative order is unspecified — never rely on it.

Update Operations and Constraint Violations

There are three update operations on a relational database — **insert**, **delete**, and **modify** — and each can break a different subset of the integrity-constraints[65]. Knowing which subset turns "does this operation violate anything?" from a search into a checklist. Introduced in 2026-03-09-the-relational-model-and-integrity-constraints[66], drilled across week4-tutorial-applied-class-3-relational-integrity[67] and week6-tutorial-case-study-4-easydrive-insurance[68]. This is the model-level account; sql-dml[69] is the same three operations as SQL statements.

Which operation can violate what

	Domain	Key	Entity integrity	Referential integrity	Semantic
Insert					
Delete					
Modify					

Insert can violate all five: it supplies fresh values for every attribute, so any of them can be out of domain, duplicate an existing key, leave a primary key component null, point at a nonexistent row, or break a business rule. Five candidate tuples for the same two-table schema, differing in one value each, show one of each:

Modification is the same picture: it can produce any of those five rows out of a legal one, so it inherits the whole column.

Delete is the interesting row. It supplies no values at all, so it cannot possibly produce an out-of-domain value, a duplicate key, or a null primary key. What it *can* do is remove a row that something else still references — a referential integrity violation on the *other* relation — or break a business rule. Both happen on the same two-table `Employee/Department` schema: deleting the department with `id = 2` violates **referential integrity**, because employees still sit in it, while deleting employee `1023` violates a **semantic** constraint if the rule "every department must have at least one employee" has been stated. This asymmetry is worth remembering: for a delete, the only question is who points at the row being deleted.

[65] courses/infs1200/integrity-constraints.pdf
[66] courses/infs1200/2026-03-09-the-relational-model-and-integrity-constraints.pdf
[67] courses/infs1200/week4-tutorial-applied-class-3-relational-integrity.pdf
[68] courses/infs1200/week6-tutorial-case-study-4-easydrive-insurance.pdf
[69] courses/infs1200/sql-dml.pdf

Employee

ID	Name	Salary	Department
1751	Paris Lane	60,000	2
4671	Anna Lee	70,000	1
1023	Ben Cho	70,000	4
2034	Jack Smith	40,000	1
2670	Grace Mills	50,000	2

Department

ID	Name	Manager
1	Marketing	4671
2	Development	1751
4	Human Resources	1023

LOL	**John Doe**	**70,000**	**4**	⊗	Domain constraints
4671	**John Doe**	**70,000**	**4**	⊗	Key constraints
NULL	**John Doe**	**70,000**	**4**	⊗	Entity Integrity constraints
1111	**John Doe**	**70,000**	**6**	⊗	Referential Integrity constraints
1111	**John Doe**	**99,999**	**4**	⊗	Semantic Integrity constraints

Figure 1: One insert, five ways to break it. Each candidate row differs from a legal `Employee` tuple in exactly one value, and each trips a different constraint: `LOL` is not a four-digit integer (domain), `4671` already exists (key), a null `ID` is a null primary key (entity integrity), department `6` is not in `Department` (referential integrity), and `99,999` exceeds the supervisor's salary, which the business rule forbids (semantic). The crimson outline plus ⊗ badge is the course's notation for an illegal tuple.

Modify behaves like an insert restricted to the columns it touches: changing a key column risks a key violation, changing a foreign key risks a referential one, and changing a primary key component to `NULL` risks entity integrity.

Handling a referential violation on delete

Rejecting the operation isn't the only option. At the relational-model stage there are exactly three outcomes for a referential integrity violation — it can be:

Policy	Effect
Rejected	Refuse the delete; the referencing rows keep the referenced row alive
Cascaded	Delete the dependent rows too
Modified to a default value	Keep the dependent rows, but reset the referencing attribute's value

That three-way choice is what is examinable *here*; the finer-grained named actions — `RESTRICT` (the default), `CASCADE`, `SET NULL` and `SET DEFAULT`, each available on both delete and update — arrive with SQL DDL and are examined there, not at the model level. See sql-ddl[70] for their exact semantics and syntax, and foreign-keys[71] for the constraint itself.

The choice is a property of the foreign key, decided at design time from what the dependent rows *mean* without the row they reference: an insurance claim makes no sense without its policy, so EasyDrive declares its claims `ON DELETE CASCADE`.

Worked diagnoses

Against the company schema `Employee [ssn, …, superSSN, dNumber]`, `Department [dName, dNumber, mgrSSN, mgrStartDate]`, `DeptLocations [dNumber, dLocation]`, `Project [pName, pNumber, pLocation, dNum]`, `WorksOn [essn, pNo, hours]`, `Dependent [essn, depName, …]`, plus two business rules — an employee's salary cannot exceed their supervisor's, and a department cannot have more than three active projects:

Operation	Violates
Insert an employee with a fresh `ssn`, a real supervisor, salary below the supervisor's	**Nothing** — a well-formed insert
Insert `('ProductA', 'A', 'Bellaire', 2)` into `Project` when no department `2` exists	Referential integrity **and** domain (`pNumber` is an integer, not `'A'`)

[70] courses/infs1200/sql-ddl.pdf
[71] courses/infs1200/foreign-keys.pdf

Operation	Violates
Insert a department whose `dNumber` already exists and whose `mgrSSN` doesn't	Key **and** referential integrity
Insert `(677678989, null, 40.0)` into `WorksOn`, `essn` not an employee	Entity integrity (`pNo` is part of the key) **and** referential integrity
Insert an employee earning more than their supervisor	Semantic
Delete the `WorksOn` rows for one employee	**Nothing** — nothing references `WorksOn`, and no rule requires an employee to have assignments
Delete an `Employee` row	Referential integrity — `WorksOn`, `Dependent`, `Department` and `Employee` itself may all reference that `ssn`
Modify a department's `mgrSSN` to another existing employee	**Nothing**, absent a stated rule about who may manage a department
Modify an employee's `superSSN` to an `ssn` that doesn't exist	Referential integrity
Insert a fourth active project for a department	Semantic

Two habits fall out of this. First, always check whether the row being deleted is *referenced*, not just whether it exists. Second, "no violation" is a legitimate answer — several of the operations above are perfectly legal, and the temptation is to invent a rule that forbids them.

Failures cascade

A rejected operation changes nothing, which means later operations that assumed it succeeded fail too. In the EasyDrive case study, inserting a `Vehicle` with `VehiclePurpose = 'Personal'` fails on a domain constraint (the column is `ENUM('Private', 'Business')`); the *next* insert, a `Policy` referencing that vehicle, then fails on referential integrity even though its own values look fine. When a sequence of operations is being analysed, each one has to be judged against the state the previous ones actually left behind — not the state they were intended to leave.

The transaction concept

A **transaction** is an executing program containing database operations — reads, inserts, deletes, updates. The consistency requirement applies at the *end* of a transaction, not between its individual steps: constraint violations are permitted at intermediate points, as long as the final state satisfies every constraint specified on the schema.

A transaction is a walk between two consistent states. It starts from consistent state 1, and each `INSERT`, `UPDATE` or `DELETE` moves the database through **intermediate states** that need

not satisfy the constraints; a `COMMIT` ends the walk at consistent state 2, while a `ROLLBACK` — or any error or failure along the way — returns the database to consistent state 1 as though nothing had happened. Either way the database is never *left* inconsistent.

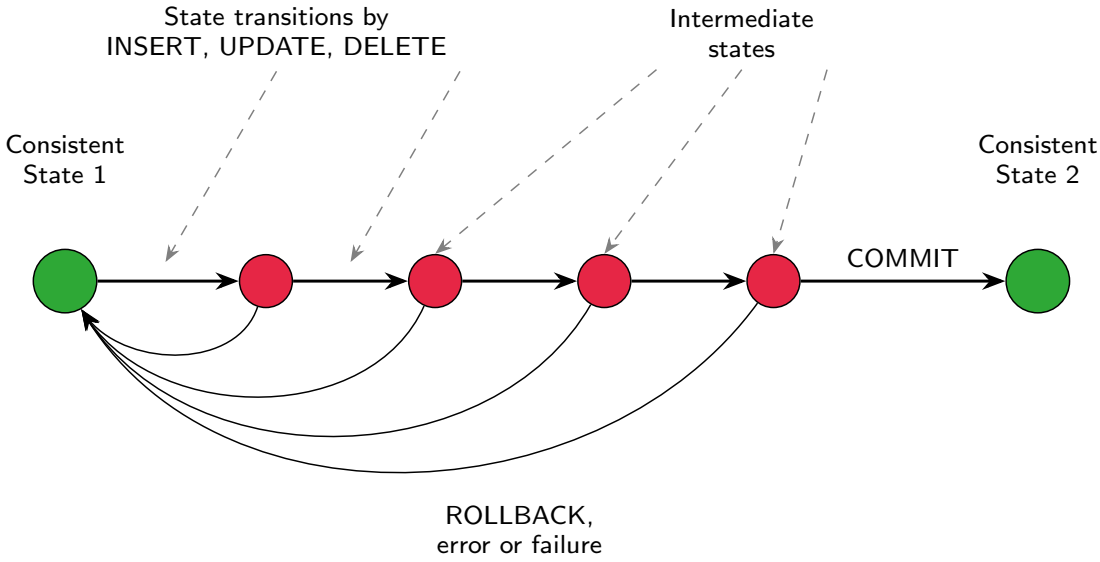

Figure 2: The transaction as a walk between consistent states. Green = a state satisfying every constraint, red = an intermediate state that need not. Only `COMMIT` and `ROLLBACK` end the walk, and both end it on a green state. The course has no house notation for transaction diagrams, so the shapes and colours here are not a convention to reproduce in an answer.

This is what makes mutually-dependent business rules satisfiable at all. Given "every department must have at least one employee" and "every employee must work for a department", neither a new department (no employees yet) nor a new employee (no department yet) can be inserted on its own without breaking a rule. A single transaction that inserts the department *and* its first employee resolves the deadlock: the intermediate state after the first insert is inconsistent, but the final state after both inserts is valid, and that is the state the constraints are judged against.