Basic SQL Syntax: Data Definition and Data Manipulation Language

Table of contents

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));
```

```sql
-- 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.

```sql
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`

> **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;

-- 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];
```

```
-- 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.

```sql
-- 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.

[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