SQL Data Definition Language (DDL)

Table of contents

Where DDL sits among SQL statement types . . . 1
Syntax notation used in this course . . . 2
CREATE TABLE . . . 2
Column types used in this course . . . 3
Declaring constraints . . . 4
Referential actions (ON DELETE / ON UPDATE) . . . 5
ALTER TABLE . . . 6
DROP TABLE . . . 7
Cyclical foreign key dependencies . . . 8

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[1] (written in relational-mapping-notation[2]) becomes a real set of 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[3]; practised in week6-tutorial-applied-class-5-basic-sql-ddl-and-dml[4] and week6-tutorial-case-study-4-easydrive-insurance[5].

Where DDL sits among SQL statement types

Family	Purpose	Statements
DDL (Data Definition Language)	Define/change the *schema*	`CREATE`, `ALTER`, `DROP`

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

Family	Purpose	Statements
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[6]

See sql-dml[7] for the data-modification statements and sql-select-statement[8] 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

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

[6] courses/infs1200/database-access-control.pdf
[7] courses/infs1200/sql-dml.pdf
[8] courses/infs1200/sql-select-statement.pdf

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[9] for what each one means in the relational model, and relational-keys[10]/foreign-keys[11] 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:

```
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),
```

[9] courses/infs1200/integrity-constraints.pdf
[10] courses/infs1200/relational-keys.pdf
[11] courses/infs1200/foreign-keys.pdf

```
    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[12].

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

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

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);
```

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[13].

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[14]).

[13] courses/infs1200/sql-nulls.pdf
[14] courses/infs1200/sql-views.pdf

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.