SQL Data Manipulation: INSERT, UPDATE, DELETE

Table of contents

INSERT 1
DELETE 3
UPDATE 4
Modifications are where constraints bite 4

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[1]. Introduced in 2026-03-30-basic-sql-ddl-and-dml[2]; practised in week6-tutorial-applied-class-5-basic-sql-ddl-and-dml[3] and week6-tutorial-case-study-4-easydrive-insurance[4].

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:

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

```sql
-- 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:

```sql
-- 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[5]). 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:

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

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:

```sql
INSERT INTO RoleGranting
SELECT 'E0024', RoleID, 'E0001', Timestamp
FROM   RoleGranting
WHERE  EmployeeID = (
```

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

```
    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[6]):

```
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;
```

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[7] 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

[6] courses/infs1200/sql-nested-queries.pdf
[7] courses/infs1200/sql-select-statement.pdf

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[8]'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[9] 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[10] for which operation can break which constraint. Running a batch of inserts against the EasyDrive schema shows the typical failure modes:

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'`

[8] courses/infs1200/sql-ddl.pdf
[9] courses/infs1200/sql-ddl.pdf
[10] courses/infs1200/update-operations-and-constraint-violations.pdf

Insert	Outcome
`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[11] for the underlying entity/referential integrity rules these errors report.

[11] courses/infs1200/integrity-constraints.pdf