Module 11 of 16

Modifying data

INSERT with column lists, many rows and INSERT … SELECT; UPDATE with expressions, CASE and values from another table; DELETE, TRUNCATE and DROP; MySQL upserts with ON DUPLICATE KEY UPDATE, REPLACE and INSERT IGNORE; and transactions with COMMIT, ROLLBACK and savepoints.

Everything before this module read data; this one changes it. Four statements do all of it — `INSERT` adds rows, `UPDATE` changes them, `DELETE` removes them, and the upsert forms decide between inserting and updating — and one mechanism, the transaction, makes several of them succeed or fail together. They are short statements with a large blast radius: an UPDATE without its WHERE rewrites every row, a TRUNCATE in MySQL cannot be rolled back, and a REPLACE quietly resets every column it was not given. The module teaches each with the habits that prevent those accidents, in MySQL 8 with InnoDB, and names the PostgreSQL spellings that interviews ask about.

INSERT covers the column list and why to always write it, multi-row values, defaults, AUTO_INCREMENT and `LAST_INSERT_ID()`, and `INSERT ... SELECT`. UPDATE covers SET and WHERE, computing from the old value, `CASE` for several rules in one pass, values from another table (correlated subqueries, MySQL's `UPDATE ... JOIN`, PostgreSQL's `UPDATE ... FROM`) and MySQL's left-to-right SET quirk. DELETE covers subqueries and `NOT EXISTS`, multi-table DELETE, `ON DELETE` actions, the DELETE vs TRUNCATE vs DROP comparison, and soft deletes. Upserts cover `ON DUPLICATE KEY UPDATE`, `REPLACE INTO`, `INSERT IGNORE`, PostgreSQL's `ON CONFLICT`, `MERGE`, and the portable update-then-insert. Transactions cover `START TRANSACTION`, `COMMIT`, `ROLLBACK`, autocommit, savepoints, errors and MySQL's implicit commits.

Every exercise is a script that changes a small, realistic dataset and ends with the SELECT that shows the result: stocking new products, archiving last year's orders, a seasonal repricing, applying a supplier's price list, purging expired sessions, removing orphaned orders, loading a product feed, counting page views, a bank transfer and a savepoint that cancels part of a transaction. The checkpoint adds a deduplicating import, a month-end close in one transaction and a stock sync from a delivery.

CONTENTS

1 INSERT — adding rows 14 min

2	UPDATE — changing rows	14 min
3	DELETE, TRUNCATE and DROP	14 min
4	Upserts — insert or update	15 min
5	Transactions in practice	14 min
✓	Checkpoint — Modifying data	25 min

Exercises in this module compile on MySQL 8 syntax on SQLite 3.49. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

INSERT — adding rows

Everything in the earlier modules read data. From here on the statements change it, and the first one is `INSERT`: it adds new rows to a table. This lesson covers the single-row and multi-row forms, the column list and why it matters, what happens to the columns you leave out (defaults, NULL and `AUTO_INCREMENT`), how to read back a generated id, and `INSERT INTO ... SELECT`, which inserts the result of a query. The examples use MySQL 8 with InnoDB tables; the differences that come up in PostgreSQL interviews are called out where they matter.

The basic form

```
INSERT INTO products (id, name, category, price, in_stock)
VALUES (101, 'Desk lamp', 'lighting', 24.99, 12);
```

The statement names the table, then a **column list** in parentheses, then `VALUES` with one value per listed column, in the same order. Strings and dates are quoted with single quotes (`'2024-03-01'`), numbers are not, and `NULL` is written bare. A single quote inside a string is doubled: `'O''Brien'`.

You may leave out the column list:

```
INSERT INTO products VALUES (102, 'Floor lamp', 'lighting', 79.00, 4);
```

This only works when you supply a value for **every** column in the order the table declares them, and it is a habit to avoid. The day someone adds a column, the statement either fails ("Column count doesn't match value count") or quietly puts values into the wrong columns if the types happen to fit. Always write the column list; it doubles as documentation.

Many rows in one statement

```sql
INSERT INTO products (id, name, category, price, in_stock)
VALUES
  (103, 'Bulb 9W',   'lighting', 3.50, 200),
  (104, 'Bulb 12W',  'lighting', 4.25, 150),
  (105, 'Extension', 'power',   15.00,  30);
```

Several value lists separated by commas insert several rows. It is faster than one statement per row (one parse, one round trip, one log flush under autocommit) and it is atomic: if row three violates a constraint, none of the three are inserted. Bulk loads of thousands of rows are usually batched this way, a few hundred or a thousand rows per statement.

The columns you leave out

A column missing from the column list gets, in order of preference:

1. its `DEFAULT` value, if the table declares one;
2. the next number, if it is an `AUTO_INCREMENT` column;
3. `NULL`, if the column allows it;
4. otherwise an error in MySQL's default strict mode ("Field 'x' doesn't have a default value").

```sql
CREATE TABLE tickets (
  id       INT AUTO_INCREMENT PRIMARY KEY,
  subject  VARCHAR(100) NOT NULL,
  status   VARCHAR(10)  NOT NULL DEFAULT 'open',
  assignee VARCHAR(40)
);

INSERT INTO tickets (subject) VALUES ('Printer jams'), ('VPN drops');
SELECT id, subject, status, assignee FROM tickets ORDER BY id;
```

id	subject	status	assignee
1	Printer jams	open	NULL
2	VPN drops	open	NULL

The `id` came from `AUTO_INCREMENT`, `status` from its default, `assignee` is NULL. Writing `NULL` explicitly for a column with a default does **not** give you the default — you asked for NULL, and a `NOT NULL` column refuses it. To ask for the default by name, MySQL accepts the

keyword `DEFAULT` in a value list (`VALUES (DEFAULT, 'x')`); leaving the column out of the list is the portable way.

Generated ids and LAST_INSERT_ID

When the database numbers the rows, the application usually needs the number it just got — to insert the order's lines, say. In MySQL:

```
INSERT INTO orders (customer_id, placed_on) VALUES (42, '2024-05-01');
SELECT LAST_INSERT_ID();          -- the id generated for that row
```

`LAST_INSERT_ID()` is per connection, so two sessions inserting at once each see their own value — never compute "the next id" with `SELECT MAX(id) + 1`, which two sessions can read at the same moment. After a multi-row insert it returns the id of the **first** row. In PostgreSQL the idiom is `INSERT ... RETURNING id`, which returns the generated columns as a result set; MySQL has no `RETURNING`.

Two facts about InnoDB's counter that interviewers like: ids are not guaranteed to be gap-free (a rolled-back insert still consumes its number), and a deleted id is never handed out again — the counter only moves forward. Treat surrogate ids as labels, not as a count of rows.

Copying rows with INSERT ... SELECT

Instead of literal values, an `INSERT` can take the rows of a query:

```
INSERT INTO orders_archive (id, customer, total, placed_on)
SELECT id, customer, total, placed_on
FROM orders
WHERE placed_on < '2023-01-01';
```

The SELECT can be anything — joins, `GROUP BY`, `DISTINCT` — as long as it returns one column per listed column. The match is **by position**, not by name: the SELECT's third column goes into the INSERT's third column whatever either is called, so keep the two lists in the same order. This is how you archive, fill a summary table, or move rows from a staging table after an import:

```sql
INSERT INTO daily_sales (sale_date, orders, revenue)
SELECT placed_on, COUNT(*), SUM(total)
FROM orders
GROUP BY placed_on;
```

A close cousin creates the table and fills it in one go: `CREATE TABLE top_customers AS SELECT ...`. It copies column names and types from the query but not keys, indexes or defaults, so it suits scratch and reporting tables rather than real schema (the schema-design module returns to it).

When an INSERT fails

An insert that breaks a rule is refused and changes nothing:

- a duplicate value in a `PRIMARY KEY` or `UNIQUE` column — "Duplicate entry '7' for key 'PRIMARY'";
- `NULL` (or a missing value with no default) in a `NOT NULL` column;
- a value that fails a `CHECK` constraint (MySQL 8.0.16 and later);
- in strict mode, a string too long for its `VARCHAR`, or a value of the wrong type.

What to do with duplicates on purpose — skip them, overwrite them or update them — is the subject of the upserts lesson. One environment note: the practice engine here is MySQL syntax on SQLite, which does not truncate or reject an over-long string the way strict MySQL does, so the exercises never rely on that.

Pitfalls

- An `INSERT` without a column list, which breaks when the table changes.
- Expecting `NULL` to mean "use the default".
- Computing ids with `MAX(id) + 1` instead of `AUTO_INCREMENT`.
- An `INSERT ... SELECT` whose two column lists are in different orders — the types often match, so nothing complains.
- Thousands of single-row inserts under autocommit, each a separate transaction and log flush; batch them.

Key takeaways

- `INSERT INTO t (cols) VALUES (...), (...);` — always the column list; many rows per statement.
- Omitted columns take their DEFAULT, an AUTO_INCREMENT number or NULL; explicit NULL is not the default.
- `LAST_INSERT_ID()` (MySQL) or `RETURNING` (PostgreSQL) for the generated id; never `MAX(id) + 1`.
- `INSERT INTO ... SELECT` copies query results, matched by position.
- A constraint violation rejects the whole statement.

EXERCISE 1

Stock a new product line

The shop is adding three products. Insert them into `products`, always with an explicit column list — the first two in one multi-row INSERT:

id	name	category	price	in_stock
201	USB-C cable	cables	9.99	40
202	HDMI cable	cables	12.50	25
203	Cable ties	accessories	3.00	*(not given — use the column's default)*

Leave `in_stock` out of the third product's column list so the table's `DEFAULT` applies — don't write the default's value yourself (it is not always 0). Then show every product: columns `id, name, price, in_stock`, ordered by `id`.

The table already holds:

id	name	category	price	in_stock
101	Desk lamp	lighting	24.99	12
102	Floor lamp	lighting	79.00	4

The judge prints numbers without trailing zeros (79.00 as `79`, 12.50 as `12.5`).

```
id | name | price | in_stock
101 | Desk lamp | 24.99 | 12
102 | Floor lamp | 79 | 4
201 | USB-C cable | 9.99 | 40
202 | HDMI cable | 12.5 | 25
203 | Cable ties | 3 | 0
```

INPUT 1

```
CREATE TABLE products (
  id INT PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  category VARCHAR(20) NOT NULL,
  price DECIMAL(8,2) NOT NULL,
  in_stock INT NOT NULL DEFAULT 0
);
INSERT INTO products (id, name,
category, price, in_stock) VALUES (101,
'Desk lamp', 'lighting', 24.99, 12),
(102, 'Floor lamp', 'lighting', 79.00,
4);
```

OUTPUT 1

```
id | name | price | in_stock
101 | Desk lamp | 24.99 | 12
102 | Floor lamp | 79 | 4
201 | USB-C cable | 9.99 | 40
202 | HDMI cable | 12.5 | 25
203 | Cable ties | 3 | 0
```

REFERENCE SOLUTION

```sql
INSERT INTO products (id, name, category, price, in_stock)
VALUES
  (201, 'USB-C cable', 'cables', 9.99, 40),
  (202, 'HDMI cable', 'cables', 12.50, 25);
INSERT INTO products (id, name, category, price)
VALUES (203, 'Cable ties', 'accessories', 3.00);

SELECT id, name, price, in_stock
FROM products
ORDER BY id;
```

EXERCISE 2

Archive last year's orders

Copy every order placed **before 2024-01-01** from `orders` into `orders_archive` with one `INSERT ... SELECT`, setting `archived_on` to `'2024-06-30'` for each copied row. Do not change `orders`. Then show the archive: columns `id, customer, total, archived_on`, ordered by `id`.

`orders`:

id	customer	total	placed_on
1	Asha	120.00	2023-11-02
2	Ben	45.50	2023-12-31
3	Chen	80.00	2024-01-01
4	Asha	15.25	2024-02-14
5	Dev	60.00	2023-07-19

`orders_archive` has the same columns plus `archived_on DATE`, and starts empty here (it may already hold rows — keep them).

```
id | customer | total | archived_on
1 | Asha | 120 | 2024-06-30
2 | Ben | 45.5 | 2024-06-30
5 | Dev | 60 | 2024-06-30
```

INPUT 1

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer VARCHAR(30) NOT NULL,
  total DECIMAL(8,2) NOT NULL,
  placed_on DATE NOT NULL
);
CREATE TABLE orders_archive (
  id INT PRIMARY KEY,
  customer VARCHAR(30) NOT NULL,
  total DECIMAL(8,2) NOT NULL,
  placed_on DATE NOT NULL,
  archived_on DATE NOT NULL
);
INSERT INTO orders VALUES (1, 'Asha',
120.00, '2023-11-02'), (2, 'Ben', 45.50,
'2023-12-31'), (3, 'Chen', 80.00, '2024-
01-01'), (4, 'Asha', 15.25, '2024-02-
14'), (5, 'Dev', 60.00, '2023-07-19');
```

OUTPUT 1

```
id | customer | total | archived_on
1 | Asha | 120 | 2024-06-30
2 | Ben | 45.5 | 2024-06-30
5 | Dev | 60 | 2024-06-30
```

REFERENCE SOLUTION

```sql
INSERT INTO orders_archive (id, customer, total, placed_on, archived_on)
SELECT id, customer, total, placed_on, '2024-06-30'
FROM orders
WHERE placed_on < '2024-01-01';

SELECT id, customer, total, archived_on
FROM orders_archive
ORDER BY id;
```

QUIZ

1. Why write the column list in `INSERT INTO products (id, name, price) VALUES (...)` instead of `INSERT INTO products VALUES (...)`?

A. The column list is required by MySQL

B. It keeps working, and keeps values in the right columns, when the table gains or reorders columns

C. It makes the insert faster

D. Without it, defaults are applied to every column

2. A table declares `status VARCHAR(10) NOT NULL DEFAULT 'open'`. What does `INSERT INTO tickets (subject, status) VALUES ('VPN', NULL)` do?

A. Inserts the row with status `'open'`

B. Inserts the row with status NULL

C. Fails, because NULL was asked for explicitly and the column is NOT NULL

D. Inserts the row with an empty string

3. Two sessions insert orders at the same time and each needs the id it was given. What should each use?

A. `SELECT MAX(id) FROM orders` after the insert

B. `SELECT MAX(id) + 1` before the insert, then insert with that id

C. `LAST_INSERT_ID()` on its own connection

D. `SELECT COUNT(*) FROM orders`

4. How does `INSERT INTO archive (id, total, customer) SELECT id, customer, total FROM orders` match columns?

A. By name, so it works correctly

B. By position: `customer` goes into `total` and `total` into `customer`

C. It fails because the names differ

D. By data type

5. A three-row `INSERT ... VALUES (...), (...), (...)` has a duplicate primary key in the third row. On InnoDB, what is inserted?

A. The first two rows

B. Nothing

C. All three, with the duplicate renumbered

D. Only the third row

ANSWER KEY · INSERT — ADDING ROWS

1. B Without a list, the values must match every column in declared order, so a schema change breaks the statement or shifts values into the wrong columns. The list is optional syntax and has no effect on speed.

2. C The default is used only when the column is omitted (or `DEFAULT` is written). An explicit NULL is a value, and NOT NULL refuses it in strict mode.

3. C `LAST_INSERT_ID()` is per connection, so each session sees its own id. MAX-based approaches race: both sessions can read the same maximum.

4. B INSERT ... SELECT matches the SELECT's columns to the INSERT's list by position. Mismatched orders often go unnoticed when the types are compatible — keep both lists in the same order.

5. B A single statement is atomic: if any row fails, the whole statement is rolled back. Skipping only the bad rows is what INSERT IGNORE is for.

LESSON 2 · 14 MIN

UPDATE — changing rows

`UPDATE` changes values in rows that already exist. It is the statement with the largest blast radius in everyday SQL: one missing line turns "fix this customer's email" into "every customer now has this email". This lesson covers the shape of an UPDATE, how SET can compute from the row's own values, `CASE` for several rules in one pass, taking new values from another table (portable subqueries, MySQL's multi-table UPDATE and PostgreSQL's `UPDATE ... FROM`), and the habits that keep updates safe.

SET and WHERE

```
UPDATE customers
SET email = 'ada@example.com',
    updated_on = '2024-05-02'
WHERE id = 17;
```

`SET` lists one or more `column = expression` pairs separated by commas; `WHERE` chooses the rows. Every row the WHERE matches gets every assignment; rows it does not match are untouched. The WHERE is the same clause you know from SELECT — comparisons, `IN`, `BETWEEN`, `LIKE`, subqueries, `IS NULL` — and it obeys the same three-valued logic, so a row where the condition is NULL is **not** updated.

Leave the WHERE out and the UPDATE applies to every row:

```
UPDATE customers SET email = 'ada@example.com';   -- every customer. Oops.
```

There is no "are you sure?". Two habits prevent this. First, write the WHERE as a SELECT, run it, look at the rows, then turn it into the UPDATE. Second, for anything that matters, run it inside a transaction (`START TRANSACTION; UPDATE ...; SELECT ...;` then `COMMIT` or `ROLLBACK`). MySQL's client has a `--safe-updates` mode (`SET sql_safe_updates = 1`) that refuses an UPDATE or DELETE whose WHERE does not use a key.

Computing from the old value

The right-hand side of an assignment is an expression evaluated per row, and it can read the row's current values:

```
UPDATE products SET price = ROUND(price * 1.10, 2) WHERE category = 'books';
UPDATE accounts SET balance = balance - 250 WHERE id = 1;
UPDATE users SET login_count = login_count + 1, last_login = '2024-05-02' WHEI
```

`balance = balance - 250` is a single atomic read-modify-write inside the database; doing the same with a SELECT in the application followed by an UPDATE with the computed number is a classic lost-update bug when two requests race.

One MySQL quirk is an interview favourite. In a single-table UPDATE, MySQL evaluates assignments **left to right**, and later ones see earlier results:

```
-- a = 1, b = 0 before
UPDATE t SET a = a + 1, b = a;
-- MySQL:            a = 2, b = 2
-- PostgreSQL/SQLite: a = 2, b = 1   (standard: every right side reads the old
```

Do not write code that depends on either behaviour; if you need the old value, compute both sides from the original columns.

Several rules in one pass with CASE

When different rows need different changes, a `CASE` expression in SET does it in one statement and one scan:

```
UPDATE products
SET price = CASE
              WHEN category = 'books' THEN ROUND(price * 1.10, 2)
              WHEN category = 'toys'  THEN price - 5
              ELSE price
            END
WHERE category IN ('books', 'toys');
```

The `ELSE price` matters: a CASE with no matching branch and no ELSE returns NULL, and the column would become NULL. The WHERE restricts the statement to rows that some branch changes, so the others are not even rewritten. Running two separate UPDATEs would also

work, but if the rules can overlap (a row in both sets) the order of the statements silently changes the result; one CASE states the rule once.

Values from another table

Often the new values live in another table — a price list from a supplier, corrections from a spreadsheet loaded into a staging table. The portable way is a **correlated subquery** in SET plus a WHERE that limits the update to rows with a match:

```
UPDATE products
SET price = (SELECT c.new_price FROM price_changes c WHERE c.sku = products.sk
WHERE sku IN (SELECT sku FROM price_changes);
```

Without the WHERE, every product with no row in `price_changes` would get NULL from the subquery. The subquery must return at most one row per product, or the statement fails ("Subquery returns more than 1 row").

MySQL also has a **multi-table UPDATE** that joins directly:

```
UPDATE products p
JOIN price_changes c ON c.sku = p.sku
SET p.price = c.new_price,
    p.updated = 1;
```

Only rows with a match are touched, which is what the WHERE did above. PostgreSQL spells the same thing `UPDATE products SET price = c.new_price FROM price_changes c WHERE c.sku = products.sku;`. A related MySQL restriction: you cannot update a table and select from **the same table** in a subquery of that statement (error 1093, "You can't specify target table for update in FROM clause"); wrap the inner query in a derived table or use a join.

Rows matched, rows changed

MySQL reports two numbers after an UPDATE: rows **matched** by the WHERE and rows **changed**. Setting a column to the value it already holds matches the row but does not change it. Drivers usually return "affected rows", which by default means changed rows — so an UPDATE that reports 0 affected rows may still have found the row. Do not use the affected count alone to decide whether a row exists.

MySQL additionally allows `ORDER BY` and `LIMIT` on a single-table UPDATE (`UPDATE jobs SET claimed_by = 7 WHERE claimed_by IS NULL ORDER BY id LIMIT 1;`), handy for work queues; it is not standard SQL and the practice engine here does not run it.

Pitfalls

- A missing or wrong WHERE — test it as a SELECT first.
- A CASE without `ELSE column`, which NULLs the unmatched rows.
- A subquery in SET without the matching WHERE, which NULLs every row with no match.
- Read-then-write in the application instead of `col = col + n`.
- Relying on MySQL's left-to-right SET evaluation.
- Updating a primary key that other tables point to; change the surrogate design instead.

Key takeaways

- `UPDATE t SET a = x, b = y WHERE ...;` — the WHERE decides which rows, and leaving it out means all of them.
- Right-hand sides can use the row's current values: `qty = qty - 1` is atomic.
- `CASE ... ELSE col END` applies several rules in one pass.
- From another table: a correlated subquery plus a WHERE (portable), `UPDATE ... JOIN` (MySQL), `UPDATE ... FROM` (PostgreSQL).
- Matched is not changed; affected-row counts count changes.

EXERCISE 1

Seasonal repricing

Apply the season's price rules in **one** UPDATE using `CASE`:

- `books`: up 10%, rounded to 2 decimal places;
- `toys`: 20% off, rounded to 2 decimal places;
- every other product (including one with no category) keeps its price.

Then show the products: columns `id, name, category, price`, ordered by `id`.

id	name	category	price
1	SQL Basics	books	30.00
2	Robot kit	toys	49.99
3	Puzzle	toys	12.50
4	Desk	furniture	150.00
5	Atlas	books	19.99

```
id | name | category | price
1 | SQL Basics | books | 33
2 | Robot kit | toys | 39.99
3 | Puzzle | toys | 10
4 | Desk | furniture | 150
5 | Atlas | books | 21.99
```

INPUT 1

```
CREATE TABLE products (
  id INT PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  category VARCHAR(20),
  price DECIMAL(8,2) NOT NULL
);
INSERT INTO products VALUES (1, 'SQL
Basics', 'books', 30.00), (2, 'Robot
kit', 'toys', 49.99), (3, 'Puzzle',
'toys', 12.50), (4, 'Desk', 'furniture',
150.00), (5, 'Atlas', 'books', 19.99);
```

OUTPUT 1

```
id | name | category | price
1 | SQL Basics | books | 33
2 | Robot kit | toys | 39.99
3 | Puzzle | toys | 10
4 | Desk | furniture | 150
5 | Atlas | books | 21.99
```

REFERENCE SOLUTION

```sql
UPDATE products
SET price = CASE
              WHEN category = 'books' THEN ROUND(price * 1.10, 2)
              WHEN category = 'toys'  THEN ROUND(price * 0.80, 2)
              ELSE price
            END
WHERE category IN ('books', 'toys');

SELECT id, name, category, price
FROM products
ORDER BY id;
```

EXERCISE 2

Apply a supplier's price list

A supplier sent new prices, loaded into `price_changes(sku, new_price)`. For every product whose `sku` appears there, set `price` to the new price **and** set `repriced` to 1. Products not in the list must be left exactly as they are, and a price-list row for a sku the shop does not stock is ignored. Write it as portable SQL (a subquery in SET plus a WHERE), not MySQL's `UPDATE ... JOIN`.

Then show the products: columns `id, sku, price, repriced`, ordered by `id`.

`products`:

id	sku	name	price	repriced
1	A100	Kettle	25.00	0
2	B200	Toaster	30.00	0
3	C300	Blender	55.00	0
4	D400	Mixer	80.00	0

`price_changes`: (B200, 27.50), (D400, 85.00), (Z999, 10.00).

```
id | sku | price | repriced
1 | A100 | 25 | 0
2 | B200 | 27.5 | 1
3 | C300 | 55 | 0
4 | D400 | 85 | 1
```

INPUT 1

```
CREATE TABLE products (
  id INT PRIMARY KEY,
  sku VARCHAR(10) NOT NULL UNIQUE,
  name VARCHAR(40) NOT NULL,
  price DECIMAL(8,2) NOT NULL,
  repriced INT NOT NULL DEFAULT 0
);
CREATE TABLE price_changes (
  sku VARCHAR(10) PRIMARY KEY,
  new_price DECIMAL(8,2) NOT NULL
);
INSERT INTO products VALUES (1, 'A100',
'Kettle', 25.00, 0), (2, 'B200',
'Toaster', 30.00, 0), (3, 'C300',
'Blender', 55.00, 0), (4, 'D400',
'Mixer', 80.00, 0);
INSERT INTO price_changes VALUES
('B200', 27.50), ('D400', 85.00),
('Z999', 10.00);
```

OUTPUT 1

```
id | sku | price | repriced
1 | A100 | 25 | 0
2 | B200 | 27.5 | 1
3 | C300 | 55 | 0
4 | D400 | 85 | 1
```

REFERENCE SOLUTION

```
UPDATE products
SET price = (SELECT c.new_price FROM price_changes c WHERE c.sku = products.sku),
    repriced = 1
WHERE sku IN (SELECT sku FROM price_changes);

SELECT id, sku, price, repriced
FROM products
ORDER BY id;
```

QUIZ

1. What does `UPDATE customers SET country = 'IN';` do?

A. Nothing — an UPDATE needs a WHERE

B. Sets `country` to 'IN' on every row of the table

C. Sets it on the first row only

D. Asks for confirmation first

2. `UPDATE products SET price = CASE WHEN category = 'books' THEN price * 1.1 END;` runs on a table with books and toys. What happens to the toys?

A. Their price is unchanged

B. Their price becomes NULL (or the statement fails on a NOT NULL column)

C. Their price is multiplied by 1.1

D. The statement is rejected for a missing ELSE

3. Why is `UPDATE accounts SET balance = balance - 250 WHERE id = 1` safer than reading the balance in the application, subtracting, and writing the result back?

A. It is shorter

B. The database reads and writes the row atomically, so two concurrent withdrawals cannot overwrite each other

C. It skips the WHERE clause check

D. It avoids the need for an index

4. In MySQL, with `a = 1, b = 0`, what does `UPDATE t SET a = a + 1, b = a;` leave in `b`?

A. 0

B. 1

C. 2

D. NULL

5. Which statement updates `products` from `price_changes` in PostgreSQL?

A. `UPDATE products p JOIN price_changes c ON c.sku = p.sku SET p.price = c.new_price;`

B. `UPDATE products SET price = c.new_price FROM price_changes c WHERE c.sku = products.sku;`

C. `UPDATE products, price_changes SET price = new_price;`

D. `MERGE products SET price = new_price;`

ANSWER KEY · UPDATE — CHANGING ROWS

1. B A missing WHERE matches every row. MySQL only refuses it when `sql_safe_updates` is on; there is no confirmation.

2. B A CASE with no matching branch and no ELSE yields NULL, and there is no WHERE, so every toy is assigned NULL. Add `ELSE price` and a WHERE.

3. B Read-then-write in the application is the lost-update race: two requests read 1000, both write 750. The in-database expression locks the row while it computes.

4. C MySQL evaluates single-table SET assignments left to right, so `b = a` sees the new a (2). Standard SQL, PostgreSQL and SQLite would give 1. Don't rely on either.

5. B PostgreSQL uses `UPDATE ... FROM ... WHERE`. The JOIN form is MySQL's multi-table UPDATE; the comma form without a condition would pair every row with every row.

LESSON 3 · 14 MIN

DELETE, TRUNCATE and DROP

Removing data comes in three sizes. `DELETE` removes chosen rows. `TRUNCATE TABLE` removes every row at once. `DROP TABLE` removes the table itself. Interviewers ask about the three together more than almost any other DML question, and production incidents come from confusing them. This lesson covers DELETE with WHERE, deleting by a condition on another table (subqueries, MySQL's multi-table DELETE), what referencing rows do when their parent goes, the comparison of DELETE, TRUNCATE and DROP, and the soft-delete pattern many applications use instead.

DELETE with WHERE

```
DELETE FROM sessions
WHERE expires_on < '2024-06-01';
```

`DELETE FROM` takes a table and a WHERE; every row the WHERE matches is removed, whole — there is no "delete a column" (that is `UPDATE ... SET col = NULL` or `ALTER TABLE ... DROP COLUMN`). Like UPDATE, a DELETE without a WHERE affects every row:

```
DELETE FROM sessions;          -- the table is now empty, but it still exists
```

The same safety habits apply: run the WHERE as `SELECT COUNT(*)` first, and wrap important deletes in a transaction. DELETE is fully transactional in InnoDB — each removed row is written to the undo log, so `ROLLBACK` brings them all back — which is also why deleting millions of rows in one statement is slow and holds locks for a long time. Large purges are done in batches (`DELETE ... WHERE ... LIMIT 10000`, repeated; MySQL allows `LIMIT` on a single-table DELETE).

Deleting by a condition on another table

"Delete the orders whose customer no longer exists" needs another table in the condition. The portable tool is a subquery:

```sql
DELETE FROM orders
WHERE NOT EXISTS (
  SELECT 1 FROM customers c WHERE c.id = orders.customer_id
);
```

`NOT EXISTS` is true when the inner query finds no row, which includes orders whose `customer_id` is NULL (no customer equals NULL). The tempting `NOT IN` version has the trap from the subqueries module:

```sql
DELETE FROM orders WHERE customer_id NOT IN (SELECT id FROM customers);
```

If `customers.id` could ever be NULL, `x NOT IN (..., NULL)` is unknown for every row and the statement deletes nothing; and an order with a NULL `customer_id` is never deleted either, because `NULL NOT IN (...)` is unknown. Prefer `NOT EXISTS` for anti-joins in DML.

The positive form uses `IN` or `EXISTS`:

```sql
DELETE FROM order_items
WHERE order_id IN (SELECT id FROM orders WHERE status = 'cancelled');
```

MySQL also has a **multi-table DELETE** with a join, naming the table to delete from before `FROM`:

```sql
DELETE oi
FROM order_items oi
JOIN orders o ON o.id = oi.order_id
WHERE o.status = 'cancelled';
```

PostgreSQL writes it `DELETE FROM order_items oi USING orders o WHERE o.id = oi.order_id AND o.status = 'cancelled';`. As with UPDATE, MySQL refuses a DELETE whose subquery reads the same table it deletes from (error 1093); use a join or a derived table.

What happens to rows that point at a deleted row

If `order_items.order_id` is a foreign key to `orders.id`, deleting an order that still has items is governed by the key's `ON DELETE` action:

Action	Deleting the parent…
`RESTRICT` / `NO ACTION` (default)	is refused while children exist
`CASCADE`	deletes the children too
`SET NULL`	keeps the children, their key becomes NULL

InnoDB enforces these. `CASCADE` is convenient and dangerous: one DELETE on a customer can remove orders, items and payments several levels down. The schema-design module covers declaring them. (The practice engine here declares foreign keys but does not enforce them, so the exercises delete children explicitly — which is also what you do when the key is `RESTRICT`.)

DELETE vs TRUNCATE vs DROP

```
DELETE FROM logs;          -- remove every row, one at a time
TRUNCATE TABLE logs;       -- empty the table in one step
DROP TABLE logs;           -- remove the table altogether
```

	DELETE	TRUNCATE	DROP
Kind	DML	DDL	DDL
WHERE	yes	no — all rows	n/a
Speed on a big table	slow (per row, logged)	fast (recreates the table)	fast
Rollback in MySQL	yes, in a transaction	no — implicit commit	no — implicit commit
AUTO_INCREMENT	kept	reset	gone with the table
Triggers	`ON DELETE` triggers fire	do not fire	do not fire
Structure, indexes, grants	kept	kept	removed
Referenced by a foreign key	allowed row by row	refused in InnoDB	refused

The short version for an interview: DELETE removes **rows** (filtered, logged, undoable), TRUNCATE removes **all rows** (fast, not undoable in MySQL, counter reset), DROP removes **the table**. In PostgreSQL, TRUNCATE and DROP are transactional and can be rolled back; in MySQL and Oracle they commit implicitly. `DROP TABLE IF EXISTS t` avoids an error when the table is absent, which migration scripts rely on.

Soft deletes

Many applications never physically delete business rows. They add a column and filter on it:

```
ALTER TABLE customers ADD COLUMN deleted_at DATETIME NULL;

UPDATE customers SET deleted_at = '2024-05-02 10:15:00' WHERE id = 17;    -- "
SELECT id, name FROM customers WHERE deleted_at IS NULL;                    --
```

The benefits are an audit trail, an easy undo and references that never dangle. The costs are that every query must remember the filter (a view over the live rows helps), unique constraints

must allow a deleted twin, and the table grows forever. Data-protection rules may still require a real delete of personal data.

Pitfalls

- A DELETE without a WHERE, or with a WHERE that was never tested as a SELECT.
- `NOT IN` against a nullable column — use `NOT EXISTS`.
- TRUNCATE inside a transaction in MySQL, expecting ROLLBACK to undo it.
- Forgetting cascades: a small delete removing far more than intended.
- One enormous DELETE on a busy table instead of batches.

Key takeaways

- `DELETE FROM t WHERE ...;` removes whole rows; without WHERE, all of them.
- Use `NOT EXISTS` / `EXISTS` (or MySQL's `DELETE t FROM t JOIN ...`) to delete by another table.
- `ON DELETE RESTRICT | CASCADE | SET NULL` decides what happens to children.
- DELETE is logged and undoable; TRUNCATE empties fast, resets the counter and commits in MySQL; DROP removes the table.
- Soft deletes mark rows instead of removing them.

EXERCISE 1

Purge expired sessions

Delete every session that expired **before 2024-06-01** (a session expiring on 2024-06-01 itself stays). A session with no expiry date (`expires_on` is NULL) never expires and must stay.

Then print two result sets: the number of sessions left as `remaining`, and the sessions left with columns `id, user_id, expires_on`, ordered by `id`.

id	user_id	expires_on
1	10	2024-05-30
2	11	2024-06-01
3	10	2024-07-15
4	12	2024-01-01
5	13	2024-06-02

```
remaining
3

id | user_id | expires_on
2 | 11 | 2024-06-01
3 | 10 | 2024-07-15
5 | 13 | 2024-06-02
```

INPUT 1

```
CREATE TABLE sessions (
  id INT PRIMARY KEY,
  user_id INT NOT NULL,
  expires_on DATE
);
INSERT INTO sessions VALUES (1, 10,
'2024-05-30'), (2, 11, '2024-06-01'),
(3, 10, '2024-07-15'), (4, 12, '2024-01-
01'), (5, 13, '2024-06-02');
```

OUTPUT 1

```
remaining
3

id | user_id | expires_on
2 | 11 | 2024-06-01
3 | 10 | 2024-07-15
5 | 13 | 2024-06-02
```

REFERENCE SOLUTION

```sql
DELETE FROM sessions
WHERE expires_on < '2024-06-01';

SELECT COUNT(*) AS remaining FROM sessions;

SELECT id, user_id, expires_on
FROM sessions
ORDER BY id;
```

EXERCISE 2

Remove orphaned orders

The `orders` table has rows whose customer was removed long ago. Clean up in two DELETEs:

1. delete every order whose `customer_id` matches no row in `customers` — **including** orders whose `customer_id` is NULL;
2. then delete every customer who has no orders left.

Then print the orders left (`id, customer_id`, ordered by `id`) and the customers left (`id, name`, ordered by `id`).

`customers`: (1, Asha), (2, Ben), (3, Chen).

`orders`:

id	customer_id	total
10	1	50.00
11	4	20.00
12	2	35.00
13	NULL	15.00
14	1	10.00
15	5	99.00

```
id | customer_id
10 | 1
12 | 2
14 | 1

id | name
1 | Asha
2 | Ben
```

INPUT 1

```
CREATE TABLE customers (
  id INT PRIMARY KEY,
  name VARCHAR(30) NOT NULL
);
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT,
  total DECIMAL(8,2) NOT NULL
);
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Chen');
INSERT INTO orders VALUES (10, 1,
50.00), (11, 4, 20.00), (12, 2, 35.00),
(13, NULL, 15.00), (14, 1, 10.00), (15,
5, 99.00);
```

OUTPUT 1

```
id | customer_id
10 | 1
12 | 2
14 | 1

id | name
1 | Asha
2 | Ben
```

REFERENCE SOLUTION

```
DELETE FROM orders
WHERE NOT EXISTS (
  SELECT 1 FROM customers c WHERE c.id = orders.customer_id
);

DELETE FROM customers
WHERE NOT EXISTS (
  SELECT 1 FROM orders o WHERE o.customer_id = customers.id
);

SELECT id, customer_id FROM orders ORDER BY id;

SELECT id, name FROM customers ORDER BY id;
```

QUIZ

1. Which of these can be rolled back in MySQL when run inside `START TRANSACTION ... ROLLBACK`?

A. `DELETE FROM logs;`

B. `TRUNCATE TABLE logs;`

C. `DROP TABLE logs;`

D. All three

2. After `DELETE FROM tickets;` on an InnoDB table whose AUTO_INCREMENT had reached 500, what id does the next inserted row get?

A. 1

B. 500

C. 501

D. It depends on the number of rows deleted

3. `DELETE FROM orders WHERE customer_id NOT IN (SELECT id FROM legacy_customers);` deletes nothing, though many orders have no legacy customer. The likely cause?

A. NOT IN does not work in DELETE

B. `legacy_customers.id` contains a NULL, making the condition unknown for every row

C. The subquery needs DISTINCT

D. DELETE needs a LIMIT

4. A foreign key `order_items.order_id → orders.id` is declared `ON DELETE CASCADE`. What does deleting an order do?

A. Fails while the order has items

B. Deletes the order and its items

C. Deletes the order and sets the items' order_id to NULL

D. Deletes only the items

5. What is a soft delete?

A. A DELETE inside a transaction

B. A DELETE with a LIMIT

C. Marking a row as deleted (e.g. setting `deleted_at`) and filtering it out of queries instead of removing it

D. TRUNCATE without resetting AUTO_INCREMENT

ANSWER KEY · DELETE, TRUNCATE AND DROP

1. A DELETE is DML and fully transactional in InnoDB. TRUNCATE and DROP are DDL in MySQL and cause an implicit commit. (In PostgreSQL all three can be rolled back.)

2. C DELETE leaves the counter where it was, so numbering continues. TRUNCATE TABLE would reset it to 1.

3. B `x NOT IN (1, 2, NULL)` is never true — it is false or unknown. NOT EXISTS has no such trap.

4. B CASCADE removes the children with the parent. RESTRICT/NO ACTION would refuse; SET NULL would keep the items with a NULL key.

5. C The row stays, so history and references survive and undo is an UPDATE; every query must filter `deleted_at IS NULL`.

LESSON 4 · 15 MIN

Upserts — insert or update

Loading data from outside rarely tells you which rows are new. A product feed lists every product; a page-view batch names pages you have seen before and pages you have not. The operation you want is "insert this row, or if a row with that key already exists, update it" — an **upsert**. Standard SQL calls it `MERGE`; MySQL has three of its own tools. This lesson covers MySQL's `ON DUPLICATE KEY UPDATE`, `REPLACE INTO` and `INSERT IGNORE`, how they differ on the row that already exists, PostgreSQL's `ON CONFLICT` and `MERGE`, and the two-statement form that works everywhere.

Every one of them depends on a **PRIMARY KEY or UNIQUE constraint**: that is how the database knows a row "already exists". Without one, every upsert is just an insert.

INSERT ... ON DUPLICATE KEY UPDATE

```
CREATE TABLE stock (
  sku VARCHAR(20) PRIMARY KEY,
  qty INT NOT NULL,
  updated_on DATE
);

INSERT INTO stock (sku, qty, updated_on)
VALUES ('A1', 5, '2024-05-02')
ON DUPLICATE KEY UPDATE
  qty = qty + 5,
  updated_on = '2024-05-02';
```

MySQL attempts the insert. If it succeeds, done. If it would violate the primary key or any unique index, MySQL instead runs the `UPDATE` part on the row that is in the way. Inside that part, a bare column name means the **existing** row's value, so `qty = qty + 5` adds to it. To refer to the value you tried to insert, MySQL 8.0.19 and later let you alias the new row:

```
INSERT INTO stock (sku, qty, updated_on)
VALUES ('A1', 5, '2024-05-02'), ('B7', 12, '2024-05-02') AS new
ON DUPLICATE KEY UPDATE
  qty = stock.qty + new.qty,
  updated_on = new.updated_on;
```

Older code writes `VALUES(qty)` for the same thing; that function is deprecated in 8.0 but you will see it everywhere. The statement's affected-row count is 1 for an insert, 2 for an update that changed something and 0 for an update that changed nothing — a common interview detail.

Because the existing row is modified in place, its id, its other columns and any rows that reference it are untouched. This is usually what "upsert" should mean.

REPLACE INTO

```
REPLACE INTO stock (sku, qty, updated_on)
VALUES ('A1', 5, '2024-05-02');
```

`REPLACE` has the same syntax as `INSERT` (single row, many rows or `REPLACE ... SELECT`), but on a duplicate key it **deletes the old row and inserts the new one**. The consequences:

- columns you did not list get their defaults, not their old values — `REPLACE INTO stock (sku, qty)` would set `updated_on` to NULL;
- you cannot add to the old value; the new row simply wins;
- `ON DELETE` triggers fire and, with real foreign keys, `ON DELETE CASCADE` can remove child rows;
- an AUTO_INCREMENT id on the replaced row changes if the duplicate was on a different unique column.

REPLACE fits when the incoming row is the complete truth — a nightly feed that carries every column of a lookup table. It is wrong for counters and for rows other tables point to.

INSERT IGNORE

```
INSERT IGNORE INTO subscribers (email, name)
VALUES ('ada@example.com', 'Ada'), ('alan@example.com', 'Alan');
```

`INSERT IGNORE` inserts the rows it can and **skips** any row that would duplicate a key, leaving the existing row exactly as it was. It is the "insert if absent" half of an upsert: deduplicating an import, recording that something happened at least once, filling a junction table. In MySQL, `IGNORE` also turns some other errors — a value out of range, a NULL forced into a NOT NULL

column in a multi-row insert — into warnings with an adjusted value, which can quietly store junk. After a bulk `INSERT IGNORE`, check `SHOW WARNINGS`. Which duplicate survives is decided by order: the first row inserted wins, so give an `INSERT IGNORE ... SELECT` an `ORDER BY` when it matters.

The portable two-step: UPDATE, then INSERT the rest

Every SQL database can upsert from a staging table in two statements: update the rows that exist, then insert the ones that do not.

```
-- page_views(path PRIMARY KEY, views); today(path, hits) is the new batch
UPDATE page_views
SET views = views + (SELECT t.hits FROM today t WHERE t.path = page_views.path
WHERE path IN (SELECT path FROM today);

INSERT INTO page_views (path, views)
SELECT t.path, t.hits
FROM today t
WHERE NOT EXISTS (SELECT 1 FROM page_views p WHERE p.path = t.path);
```

The order matters: update first, so that rows inserted in step two are not updated again. Run both inside one transaction so no other session sees the half-done state. The set-based form — the whole batch in two statements — is far faster than a loop doing one upsert per row.

PostgreSQL and the standard

PostgreSQL's equivalent of `ON DUPLICATE KEY UPDATE` names the conflicting key and refers to the rejected row as `EXCLUDED`:

```
INSERT INTO stock (sku, qty) VALUES ('A1', 5)
ON CONFLICT (sku) DO UPDATE SET qty = stock.qty + EXCLUDED.qty;

INSERT INTO subscribers (email, name) VALUES ('ada@example.com', 'Ada')
ON CONFLICT (email) DO NOTHING;          -- PostgreSQL's INSERT IGNORE
```

SQL Server, Oracle and PostgreSQL 15+ also implement the standard `MERGE`:

```sql
MERGE INTO stock s
USING feed f ON s.sku = f.sku
WHEN MATCHED THEN UPDATE SET qty = f.qty
WHEN NOT MATCHED THEN INSERT (sku, qty) VALUES (f.sku, f.qty);
```

MySQL supports neither `ON CONFLICT` nor `MERGE`. In an interview, say which database you mean.

Why not SELECT, then decide?

The intuitive version — `SELECT` the key, then `INSERT` if no row came back or `UPDATE` if one did — has a race. Two sessions run the SELECT at the same moment, both see nothing, both INSERT; one fails with a duplicate-key error, or, with no unique key, you now have two rows. A single upsert statement lets the database check and write atomically under the key's lock. If you must decide in the application, the unique key is still your safety net: catch the duplicate-key error and retry as an update.

In the practice engine

The engine here runs `REPLACE INTO`, `INSERT IGNORE` and the two-step form, which is what the exercises judge. `ON DUPLICATE KEY UPDATE`, `ON CONFLICT` and `MERGE` are for reading — learn their shape and the differences above, which is what interviews ask about.

Key takeaways

- An upsert needs a PRIMARY KEY or UNIQUE constraint to detect "already exists".
- `ON DUPLICATE KEY UPDATE` updates the existing row in place; `new.col` (or `VALUES(col)`) is the attempted value.
- `REPLACE` deletes and re-inserts: unlisted columns reset, triggers and cascades fire.
- `INSERT IGNORE` skips duplicates and keeps the existing row; watch its warnings.
- PostgreSQL: `ON CONFLICT ... DO UPDATE / DO NOTHING` with `EXCLUDED`; elsewhere `MERGE`.
- Portable: UPDATE the matches, then INSERT the rest with `NOT EXISTS`, in one transaction.

EXERCISE 1

Load a product feed with REPLACE

The nightly `feed` table is the complete truth for every product it mentions: its name and price replace whatever the catalogue had, and the product's `on_sale` flag goes back to its default (0), because the feed does not carry it. Products the feed does not mention stay as they are; new skus are added.

Do it with **one** `REPLACE INTO ... SELECT`, then show the catalogue: columns `sku, name, price, on_sale`, ordered by `sku`.

`catalog` (sku is the primary key; `on_sale` defaults to 0):

sku	name	price	on_sale
A1	Pen	1.50	1
B2	Pad	3.00	1
C3	Ink	7.25	0

`feed(sku, name, price)`: (B2, Notepad, 3.25), (D4, Ruler, 2.00).

```
sku | name | price | on_sale
A1 | Pen | 1.5 | 1
B2 | Notepad | 3.25 | 0
C3 | Ink | 7.25 | 0
D4 | Ruler | 2 | 0
```

INPUT 1

```
CREATE TABLE catalog (
  sku VARCHAR(10) PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  price DECIMAL(8,2) NOT NULL,
  on_sale INT NOT NULL DEFAULT 0
);
CREATE TABLE feed (
  sku VARCHAR(10) PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  price DECIMAL(8,2) NOT NULL
);
INSERT INTO catalog VALUES ('A1', 'Pen',
1.50, 1), ('B2', 'Pad', 3.00, 1), ('C3',
'Ink', 7.25, 0);
INSERT INTO feed VALUES ('B2',
'Notepad', 3.25), ('D4', 'Ruler', 2.00);
```

OUTPUT 1

```
sku | name | price | on_sale
A1 | Pen | 1.5 | 1
B2 | Notepad | 3.25 | 0
C3 | Ink | 7.25 | 0
D4 | Ruler | 2 | 0
```

REFERENCE SOLUTION

```
REPLACE INTO catalog (sku, name, price)
SELECT sku, name, price
FROM feed;

SELECT sku, name, price, on_sale
FROM catalog
ORDER BY sku;
```

EXERCISE 2

Count page views: update, then insert

`page_views(path, views)` keeps a running total per page; `today(path, hits)` is today's batch, one row per path. Add today's hits to the pages already counted, and insert the pages seen for the first time with their hits. REPLACE would lose the running total, so use the portable two-step: an UPDATE for the existing paths, then an `INSERT ... SELECT` with `NOT EXISTS` for the new ones.

Then show the totals: columns `path, views`, ordered by `views` descending, then `path`.

`page_views`: (/, 120), (/pricing, 40), (/blog, 15). `today`: (/, 30), (/docs, 12), (/blog, 15).

```
path | views
/ | 150
/pricing | 40
/blog | 30
/docs | 12
```

INPUT 1

```
CREATE TABLE page_views (
  path VARCHAR(50) PRIMARY KEY,
  views INT NOT NULL
);
CREATE TABLE today (
  path VARCHAR(50) PRIMARY KEY,
  hits INT NOT NULL
);
INSERT INTO page_views VALUES ('/',
120), ('/pricing', 40), ('/blog', 15);
INSERT INTO today VALUES ('/', 30),
('/docs', 12), ('/blog', 15);
```

OUTPUT 1

```
path | views
/ | 150
/pricing | 40
/blog | 30
/docs | 12
```

REFERENCE SOLUTION

```
UPDATE page_views
SET views = views + (SELECT t.hits FROM today t WHERE t.path = page_views.path)
WHERE path IN (SELECT path FROM today);

INSERT INTO page_views (path, views)
SELECT t.path, t.hits
FROM today t
WHERE NOT EXISTS (SELECT 1 FROM page_views p WHERE p.path = t.path);

SELECT path, views
FROM page_views
ORDER BY views DESC, path;
```

QUIZ

1. `stock(sku PRIMARY KEY, qty)` holds `('A1', 10)`. What does `INSERT INTO stock (sku, qty) VALUES ('A1', 5) ON DUPLICATE KEY UPDATE qty = qty + 5;` leave?

   A. `('A1', 5)`

   B. `('A1', 10)`

   C. `('A1', 15)`

   D. Two rows for A1

2. `users(id AUTO_INCREMENT PRIMARY KEY, email UNIQUE, name, plan DEFAULT 'free')` has `(7, 'a@x.io', 'Ann', 'pro')`. What does `REPLACE INTO users (email, name) VALUES ('a@x.io', 'Ann B')` produce?

A. `(7, 'a@x.io', 'Ann B', 'pro')`

B. A new row with a new id, name 'Ann B' and plan 'free'; the old row is deleted

C. An error: duplicate entry

D. Two rows for a@x.io

3. Which statement inserts a subscriber only if the email is not already present, leaving an existing row untouched?

A. `REPLACE INTO subscribers ...`

B. `INSERT IGNORE INTO subscribers ...`

C. `INSERT INTO subscribers ... ON DUPLICATE KEY UPDATE name = new.name`

D. `UPDATE subscribers ...`

4. What is PostgreSQL's equivalent of MySQL's `ON DUPLICATE KEY UPDATE`?

A. `ON CONFLICT (key) DO UPDATE SET col = EXCLUDED.col`

B. `REPLACE INTO`

C. `INSERT OR UPDATE`

D. `UPSERT INTO`

5. Why is "SELECT the key; if no row, INSERT, else UPDATE" unsafe under concurrency?

A. SELECT locks the whole table

B. Two sessions can both see no row and both insert, producing a duplicate or a key error

C. UPDATE cannot follow a SELECT

D. It is safe; it is only slower

ANSWER KEY · UPSERTS — INSERT OR UPDATE

1. C The insert collides on the primary key, so the UPDATE part runs on the existing row, where a bare `qty` is the existing value: 10 + 5.

2. B REPLACE deletes the conflicting row and inserts a fresh one: the id is generated again and unlisted columns take their defaults. ON DUPLICATE KEY UPDATE would have kept id 7 and the plan.

3. B INSERT IGNORE skips rows that would duplicate a key. REPLACE overwrites, and ON DUPLICATE KEY UPDATE changes the existing row.

4. A PostgreSQL names the conflict target and calls the rejected row `EXCLUDED`; `DO NOTHING` is its INSERT IGNORE. It has no REPLACE.

5. B The check and the write are separate steps, so another session can act in between. An upsert statement (or a unique key plus error handling) makes the decision atomic.

LESSON 5 · 14 MIN

Transactions in practice

The previous lessons changed data one statement at a time. Real changes usually take several statements that only make sense together: move money out of one account **and** into another; create an order **and** its lines **and** reduce stock. If the second statement fails or the server crashes between them, the data is wrong. A **transaction** makes a group of statements all-or-nothing. This lesson is the practical side — the statements, autocommit, savepoints, the DDL that commits behind your back, and how to shape a transaction. Isolation levels, locking and MVCC, which decide what concurrent transactions see of each other, get their own module later.

START TRANSACTION, COMMIT, ROLLBACK

```
START TRANSACTION;                                  -- or BEGIN
UPDATE accounts SET balance = balance - 250 WHERE id = 1;
UPDATE accounts SET balance = balance + 250 WHERE id = 2;
INSERT INTO transfers (from_id, to_id, amount) VALUES (1, 2, 250);
COMMIT;
```

Between `START TRANSACTION` and `COMMIT`, the changes exist only for this session. `COMMIT` makes all three permanent at once; until then, other sessions still see the old balances (under InnoDB's default isolation), and a crash rolls everything back on restart. `ROLLBACK` instead of `COMMIT` discards all three:

```
START TRANSACTION;
DELETE FROM orders WHERE placed_on < '2020-01-01';
SELECT COUNT(*) FROM orders;      -- inside the transaction you see your own c
ROLLBACK;                         -- … and now they never happened
```

Inside a transaction you always **read your own writes**: the SELECT above counts the orders that are left after the DELETE, even though nobody else can see that state yet. This is the basis of the "try it, look, then decide" workflow for risky changes — run the UPDATE, run a SELECT to check it, then COMMIT or ROLLBACK.

These four properties are what the acronym **ACID** names: *atomicity* (all or nothing), *consistency* (constraints hold at commit), *isolation* (concurrent transactions do not see each

other's partial work) and *durability* (once committed, it survives a crash).

Autocommit

MySQL starts every session with `autocommit = 1`: a statement run outside an explicit transaction is a transaction by itself and is committed the moment it succeeds. That is why a stray `DELETE` cannot be undone — it was already committed. `START TRANSACTION` switches autocommit off until the next COMMIT or ROLLBACK. You can also turn it off for the session:

```
SET autocommit = 0;
UPDATE products SET price = price * 0.9;
-- nothing is permanent until:
COMMIT;
```

Many drivers and frameworks manage this for you ("begin a transaction, run the block, commit, or roll back on an exception"). PostgreSQL also autocommits by default; Oracle does not, and its clients commit on exit.

Savepoints

A savepoint marks a point inside a transaction you can roll back to without abandoning the whole transaction:

```
START TRANSACTION;
INSERT INTO orders (id, customer_id) VALUES (900, 7);
SAVEPOINT lines;
INSERT INTO order_items (order_id, sku, qty) VALUES (900, 'A1', 2);
INSERT INTO order_items (order_id, sku, qty) VALUES (900, 'Z9', 1);   -- wrong
ROLLBACK TO SAVEPOINT lines;       -- both item rows are gone, the order stays
INSERT INTO order_items (order_id, sku, qty) VALUES (900, 'A1', 2);
COMMIT;
```

`ROLLBACK TO SAVEPOINT name` undoes everything after the savepoint and leaves the transaction open; `RELEASE SAVEPOINT name` removes the marker; a plain `ROLLBACK` still undoes the whole transaction, savepoints included. Savepoints matter in long batch jobs (retry one chunk) and are what frameworks use to implement "nested transactions".

Errors inside a transaction

In MySQL, a statement that fails inside a transaction — a duplicate key, say — is rolled back **by itself**; the transaction stays open with the earlier statements still applied. MySQL does not roll back the whole transaction for you (a deadlock is the exception: the victim's whole transaction is rolled back). So the application must decide: on an error, issue `ROLLBACK`. In practice that is a `try/except` around the transaction in the application, or a `DECLARE ... HANDLER` in a stored procedure. PostgreSQL behaves differently and refuses every further statement in a failed transaction until you roll back — a frequent surprise when moving between the two.

DDL commits implicitly

In MySQL, `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, `TRUNCATE TABLE`, `RENAME TABLE` and `CREATE INDEX` cause an **implicit commit**: before running, they commit whatever the current transaction has done, and they cannot be rolled back.

```
START TRANSACTION;
UPDATE accounts SET balance = 0;
TRUNCATE TABLE audit_log;          -- implicit COMMIT here: the UPDATE is now
ROLLBACK;                          -- rolls back nothing
```

PostgreSQL and SQL Server run most DDL inside transactions, so a migration that fails halfway rolls back cleanly there and does not in MySQL. Keep DDL out of data transactions in MySQL, and write migrations so each step can be re-run. (The practice engine here keeps DDL transactional like SQLite does, so the exercises never depend on either behaviour.)

Shaping a good transaction

- **Short.** An open transaction holds row locks and keeps old versions of rows alive. Never hold one open while waiting for a user, an HTTP call or a file.
- **Complete.** Put in it everything that must succeed together and nothing else.
- **Consistent order.** When several transactions update the same tables, updating rows in the same order (by id, say) avoids most deadlocks.
- **Set-based.** One UPDATE over a thousand rows beats a thousand single-row UPDATEs inside a loop.

- **Retry-safe.** Deadlocks and lock timeouts happen under load; the application should retry the whole transaction.

Key takeaways

- `START TRANSACTION` … `COMMIT` makes a group of changes permanent together; `ROLLBACK` undoes them all.
- Inside a transaction you read your own uncommitted writes; others do not.
- MySQL autocommits each statement unless a transaction is open.
- `SAVEPOINT` / `ROLLBACK TO SAVEPOINT` undo part of a transaction and keep it open.
- A failed statement in MySQL is undone alone; the application decides to roll back the rest.
- DDL commits implicitly in MySQL and cannot be rolled back.

EXERCISE 1

Transfer between accounts

Move **250** from account 1 to account 2 as one transaction: start a transaction, take 250 from account 1's balance, add 250 to account 2's, record the transfer in `transfers(from_id, to_id, amount)`, and commit.

Then print the accounts (`id, owner, balance`, ordered by `id`) and the transfer log (`from_id, to_id, amount`, ordered by `from_id`, `to_id`, `amount`).

`accounts`: (1, Asha, 1000.00), (2, Ben, 300.00), (3, Chen, 50.00). `transfers` starts empty here (it may already hold rows — keep them).

```
id | owner | balance
1 | Asha | 750
2 | Ben | 550
3 | Chen | 50

from_id | to_id | amount
1 | 2 | 250
```

INPUT 1

```
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  owner VARCHAR(30) NOT NULL,
  balance DECIMAL(10,2) NOT NULL
);
CREATE TABLE transfers (
  from_id INT NOT NULL,
  to_id INT NOT NULL,
  amount DECIMAL(10,2) NOT NULL
);
INSERT INTO accounts VALUES (1, 'Asha',
1000.00), (2, 'Ben', 300.00), (3,
'Chen', 50.00);
```

OUTPUT 1

```
id | owner | balance
1 | Asha | 750
2 | Ben | 550
3 | Chen | 50

from_id | to_id | amount
1 | 2 | 250
```

REFERENCE SOLUTION

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 250 WHERE id = 1;
UPDATE accounts SET balance = balance + 250 WHERE id = 2;
INSERT INTO transfers (from_id, to_id, amount) VALUES (1, 2, 250);
COMMIT;

SELECT id, owner, balance FROM accounts ORDER BY id;

SELECT from_id, to_id, amount FROM transfers ORDER BY from_id, to_id, amount;
```

EXERCISE 2

Undo part of a transaction with a savepoint

Follow these steps exactly, in one script:

1. start a transaction;
2. mark every `pending` order as `shipped`;
3. create a savepoint named `before_purge`;
4. delete every order whose `total` is 0;
5. print `SELECT COUNT(*) AS orders_left FROM orders` — inside the transaction you see your own delete;
6. the purge was a mistake: roll back to `before_purge` (the shipping from step 2 must survive);
7. commit;
8. print the orders: `id, status, total`, ordered by `id`.

id	status	total
1	pending	40.00
2	shipped	0.00
3	pending	0.00
4	cancelled	15.00

```
orders_left
2

id | status | total
1 | shipped | 40
2 | shipped | 0
3 | shipped | 0
4 | cancelled | 15
```

INPUT 1

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  status VARCHAR(10) NOT NULL,
  total DECIMAL(8,2) NOT NULL
);
INSERT INTO orders VALUES (1, 'pending',
40.00), (2, 'shipped', 0.00), (3,
'pending', 0.00), (4, 'cancelled',
15.00);
```

OUTPUT 1

```
orders_left
2

id | status | total
1 | shipped | 40
2 | shipped | 0
3 | shipped | 0
4 | cancelled | 15
```

REFERENCE SOLUTION

```sql
START TRANSACTION;
UPDATE orders SET status = 'shipped' WHERE status = 'pending';
SAVEPOINT before_purge;
DELETE FROM orders WHERE total = 0;
SELECT COUNT(*) AS orders_left FROM orders;
ROLLBACK TO SAVEPOINT before_purge;
COMMIT;

SELECT id, status, total FROM orders ORDER BY id;
```

QUIZ

1. With MySQL's default settings, you run `DELETE FROM orders WHERE id = 5;` and then `ROLLBACK;`. Is the row back?

A. Yes, ROLLBACK always undoes the last statement

B. No — with autocommit on, the DELETE was committed as soon as it succeeded

C. Only if the table has a primary key

D. Yes, but only for 60 seconds

2. Inside an open transaction you run `UPDATE products SET price = 0;` and then `SELECT SUM(price) FROM products;`. What does the SELECT return?

A. The old total, until you COMMIT

B. 0 — a transaction reads its own uncommitted writes

C. An error: the table is locked

D. NULL

3. `START TRANSACTION; UPDATE accounts SET balance = 0; TRUNCATE TABLE audit_log; ROLLBACK;` — in MySQL, what is the state afterwards?

A. Both changes are undone

B. Both changes are permanent: TRUNCATE committed the UPDATE implicitly, and TRUNCATE itself cannot be rolled back

C. The UPDATE is undone, the TRUNCATE is permanent

D. The TRUNCATE is undone, the UPDATE is permanent

4. What does `ROLLBACK TO SAVEPOINT s1` do?

A. Undoes the whole transaction and ends it

B. Undoes the changes made after `s1` and keeps the transaction open

C. Commits everything before `s1`

D. Deletes the savepoint and nothing else

5. Inside a MySQL transaction, the third of five statements fails with a duplicate-key error. What has MySQL done?

A. Rolled back the entire transaction

B. Rolled back only the failed statement; the transaction is still open with statements 1–2 applied

C. Committed statements 1–2

D. Refused every further statement until ROLLBACK

ANSWER KEY · TRANSACTIONS IN PRACTICE

1. B Outside an explicit transaction each statement is its own committed transaction. ROLLBACK only undoes work since START TRANSACTION (or with autocommit off).

2. B Your own session always sees its own changes. Other sessions keep seeing the committed values until you commit.

3. B DDL in MySQL causes an implicit commit before it runs. By the time ROLLBACK executes there is nothing left to roll back.

4. B Work before the savepoint stays pending and can still be committed or rolled back. `RELEASE SAVEPOINT` removes a marker without undoing anything.

5. B MySQL undoes just the failing statement; the application decides whether to roll back the rest. Refusing further statements is PostgreSQL's behaviour.

CHECKPOINT · 25 MIN

Checkpoint — Modifying data

This checkpoint covers the whole module: `INSERT` with a column list, multi-row values, defaults and `INSERT ... SELECT`; `UPDATE` with expressions, `CASE` and values from another table; `DELETE` with subqueries, the difference between DELETE, TRUNCATE and DROP, and soft deletes; upserts with `ON DUPLICATE KEY UPDATE`, `REPLACE`, `INSERT IGNORE` and the two-step form; and transactions with `COMMIT`, `ROLLBACK` and savepoints.

How it works. Twelve questions and three scripts. You need 70% on the questions and every script accepted to clear the module. You can retake it as often as you like; your best score counts. Each script changes data and ends with the SELECT that shows the result — the closing SELECT is already in the starter.

Before you start, make sure you can answer these from memory:

- What does a column left out of an INSERT receive, and does an explicit NULL give you the default?
- How do you copy rows from one table into another, and how are the columns matched?
- What does an UPDATE with a CASE and no ELSE do to the rows no branch matches?
- How do you update one table from another in portable SQL, and how does MySQL spell it with a join?
- DELETE, TRUNCATE, DROP: which takes a WHERE, which can be rolled back in MySQL, which resets AUTO_INCREMENT?
- On a duplicate key, what do `ON DUPLICATE KEY UPDATE`, `REPLACE` and `INSERT IGNORE` each do to the existing row?
- What does `ROLLBACK TO SAVEPOINT` undo, and what state is the transaction in afterwards?

The three scripts are a deduplicating import with `INSERT IGNORE`, a month-end close that updates, archives and deletes inside one transaction, and a stock sync that updates the products a delivery mentions and inserts the ones it introduces.

EXERCISE 1

Deduplicate a signup import

Import `signups` into `subscribers` (email is the primary key) with one `INSERT IGNORE ... SELECT`, giving every imported row `source = 'import'`:

- a subscriber who already exists keeps their row exactly as it is;
- when the same email appears several times in `signups`, the signup with the **lowest id** wins.

Then show `email, name, source`, ordered by `email`.

`subscribers`: (ada@x.io, Ada, web).

`signups`:

id	email	name
1	alan@x.io	Alan
2	ada@x.io	Ada L
3	grace@x.io	Grace
4	alan@x.io	Alan T

```
email | name | source
ada@x.io | Ada | web
alan@x.io | Alan | import
grace@x.io | Grace | import
```

INPUT 1

```
CREATE TABLE subscribers (
  email VARCHAR(100) PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  source VARCHAR(20) NOT NULL
);
CREATE TABLE signups (
  id INT PRIMARY KEY,
  email VARCHAR(100) NOT NULL,
  name VARCHAR(50) NOT NULL
);
INSERT INTO subscribers VALUES
('ada@x.io', 'Ada', 'web');
INSERT INTO signups VALUES (1,
'alan@x.io', 'Alan'), (2, 'ada@x.io',
'Ada L'), (3, 'grace@x.io', 'Grace'),
(4, 'alan@x.io', 'Alan T');
```

OUTPUT 1

```
email | name | source
ada@x.io | Ada | web
alan@x.io | Alan | import
grace@x.io | Grace | import
```

REFERENCE SOLUTION

```sql
INSERT IGNORE INTO subscribers (email, name, source)
SELECT email, name, 'import'
FROM signups
ORDER BY id;

SELECT email, name, source
FROM subscribers
ORDER BY email;
```

EXERCISE 2

Close the month

Close May in **one transaction** (start it, do the three steps in this order, commit):

1. every `open` invoice issued **before 2024-06-01** becomes `overdue`;
2. every `paid` invoice is copied into `invoices_archive(id, customer, amount, closed_on)` with `closed_on = '2024-06-30'`;
3. the paid invoices are deleted from `invoices`.

Then print the invoices left (`id, customer, status`, ordered by `id`) and the archive (`id, customer, amount, closed_on`, ordered by `id`; keep any rows it already had).

id	customer	amount	status	issued_on
1	Asha	100.00	paid	2024-05-03
2	Ben	250.00	open	2024-05-20
3	Chen	75.50	open	2024-06-04
4	Dev	40.00	paid	2024-06-10
5	Asha	60.00	overdue	2024-04-01

```
id | customer | status
2 | Ben | overdue
3 | Chen | open
5 | Asha | overdue

id | customer | amount | closed_on
1 | Asha | 100 | 2024-06-30
4 | Dev | 40 | 2024-06-30
```

INPUT 1

```
CREATE TABLE invoices (
  id INT PRIMARY KEY,
  customer VARCHAR(30) NOT NULL,
  amount DECIMAL(10,2) NOT NULL,
  status VARCHAR(10) NOT NULL,
  issued_on DATE NOT NULL
);
CREATE TABLE invoices_archive (
  id INT PRIMARY KEY,
  customer VARCHAR(30) NOT NULL,
  amount DECIMAL(10,2) NOT NULL,
  closed_on DATE NOT NULL
);
INSERT INTO invoices VALUES (1, 'Asha',
100.00, 'paid', '2024-05-03'), (2,
'Ben', 250.00, 'open', '2024-05-20'),
(3, 'Chen', 75.50, 'open', '2024-06-
04'), (4, 'Dev', 40.00, 'paid', '2024-
06-10'), (5, 'Asha', 60.00, 'overdue',
'2024-04-01');
```

OUTPUT 1

```
id | customer | status
2 | Ben | overdue
3 | Chen | open
5 | Asha | overdue

id | customer | amount | closed_on
1 | Asha | 100 | 2024-06-30
4 | Dev | 40 | 2024-06-30
```

REFERENCE SOLUTION

```sql
START TRANSACTION;
UPDATE invoices
SET status = 'overdue'
WHERE status = 'open' AND issued_on < '2024-06-01';

INSERT INTO invoices_archive (id, customer, amount, closed_on)
SELECT id, customer, amount, '2024-06-30'
FROM invoices
WHERE status = 'paid';

DELETE FROM invoices WHERE status = 'paid';
COMMIT;

SELECT id, customer, status FROM invoices ORDER BY id;

SELECT id, customer, amount, closed_on FROM invoices_archive ORDER BY id;
```

EXERCISE 3

Sync stock from a delivery

A delivery arrived, loaded into `delivery(sku, name, qty, price)` with one row per sku. Bring `products` up to date:

- a product the shop already has: add the delivered `qty` to its stock and take the delivery's `price`; keep the product's own `name`;
- a sku the shop has never had: insert it with the delivery's name, qty and price.

Use the portable update-then-insert form. Then show `sku, name, qty, price`, ordered by `sku`.

`products`: (A1, Pen, 10, 1.50), (B2, Pad, 0, 3.00), (C3, Ink, 5, 7.25). `delivery`: (B2, Notepad, 20, 3.40), (D4, Ruler, 15, 2.00), (A1, Pen, 5, 1.50).

```
sku | name | qty | price
A1 | Pen | 15 | 1.5
B2 | Pad | 20 | 3.4
C3 | Ink | 5 | 7.25
D4 | Ruler | 15 | 2
```

INPUT 1

```
CREATE TABLE products (
  sku VARCHAR(10) PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  qty INT NOT NULL,
  price DECIMAL(8,2) NOT NULL
);
CREATE TABLE delivery (
  sku VARCHAR(10) PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  qty INT NOT NULL,
  price DECIMAL(8,2) NOT NULL
);
INSERT INTO products VALUES ('A1',
'Pen', 10, 1.50), ('B2', 'Pad', 0,
3.00), ('C3', 'Ink', 5, 7.25);
INSERT INTO delivery VALUES ('B2',
'Notepad', 20, 3.40), ('D4', 'Ruler',
15, 2.00), ('A1', 'Pen', 5, 1.50);
```

OUTPUT 1

```
sku | name | qty | price
A1 | Pen | 15 | 1.5
B2 | Pad | 20 | 3.4
C3 | Ink | 5 | 7.25
D4 | Ruler | 15 | 2
```

REFERENCE SOLUTION

```sql
UPDATE products
SET qty = qty + (SELECT d.qty FROM delivery d WHERE d.sku = products.sku),
    price = (SELECT d.price FROM delivery d WHERE d.sku = products.sku)
WHERE sku IN (SELECT sku FROM delivery);

INSERT INTO products (sku, name, qty, price)
SELECT d.sku, d.name, d.qty, d.price
FROM delivery d
WHERE NOT EXISTS (SELECT 1 FROM products p WHERE p.sku = d.sku);

SELECT sku, name, qty, price
FROM products
ORDER BY sku;
```

CHECKPOINT QUESTIONS

1. Which statement removes all rows from `logs`, resets its AUTO_INCREMENT and cannot be rolled back in MySQL?

A. `DELETE FROM logs;`

B. `TRUNCATE TABLE logs;`

C. `DROP TABLE logs;`

D. `DELETE FROM logs WHERE 1 = 1;`

2. A column is declared `qty INT NOT NULL DEFAULT 0`. Which INSERT gives it 0?

A. `INSERT INTO t (sku, qty) VALUES ('A', NULL);`

B. `INSERT INTO t (sku) VALUES ('A');`

C. `INSERT INTO t (sku, qty) VALUES ('A', '');`

D. None — DEFAULT applies only to UPDATE

3. `INSERT INTO a (x, y) SELECT y, x FROM b;` — what lands in `a.x`?

A. `b.x`

B. `b.y`

C. NULL

D. The statement fails

4. What does this do to products whose sku is NOT in `changes`?

```
UPDATE products
SET price = (SELECT c.price FROM changes c WHERE c.sku = products.sku);
```

A. Nothing

B. Sets their price to NULL (or fails on NOT NULL)

C. Sets their price to 0

D. Deletes them

5. Which MySQL statement updates `products` from `changes` using a join?

A. `UPDATE products p JOIN changes c ON c.sku = p.sku SET p.price = c.price;`

B. `UPDATE products SET price = c.price FROM changes c WHERE c.sku = products.sku;`

C. `UPDATE JOIN products, changes SET price = c.price;`

D. `MERGE INTO products USING changes ...`

6. Which condition deletes the orders whose customer does not exist, including orders with a NULL customer_id?

A. `WHERE customer_id NOT IN (SELECT id FROM customers)`

B. `WHERE NOT EXISTS (SELECT 1 FROM customers c WHERE c.id = orders.customer_id)`

C. `WHERE customer_id <> ALL (SELECT id FROM customers)`

D. `WHERE customer_id IS NULL`

7. On a duplicate key, which form keeps the existing row's other columns and can add to its counter?

A. `REPLACE INTO`

B. `INSERT IGNORE`

C. `INSERT ... ON DUPLICATE KEY UPDATE`

D. `INSERT ... SELECT`

8. Why does `REPLACE` sometimes delete child rows in other tables?

A. It runs TRUNCATE internally

B. It deletes the conflicting row before inserting, so ON DELETE CASCADE foreign keys fire

C. It drops and recreates the table

D. It never does

9. In MySQL, a transaction runs `UPDATE ...; CREATE TABLE tmp (x INT); ROLLBACK;`. What happens to the UPDATE?

A. It is rolled back

B. It is committed — CREATE TABLE caused an implicit commit

C. It is lost without commit or rollback

D. The CREATE TABLE fails inside a transaction

10. `START TRANSACTION; INSERT A; SAVEPOINT s; INSERT B; ROLLBACK TO SAVEPOINT s; INSERT C; COMMIT;` — which rows are saved?

A. A, B and C

B. A and C

C. Only C

D. None

11. What does MySQL's autocommit mode mean?

A. The server commits every few seconds

B. Each statement outside an explicit transaction is committed as soon as it succeeds

C. Transactions are committed when the connection closes

D. ROLLBACK is disabled

12. Which upsert spelling belongs to PostgreSQL?

A. `INSERT ... ON DUPLICATE KEY UPDATE`

B. `INSERT ... ON CONFLICT (id) DO NOTHING`

C. `INSERT IGNORE`

D. `REPLACE INTO`

ANSWER KEY · CHECKPOINT — MODIFYING DATA

1. B TRUNCATE empties the table by recreating it, resetting the counter, and commits implicitly. DELETE keeps the counter and is transactional; DROP removes the table itself.

2. B Omitting the column applies the default. An explicit NULL is rejected by NOT NULL; an empty string is an error or a converted value, not the default.

3. B INSERT ... SELECT matches by position: the SELECT's first column goes into the first listed column, whatever the names.

4. B With no WHERE every product is updated, and the subquery returns no row — NULL — for the unmatched ones. Add `WHERE sku IN (SELECT sku FROM changes)`.

5. A MySQL's multi-table UPDATE joins in the UPDATE clause. `UPDATE ... FROM` is PostgreSQL; MySQL has no MERGE.

6. B NOT EXISTS is true when nothing matches, and a NULL key matches nothing. `NULL NOT IN (...)` and `NULL <> ALL (...)` are unknown, so those rows survive.

7. C ON DUPLICATE KEY UPDATE modifies the existing row in place. REPLACE deletes and re-inserts; INSERT IGNORE leaves the row unchanged.

8. B REPLACE is a DELETE followed by an INSERT. With InnoDB foreign keys declared ON DELETE CASCADE, the delete removes the children.

9. B DDL in MySQL commits the current transaction before it runs, so the later ROLLBACK has nothing to undo.

10. B Rolling back to the savepoint undoes B only and keeps the transaction open; C is added and COMMIT saves A and C.

11. B Autocommit is on by default; `START TRANSACTION` suspends it until COMMIT or ROLLBACK.

12. B `ON CONFLICT ... DO NOTHING / DO UPDATE` is PostgreSQL's; the other three are MySQL's.