Module 14 of 16

Transactions and concurrency

ACID and the InnoDB machinery behind it, the isolation levels and the dirty, non-repeatable and phantom reads they allow, row, gap and next-key locks, SELECT … FOR UPDATE, deadlocks and optimistic locking, MVCC with undo logs and read views, and savepoints.

A database serves many sessions at once, and every one of them must see data that makes sense: a transfer never half-applied, a report that does not contradict itself, a seat never sold twice. This module explains how MySQL 8's InnoDB engine delivers that — the ACID promises and the logs that keep them, the isolation levels and the anomalies each allows, the locks writers take and the deadlocks they can form, and the multi-version snapshots that let readers and writers work without waiting for each other — with PostgreSQL's different choices alongside, because interviews compare the two.

ACID covers transactions, autocommit, atomicity through the undo log, consistency and constraints, durability through the redo log and `innodb_flush_log_at_trx_commit`, and MySQL's implicit commit on DDL. Isolation levels covers dirty reads, non-repeatable reads and phantoms as schedules, the standard table, InnoDB's `REPEATABLE READ`, lost updates and write skew. Locks and deadlocks covers shared and exclusive locks, `FOR UPDATE`, `FOR SHARE` and `SKIP LOCKED`, gap and next-key locks, lock waits, deadlock detection and the victim, and optimistic locking. MVCC covers row versions, undo chains, read views and purge. Savepoints covers `SAVEPOINT`, `ROLLBACK TO` and `RELEASE`, and how frameworks build nested transactions from them.

The exercise engine runs a single connection, so concurrency is studied the way a DBA studies it after the fact: from schedules, lock-wait snapshots and version tables stored as data. The exercises run real transactions — a transfer, an order placed as one unit, partial undos with savepoints — and analyse concurrency in SQL: dirty and non-repeatable reads in a schedule, deadlock cycles with a recursive CTE, what each MVCC snapshot sees and which versions purge may remove. The checkpoint undoes a fee with a savepoint, runs an optimistic-locking update against a stale writer, and ranks the transactions blocking everyone else.

CONTENTS

1 Transactions and ACID 15 min

2	Isolation levels and read anomalies	16 min
3	Locks and deadlocks	16 min
4	MVCC — how readers and writers avoid each other	15 min
5	Savepoints — partial rollback inside a transaction	13 min
✓	Checkpoint — Transactions and concurrency	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 · 15 MIN

Transactions and ACID

Moving money is the example every interview uses, because it fails so visibly without transactions. Debit ₹500 from Asha, credit ₹500 to Ravi: two `UPDATE`s. If the server crashes between them, ₹500 has vanished. If another session reads the balances between them, it sees money that exists nowhere. A **transaction** groups statements into one unit of work, and the **ACID** properties are the four promises the database makes about that unit. This lesson defines each property precisely, shows how MySQL's InnoDB engine keeps it, and covers the statements that begin and end a transaction.

A transaction in MySQL

```sql
START TRANSACTION;                                  -- or BEGIN

UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;
INSERT INTO transfers (from_id, to_id, amount) VALUES (1, 2, 500);

COMMIT;                                             -- make it permanent
```

Between `START TRANSACTION` and `COMMIT`, the changes are visible to this session but not yet to others, and they can still be abandoned:

```sql
START TRANSACTION;
DELETE FROM accounts;          -- a mistake
ROLLBACK;                      -- every row is back
```

Autocommit. MySQL starts every session with `autocommit = 1`: a statement written outside `START TRANSACTION` is a transaction of its own, committed the moment it finishes. That is why a bare `UPDATE` in a client is permanent immediately. `SET autocommit = 0;` makes every statement join an open transaction until you `COMMIT` — common in some drivers' defaults, and a source of "my changes disappeared" when the connection closes without one.

A is for atomicity

All or nothing. Either every change in the transaction is applied, or none is — whether the transaction ends with `ROLLBACK`, the client disconnects, or the server crashes halfway.

InnoDB implements it with the **undo log**: before changing a row, it records how to put the row back. `ROLLBACK` replays the undo records in reverse; crash recovery does the same for every transaction that was still open.

One MySQL subtlety interviewers probe: when a single statement inside a transaction fails — a duplicate key, say — InnoDB rolls back **that statement**, not the whole transaction. The transaction stays open, and it is your code's job to decide whether to `ROLLBACK` or carry on. (A deadlock is different: it rolls back the entire transaction.) PostgreSQL is stricter: after any error the transaction is aborted and refuses further statements until `ROLLBACK`.

C is for consistency

Valid state to valid state. A transaction starts from data that satisfies every rule and must leave data that does — primary and unique keys, `NOT NULL`, `CHECK` constraints (enforced since MySQL 8.0.16), foreign keys, and the business invariants no constraint can express, such as "the sum of all balances does not change during a transfer".

Consistency is the odd one out: the database enforces the declared constraints, but only the application can make sure each transaction is a correct transformation. If your transfer debits ₹500 and credits ₹50, ACID will faithfully and durably preserve the bug. Atomicity and isolation exist largely so that correct transactions stay correct under crashes and concurrency.

```
CREATE TABLE accounts (
  id      INT PRIMARY KEY,
  owner   VARCHAR(40) NOT NULL,
  balance INT NOT NULL CHECK (balance ⩾ 0)
);
-- a debit that would overdraw now fails, and with it, if the
-- application rolls back, the whole transfer
```

I is for isolation

Concurrent transactions do not see each other's partial work. Ideally each transaction behaves as if it ran alone. In practice, perfect isolation (serializability) costs throughput, so

SQL defines **isolation levels** that allow certain anomalies in exchange for speed. InnoDB's default is `REPEATABLE READ`; PostgreSQL's and SQL Server's is `READ COMMITTED`. InnoDB implements isolation with **MVCC** (multi-version concurrency control: readers see a snapshot built from undo records) and **locks** for writers. Both get a lesson of their own in this module.

D is for durability

Committed means committed. Once `COMMIT` returns, the change survives a crash, a power cut or a restart.

InnoDB uses **write-ahead logging**: each change is written to the **redo log** before the data pages themselves, and at commit the log is flushed to disk. Data pages are written back to the tablespace later, in the background. After a crash, recovery replays the redo log to redo committed work and the undo log to remove uncommitted work. The **doublewrite buffer** protects against a page being half-written during a crash.

The setting behind the promise is `innodb_flush_log_at_trx_commit`:

Value	At each commit	Can lose on a crash
`1` (default)	write and flush the redo log	nothing committed
`2`	write to the OS, flush about once a second	about a second, on an OS crash or power loss
`0`	nothing; write and flush about once a second	about a second, even if only mysqld crashes

Teams sometimes trade durability for write speed with `2`; knowing exactly what is traded is the point. With replication, `sync_binlog = 1` gives the binary log the same promise.

DDL commits implicitly in MySQL

A trap for anyone coming from PostgreSQL: in MySQL, `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, `TRUNCATE TABLE`, `CREATE INDEX` and similar statements cause an **implicit commit** — they commit the open transaction before running, and they cannot be rolled back.

```
START TRANSACTION;
UPDATE accounts SET balance = 0 WHERE id = 1;
CREATE TABLE audit_copy (id INT);    -- implicit COMMIT: the UPDATE is now per
ROLLBACK;                            -- undoes nothing
```

PostgreSQL has **transactional DDL**, so the same script there rolls back both the update and the new table. Migration tools rely on that in PostgreSQL and cannot in MySQL.

Keep transactions short

A transaction holds locks on the rows it changed until it ends, and an old open transaction stops InnoDB from discarding old row versions (you will see why in the MVCC lesson). So:

- do the slow work — calling a payment API, rendering a page, waiting for a user — **outside** the transaction;
- touch rows in a consistent order across code paths;
- commit as soon as the unit of work is complete.

Practise

The exercise engine runs one connection, so isolation cannot be shown here, but atomic units of work can. The first exercise moves money between two accounts inside a transaction and records the transfer; the second places an order, its items and the stock movements as one unit.

EXERCISE 1

A transfer as one unit

Move **500** from account **1** to account **2** and record it in `transfers` with the date `'2024-06-02'` — all inside one transaction (`START TRANSACTION` … `COMMIT`). Then show the balances and the transfer log.

`accounts` :

id	owner	balance
1	Asha	1200
2	Ravi	300
3	Lee	50

`transfers` (`from_id` , `to_id` , `amount` , `made_on`): one earlier row, `(3, 1, 100, '2024-06-01')` .

Output: two result sets — `id` , `owner` , `balance` ordered by `id` ; then `from_id` , `to_id` , `amount` , `made_on` ordered by `made_on` , then `from_id` (both SELECTs are in the starter).

```
id | owner | balance
1 | Asha | 700
2 | Ravi | 800
3 | Lee | 50

from_id | to_id | amount | made_on
3 | 1 | 100 | 2024-06-01
1 | 2 | 500 | 2024-06-02
```

INPUT 1

```
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  owner VARCHAR(40) NOT NULL,
  balance INT NOT NULL
);
INSERT INTO accounts VALUES (1, 'Asha',
1200), (2, 'Ravi', 300), (3, 'Lee', 50);
CREATE TABLE transfers (
  from_id INT NOT NULL,
  to_id INT NOT NULL,
  amount INT NOT NULL,
  made_on DATE NOT NULL
);
INSERT INTO transfers VALUES (3, 1, 100,
'2024-06-01');
```

OUTPUT 1

```
id | owner | balance
1 | Asha | 700
2 | Ravi | 800
3 | Lee | 50

from_id | to_id | amount | made_on
3 | 1 | 100 | 2024-06-01
1 | 2 | 500 | 2024-06-02
```

REFERENCE SOLUTION

```sql
START TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;
INSERT INTO transfers (from_id, to_id, amount, made_on) VALUES (1, 2, 500, '2024-06-

COMMIT;

SELECT id, owner, balance FROM accounts ORDER BY id;
SELECT from_id, to_id, amount, made_on FROM transfers ORDER BY made_on, from_id;
```

EXERCISE 2

Place an order atomically

Placing an order touches three tables, and they must change together. In one transaction:

1. insert order **101** for customer `'meera'` placed on `'2024-05-02'`;
2. insert its items: 2 × product 1 and 1 × product 3;
3. reduce each product's `stock` by the quantity ordered in order 101;
4. commit.

Then show the stock and every order's total units (both SELECTs are in the starter).

`products`: `(1, 'Notebook', 40)`, `(2, 'Pen', 120)`, `(3, 'Backpack', 6)`. `orders`: `(100, 'ravi', '2024-05-01')`. `order_items` (`order_id`, `product_id`, `qty`): `(100, 2, 10)`.

Output: `id`, `name`, `stock` ordered by `id`; then `order_id`, `customer`, `units` ordered by `order_id`.

```
id | name | stock
1 | Notebook | 38
2 | Pen | 120
3 | Backpack | 5

order_id | customer | units
100 | ravi | 10
101 | meera | 3
```

INPUT 1

```
CREATE TABLE products (id INT PRIMARY
KEY, name VARCHAR(40) NOT NULL, stock
INT NOT NULL);
INSERT INTO products VALUES (1,
'Notebook', 40), (2, 'Pen', 120), (3,
'Backpack', 6);
CREATE TABLE orders (id INT PRIMARY KEY,
customer VARCHAR(40) NOT NULL, placed_on
DATE NOT NULL);
INSERT INTO orders VALUES (100, 'ravi',
'2024-05-01');
CREATE TABLE order_items (order_id INT
NOT NULL, product_id INT NOT NULL, qty
INT NOT NULL);
INSERT INTO order_items VALUES (100, 2,
10);
```

OUTPUT 1

```
id | name | stock
1 | Notebook | 38
2 | Pen | 120
3 | Backpack | 5

order_id | customer | units
100 | ravi | 10
101 | meera | 3
```

REFERENCE SOLUTION

```sql
START TRANSACTION;

INSERT INTO orders (id, customer, placed_on) VALUES (101, 'meera', '2024-05-02');
INSERT INTO order_items (order_id, product_id, qty) VALUES (101, 1, 2), (101, 3, 1);

UPDATE products
SET stock = stock - (SELECT i.qty
                     FROM order_items i
                     WHERE i.order_id = 101 AND i.product_id = products.id)
WHERE id IN (SELECT product_id FROM order_items WHERE order_id = 101);

COMMIT;

SELECT id, name, stock FROM products ORDER BY id;
SELECT o.id AS order_id, o.customer, SUM(i.qty) AS units
FROM orders o
JOIN order_items i ON i.order_id = o.id
GROUP BY o.id, o.customer
ORDER BY o.id;
```

QUIZ

1. A transfer debits one account, then the server loses power before the credit runs. Which ACID property guarantees the debit is undone on restart?

A. Consistency

B. Isolation

C. Atomicity

D. Durability

2. Which InnoDB structure makes a committed change survive a crash?

A. The undo log

B. The redo log, flushed at commit

C. The buffer pool

D. The binary log only

3. In MySQL: `START TRANSACTION; UPDATE …; CREATE TABLE t2 (id INT); ROLLBACK;` — what happens to the UPDATE?

A. It is rolled back

B. It is committed: the CREATE TABLE caused an implicit commit

C. The ROLLBACK fails

D. The CREATE TABLE is rolled back too

4. With MySQL's default `autocommit = 1`, what is a single `UPDATE` run outside `START TRANSACTION`?

A. Pending until the session closes

B. Its own transaction, committed as soon as it succeeds

C. Part of the previous transaction

D. Rolled back automatically

5. Inside a MySQL transaction, an `INSERT` fails on a duplicate key. What is the state afterwards?

A. The whole transaction is rolled back

B. Only the failed statement is undone; the transaction is still open with its earlier changes

C. The transaction is aborted and refuses further statements

D. The duplicate row is inserted anyway

ANSWER KEY · TRANSACTIONS AND ACID

1. C Atomicity is all or nothing, even across a crash: InnoDB's recovery rolls back every transaction that never committed, using the undo log. Durability is the promise about **committed** work.

2. B InnoDB writes changes to the redo log before the data pages (write-ahead logging) and, with `innodb_flush_log_at_trx_commit = 1`, flushes it at every commit; recovery replays it. The undo log serves rollback and MVCC.

3. B DDL in MySQL commits the open transaction before it runs and cannot itself be rolled back. PostgreSQL has transactional DDL, where the same script would undo both.

4. B Autocommit wraps each statement in its own transaction. Use `START TRANSACTION` (or `SET autocommit = 0`) to group several statements into one unit.

5. B InnoDB rolls back just the failing statement and leaves the decision to the application. PostgreSQL behaves like the third option: the transaction is aborted until `ROLLBACK`. A deadlock, by contrast, rolls back the whole InnoDB transaction.

LESSON 2 · 16 MIN

Isolation levels and read anomalies

The I in ACID promises that concurrent transactions do not interfere. Taken literally that means **serializability**: the result is the same as if the transactions had run one after another in some order. Serializable execution is expensive, so databases let you choose a weaker **isolation level** that allows specific, named anomalies in exchange for concurrency. This lesson defines the anomalies with concrete schedules, gives the standard table that maps levels to anomalies — the most-asked transaction question in interviews — and explains how MySQL's and PostgreSQL's defaults actually behave.

Schedules

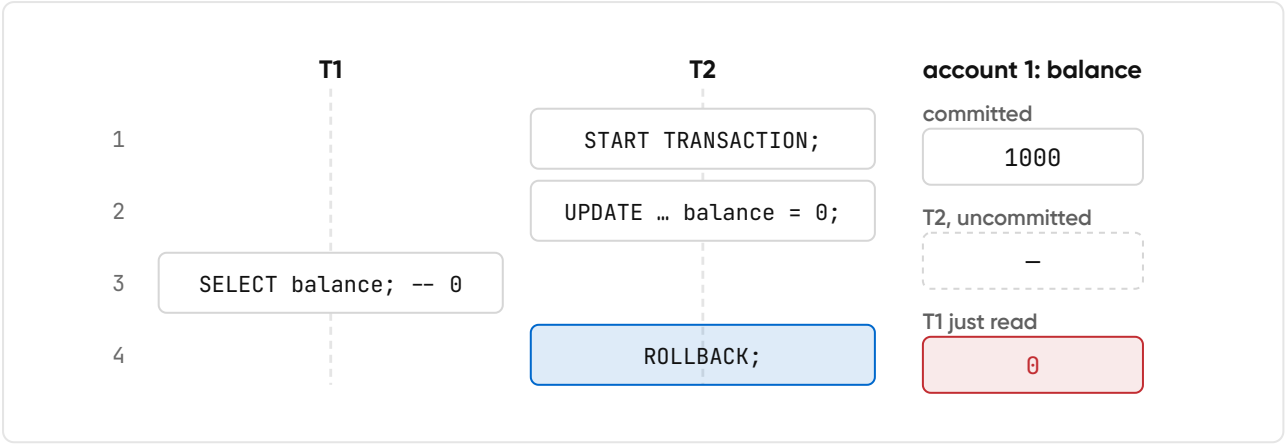
A **schedule** is the interleaved order in which steps of concurrent transactions run. The examples use two sessions, T1 and T2, on this table:

```
CREATE TABLE accounts (id INT PRIMARY KEY, owner VARCHAR(40), balance INT);
-- id 1, Asha, 1000
```

Dirty read

T1 reads a value T2 has **written but not committed**.

A dirty read: reading a write that is rolled back



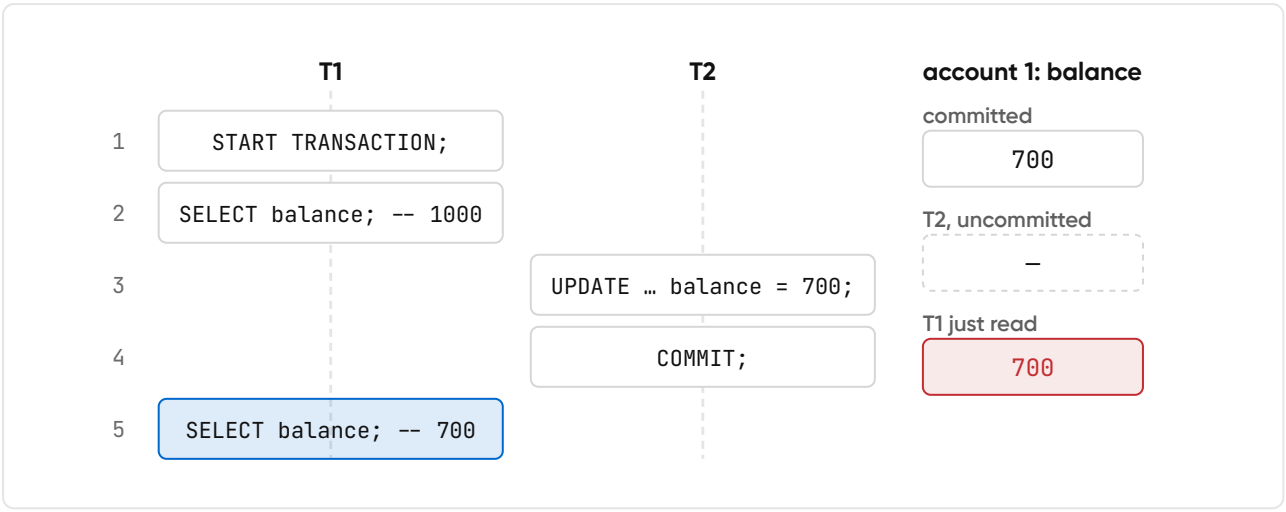
1. T2 opens a transaction on Asha's account, whose committed balance is 1000. T1 runs under READ UNCOMMITTED.
2. T2 sets the balance to 0 but has not committed. The committed value is still 1000; the 0 exists only inside T2.
3. T1 reads the newest version, committed or not, and sees 0: a dirty read of a write that may never happen.
4. T2 rolls back. The balance was 1000 all along in every committed state, so the 0 T1 acted on never existed. READ COMMITTED and above prevent this.

T1 has read a balance of 0 that never existed in any committed state. If T1 refused a loan because of it, the decision was based on fiction.

Non-repeatable read

T1 reads the **same row twice** and gets different values, because T2 updated it and **committed** in between.

A non-repeatable read: the same row, two answers



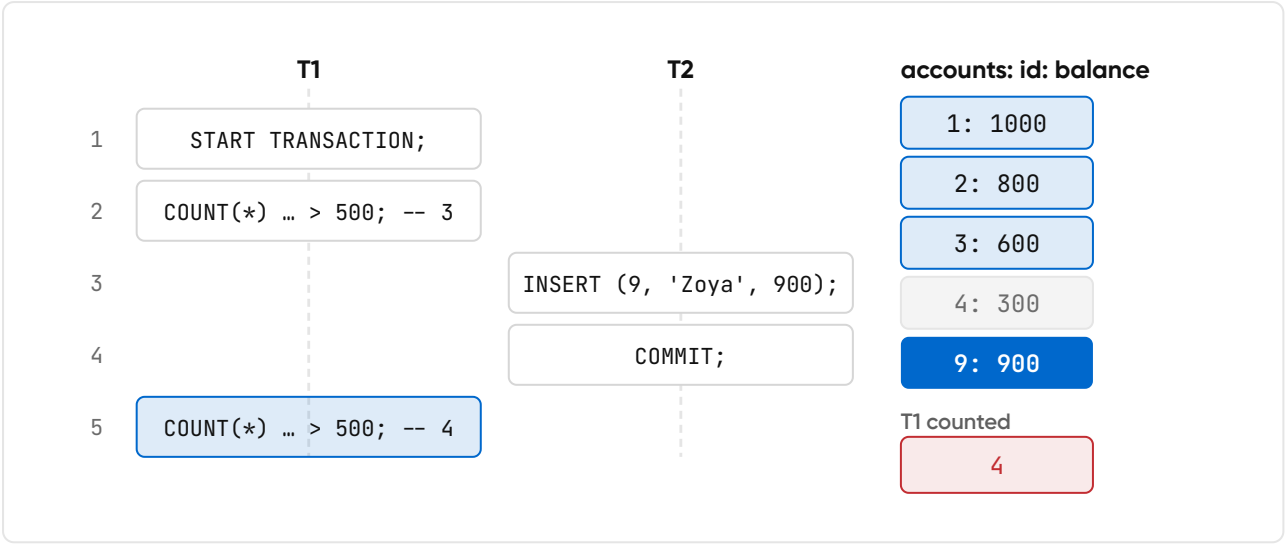
1. T1 starts a transaction under READ COMMITTED, where every statement reads the latest committed data.
2. T1's first read sees the committed 1000.
3. T2 updates the balance to 700. Uncommitted, so T1 could not see it even if it read now.
4. T2 commits: 700 is now the committed balance.
5. T1 reads the same row again and gets 700, not 1000. Nothing dirty was read, yet one transaction saw two values — REPEATABLE READ keeps one snapshot and prevents it.

Nothing uncommitted was read, yet T1's view of the world changed under it — a report that reads the balance in two places would contradict itself.

Phantom read

T1 runs the **same query twice** and gets a different **set of rows**, because T2 inserted (or deleted) rows that match and committed.

A phantom read: the same query, a new row



1. T1 starts. Of the 4 accounts, 3 hold more than 500.
2. T1 counts the accounts over 500: 3.
3. T2 inserts Zoya's account with 900 — a row that matches T1's condition but did not exist when T1 read. Not yet committed.
4. T2 commits, and the new row is part of the committed data.
5. T1 runs the same query and gets 4. No row it saw changed; a new one appeared — a phantom. Locking the rows read cannot stop it; preventing it means locking the range, gaps included.

No row T1 had seen changed; a new one appeared. Locking the rows you read cannot stop this, because the phantom did not exist when you read — preventing it means locking the **range**, gaps included.

The standard table

Isolation level	Dirty read	Non-repeatable read	Phantom read
`READ UNCOMMITTED`	possible	possible	possible
`READ COMMITTED`	prevented	possible	possible
`REPEATABLE READ`	prevented	prevented	possible (by the standard)
`SERIALIZABLE`	prevented	prevented	prevented

Learn it row by row: each level forbids one more anomaly than the one above. "Possible" means the standard allows it; a particular engine may do better.

Setting the level in MySQL

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;          -- the next transact
START TRANSACTION;
-- …
COMMIT;

SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- this connection f
SELECT @@transaction_isolation;                          -- 'REPEATABLE-READ'
```

The server-wide default comes from the `transaction_isolation` system variable. (`tx_isolation` was its name until MySQL 5.7 and is gone in 8.0.)

How InnoDB actually behaves

`READ UNCOMMITTED` reads the newest version of every row, committed or not. Almost never used.

`READ COMMITTED` gives every **statement** a fresh snapshot of committed data: no dirty reads, but two statements in one transaction can disagree. Locking reads and updates lock only the rows they touch (no gap locks), so concurrency is higher. Many teams running MySQL with row-based replication choose it.

`REPEATABLE READ` — the **InnoDB default** — takes one snapshot at the transaction's **first read** and uses it for every plain `SELECT` until commit. Repeated reads agree, and a repeated plain `SELECT` does not see phantoms either, because it reads the same snapshot. Locking statements (`SELECT … FOR UPDATE`, `UPDATE`, `DELETE`) work differently: they read the **latest committed** row versions and take **next-key locks** — locks on the index records *and the gaps between them* — so no other transaction can insert into the scanned range until commit. That is how InnoDB closes the phantom hole the standard leaves open for this level, for locking reads.

The mix has a famous edge: within one `REPEATABLE READ` transaction, a plain `SELECT` may say a row does not exist while an `UPDATE` of it succeeds, because the update reads the latest version, not the snapshot.

`SERIALIZABLE` in InnoDB is `REPEATABLE READ` plus one rule: when autocommit is off, every plain `SELECT` becomes `SELECT … FOR SHARE`, so reads take shared locks and conflicting writers wait. Correct, and much more prone to waits and deadlocks.

PostgreSQL, for comparison

- The default is `READ COMMITTED`, as in Oracle and SQL Server.
- `READ UNCOMMITTED` behaves as `READ COMMITTED`: PostgreSQL never shows dirty data.
- `REPEATABLE READ` is **snapshot isolation**: no phantoms at all, and if two transactions update the same row the second fails with a serialization error that the application must retry — where InnoDB would let it overwrite the newer version.
- `SERIALIZABLE` uses **serializable snapshot isolation (SSI)**: no read locks, but it detects dangerous patterns and aborts one transaction.

Beyond the table: lost updates and write skew

The three classic anomalies are not the whole story. Two more come up in system-design interviews:

- **Lost update**: T1 and T2 both read a stock of 10, both write `10 - 1`, and one sale disappears. Fix it with an atomic `UPDATE … SET stock = stock - 1`, a locking read (`SELECT … FOR UPDATE`), or a version check (optimistic locking, next lesson).
- **Write skew**: two doctors are on call; each transaction checks "at least one other doctor is on call" and takes itself off. Each read was valid, they wrote different rows, and now nobody is on call. Snapshot isolation allows it; `SERIALIZABLE` or locking the rows you based the decision on prevents it.

Choosing a level

Most applications run fine on their database's default and use **explicit locking or atomic updates** for the few places that need stronger guarantees — a seat booking, a balance check. Raise the level for a whole workload only when you understand the extra waits and retries it brings.

Practise

Concurrency cannot run on a single connection, but schedules can be stored and analysed. Both exercises read a `schedule` table — one row per step: which transaction, what action,

which row, which value — and detect anomalies in SQL: the first finds dirty reads, the second non-repeatable reads.

EXERCISE 1

Spot the dirty reads

`schedule` records an interleaving of transactions, one row per step: `action` is `begin`, `read`, `write`, `commit` or `rollback`; `item` and `seen_value` are set on reads and writes. A read is **dirty** when another transaction wrote the same item earlier and had neither committed nor rolled back by the time of the read. List every dirty read with the transaction it read from.

`schedule`:

step	txn	action	item	seen_value
1	T1	begin	NULL	NULL
2	T2	begin	NULL	NULL
3	T2	write	A	0
4	T1	read	A	0
5	T2	rollback	NULL	NULL
6	T1	read	A	1000
7	T3	begin	NULL	NULL
8	T3	write	B	50
9	T3	commit	NULL	NULL
10	T1	read	B	50
11	T1	commit	NULL	NULL

Output columns: `step` (of the read), `reader`, `item`, `writer` — one row per distinct read and writer — ordered by `step`, then `writer`.

```
step | reader | item | writer
4 | T1 | A | T2
```

INPUT 1

```
CREATE TABLE schedule (
  step INT PRIMARY KEY,
  txn VARCHAR(5) NOT NULL,
  action VARCHAR(10) NOT NULL,
  item VARCHAR(5),
  seen_value INT
);
INSERT INTO schedule VALUES
  (1, 'T1', 'begin', NULL, NULL),
  (2, 'T2', 'begin', NULL, NULL),
  (3, 'T2', 'write', 'A', 0),
  (4, 'T1', 'read', 'A', 0),
  (5, 'T2', 'rollback', NULL, NULL),
  (6, 'T1', 'read', 'A', 1000),
  (7, 'T3', 'begin', NULL, NULL),
  (8, 'T3', 'write', 'B', 50),
  (9, 'T3', 'commit', NULL, NULL),
  (10, 'T1', 'read', 'B', 50),
  (11, 'T1', 'commit', NULL, NULL);
```

OUTPUT 1

```
step | reader | item | writer
4 | T1 | A | T2
```

REFERENCE SOLUTION

```sql
SELECT DISTINCT r.step, r.txn AS reader, r.item, w.txn AS writer
FROM schedule r
JOIN schedule w
  ON w.action = 'write'
 AND w.item = r.item
 AND w.txn <> r.txn
 AND w.step < r.step
WHERE r.action = 'read'
  AND NOT EXISTS (
    SELECT 1
    FROM schedule e
    WHERE e.txn = w.txn
      AND e.action IN ('commit', 'rollback')
      AND e.step < r.step
  )
ORDER BY r.step, writer;
```

EXERCISE 2

Find the non-repeatable reads

Using the same kind of `schedule` table, find **non-repeatable reads**: a transaction read the same item more than once and saw more than one value, **without having written that item itself** (seeing your own write is not an anomaly).

`schedule`:

step	txn	action	item	seen_value
1	T1	begin	NULL	NULL
2	T1	read	A	1000
3	T2	begin	NULL	NULL
4	T2	write	A	700
5	T2	commit	NULL	NULL
6	T1	read	A	700
7	T1	read	B	40
8	T3	begin	NULL	NULL
9	T3	read	B	40
10	T3	write	B	45
11	T3	read	B	45
12	T3	commit	NULL	NULL
13	T1	read	B	45
14	T1	read	A	700
15	T1	commit	NULL	NULL

Output columns: `txn`, `item`, `times_read` (how many times it read the item), `distinct_values`, ordered by `txn`, then `item`.

```
txn | item | times_read | distinct_values
T1 | A | 3 | 2
T1 | B | 2 | 2
```

INPUT 1

```
CREATE TABLE schedule (
  step INT PRIMARY KEY,
  txn VARCHAR(5) NOT NULL,
  action VARCHAR(10) NOT NULL,
  item VARCHAR(5),
  seen_value INT
);
INSERT INTO schedule VALUES
  (1, 'T1', 'begin', NULL, NULL),
  (2, 'T1', 'read', 'A', 1000),
  (3, 'T2', 'begin', NULL, NULL),
  (4, 'T2', 'write', 'A', 700),
  (5, 'T2', 'commit', NULL, NULL),
  (6, 'T1', 'read', 'A', 700),
  (7, 'T1', 'read', 'B', 40),
  (8, 'T3', 'begin', NULL, NULL),
  (9, 'T3', 'read', 'B', 40),
  (10, 'T3', 'write', 'B', 45),
  (11, 'T3', 'read', 'B', 45),
  (12, 'T3', 'commit', NULL, NULL),
  (13, 'T1', 'read', 'B', 45),
  (14, 'T1', 'read', 'A', 700),
  (15, 'T1', 'commit', NULL, NULL);
```

OUTPUT 1

```
txn | item | times_read |
distinct_values
T1 | A | 3 | 2
T1 | B | 2 | 2
```

REFERENCE SOLUTION

```sql
SELECT r.txn,
       r.item,
       COUNT(*) AS times_read,
       COUNT(DISTINCT r.seen_value) AS distinct_values
FROM schedule r
WHERE r.action = 'read'
  AND NOT EXISTS (
    SELECT 1
    FROM schedule w
    WHERE w.action = 'write'
      AND w.txn = r.txn
      AND w.item = r.item
  )
GROUP BY r.txn, r.item
HAVING COUNT(DISTINCT r.seen_value) > 1
ORDER BY r.txn, r.item;
```

QUIZ

1. Which isolation level is the **lowest** that prevents dirty reads?

A. READ UNCOMMITTED

B. READ COMMITTED

C. REPEATABLE READ

D. SERIALIZABLE

2. T1 runs `SELECT COUNT(*) FROM orders WHERE status = 'new'` twice and gets 5, then 6, because T2 inserted a new order and committed between. Which anomaly is this?

A. Dirty read

B. Non-repeatable read

C. Phantom read

D. Lost update

3. What is the default isolation level of MySQL's InnoDB, and of PostgreSQL?

A. Both READ COMMITTED

B. InnoDB REPEATABLE READ; PostgreSQL READ COMMITTED

C. InnoDB SERIALIZABLE; PostgreSQL REPEATABLE READ

D. Both REPEATABLE READ

4. Under InnoDB's REPEATABLE READ, when is the snapshot for plain SELECTs taken?

A. At `START TRANSACTION`, always

B. At the transaction's first consistent read, then reused until commit

C. At every statement

D. At commit

5. Two transactions each read that two doctors are on call and each takes a **different** doctor off call; now nobody is. Which anomaly is this, and what prevents it?

A. Dirty read; READ COMMITTED

B. Write skew; SERIALIZABLE or locking the rows read with FOR UPDATE

C. Phantom read; REPEATABLE READ

D. Lost update; autocommit

ANSWER KEY · ISOLATION LEVELS AND READ ANOMALIES

1. B READ COMMITTED shows only committed data. It still allows non-repeatable reads and phantoms; REPEATABLE READ and SERIALIZABLE forbid more.

2. C A new row matching the predicate appeared — a phantom. A non-repeatable read is an existing row whose values changed; a dirty read involves uncommitted data.

3. B InnoDB defaults to REPEATABLE READ; PostgreSQL, Oracle and SQL Server default to READ COMMITTED — a classic comparison question.

4. B The read view is created at the first consistent read (or immediately with `START TRANSACTION WITH CONSISTENT SNAPSHOT`) and reused. Under READ COMMITTED a fresh one is taken for each statement.

5. B Write skew: both decisions were based on a valid snapshot, but they wrote different rows, so no write–write conflict occurred. Snapshot isolation allows it; serializable execution or explicit locks of the rows the decision depends on prevent it.

LESSON 3 · 16 MIN

Locks and deadlocks

MVCC lets readers and writers stay out of each other's way, but two **writers** to the same row must take turns, and some readers need to block writers too. That is the job of **locks**. Most of the time they are invisible; then one day a checkout page hangs for fifty seconds, or the logs fill with `Deadlock found when trying to get lock`. This lesson covers InnoDB's lock types, locking reads, what a lock wait and a deadlock look like, how InnoDB breaks a deadlock, and the habits that keep both rare — plus the optimistic alternative to locking at all.

Shared and exclusive row locks

InnoDB locks **rows** (precisely: index records), not whole tables, for ordinary DML.

- A **shared (S) lock** lets the holder read the row; other transactions may also take S locks but not X.
- An **exclusive (X) lock** lets the holder change the row; nobody else can take S or X on it.

Held \ Requested	S	X
S	granted	waits
X	waits	waits

`UPDATE` and `DELETE` take X locks on every row they change, held until `COMMIT` or `ROLLBACK`. Plain `SELECT` takes **no** locks under the default isolation levels — it reads a snapshot (the next lesson). Table-level **intention locks** (IS, IX) are taken automatically to say "this transaction holds row locks in here", so that a `LOCK TABLES` or DDL knows to wait.

Locking reads

When you read a value in order to write based on it, a snapshot read is not enough — another transaction could change the row after you read it. Ask for a lock while reading:

```sql
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1 FOR UPDATE;   -- X lock on row 1
-- application checks balance ⩾ 500
UPDATE accounts SET balance = balance - 500 WHERE id = 1;
COMMIT;
```

- `FOR UPDATE` takes X locks: others cannot change or lock the rows until you finish.
- `FOR SHARE` (MySQL 8; `LOCK IN SHARE MODE` before) takes S locks: others can read-lock but not change them — useful for "this parent row must still exist when I insert the child".
- `NOWAIT` fails at once instead of waiting if a row is locked.
- `SKIP LOCKED` skips locked rows — the standard way to build a job queue where several workers each grab a different pending job:

```sql
SELECT id FROM jobs
WHERE status = 'pending'
ORDER BY id
LIMIT 1
FOR UPDATE SKIP LOCKED;
```

Gap locks and next-key locks

Locking existing rows cannot stop a **new** row being inserted into the range you queried — the phantom problem. Under `REPEATABLE READ`, InnoDB's locking reads and updates therefore lock **gaps** too:

- a **record lock** locks one index record;
- a **gap lock** locks the open interval between two index records, so nothing can be inserted there;
- a **next-key lock** is a record lock plus the gap before it — the default for range scans under `REPEATABLE READ`;
- an **insert intention lock** is the gap-level lock an `INSERT` takes; it waits behind another transaction's gap lock.

```sql
-- REPEATABLE READ; index on age; existing ages 20, 30, 40
SELECT * FROM members WHERE age BETWEEN 25 AND 35 FOR UPDATE;
-- locks record 30 and the gaps (20, 30) and (30, 40):
-- another session's INSERT of age 27 or 33 now waits
```

An equality lookup on a **unique** index for a row that exists locks only that record — no gap. Under `READ COMMITTED`, gap locking is disabled for searches, which is one reason it has fewer deadlocks.

Locks follow the index

InnoDB locks the index records a statement **scans**, not just the rows it finally changes. Without a usable index:

```
UPDATE orders SET status = 'cancelled' WHERE customer_email = 'x@y.com';
-- no index on customer_email: every row is scanned, and under
-- REPEATABLE READ every row (and gap) is locked until commit
```

One missing index can turn a one-row update into a table-wide lock. Indexing the `WHERE` columns of your updates is a concurrency fix, not just a speed fix.

Lock waits

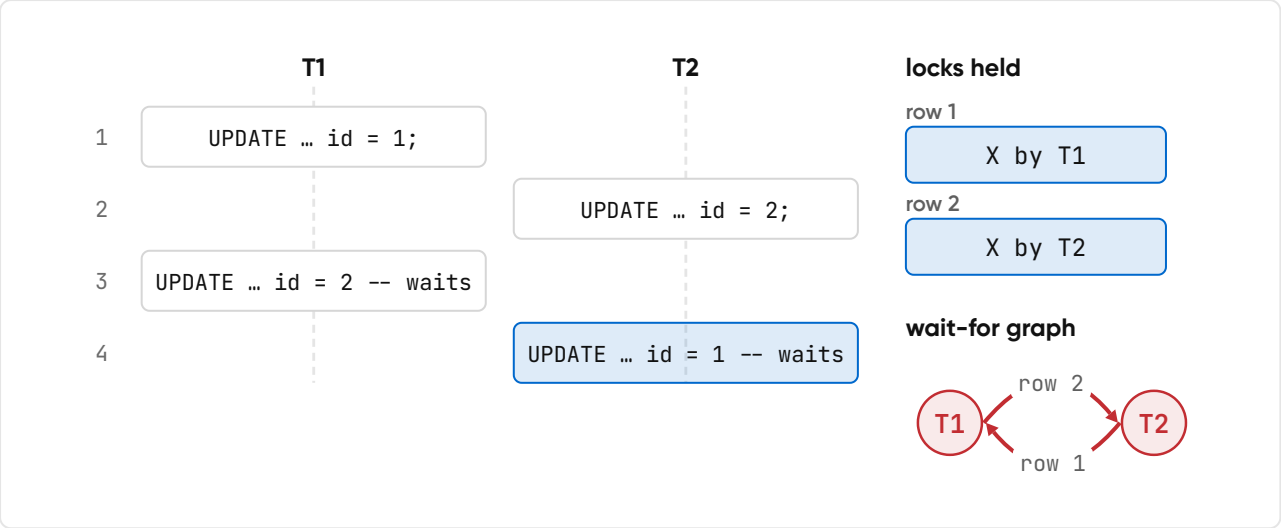
When a lock is unavailable, the transaction **waits**. If it waits longer than `innodb_lock_wait_timeout` (50 seconds by default), the statement fails with **error 1205**, `Lock wait timeout exceeded; try restarting transaction`. By default only that statement is rolled back, not the whole transaction (`innodb_rollback_on_timeout` changes that).

To see who blocks whom in MySQL 8, query `performance_schema.data_locks` and `performance_schema.data_lock_waits`, or the friendlier `sys.innodb_lock_waits` view, which pairs each waiting transaction with its blocker. The usual culprit is a transaction someone left open.

Deadlocks

A **deadlock** is a cycle of waits:

A deadlock: two transactions, each waiting for the other



1. Both transactions have started. T1 updates row 1 and takes an exclusive lock on it, held until it commits.
2. T2 updates row 2 and takes its exclusive lock. So far there is no conflict: each holds a different row.
3. T1 now wants row 2, which T2 holds, so T1 waits — an edge T1 → T2 in the wait-for graph. A wait alone is normal; it ends when T2 commits.
4. T2 wants row 1, which T1 holds: an edge T2 → T1 closes a cycle, and neither can ever proceed. InnoDB sees it at once, rolls back the transaction it judges smaller with error 1213, and the other goes on.

Neither can ever proceed. InnoDB's **deadlock detector** (on by default, `innodb_deadlock_detect`) builds the **wait-for graph** — an edge from each waiting transaction to the transaction holding what it wants — and when a new wait closes a cycle it immediately picks a **victim**: the transaction it judges smallest, measured by the number of rows it has inserted, updated or deleted. The victim is **rolled back entirely** with **error 1213**, `Deadlock found when trying to get lock; try restarting transaction`, and the other transaction continues. `SHOW ENGINE INNODB STATUS` prints the latest detected deadlock, with both transactions' statements and locks; `innodb_print_all_deadlocks` writes every one to the error log.

Deadlocks are normal in a busy OLTP system. The rule is: **expect them, retry them, and make them rare**.

Making deadlocks rare

- **Same order everywhere.** If every transfer locks the lower account id first, the cycle above cannot form.
- **Short transactions.** Fewer locks held for less time.
- **Good indexes.** Statements lock what they scan; scan less.

- **Lock up front.** `SELECT … FOR UPDATE` the rows you will change at the start, in order, rather than escalating midway.
- **Retry on 1213** in application code — the transaction was rolled back cleanly, so running it again is safe.
- Consider `READ COMMITTED` when gap locks cause the cycles.

Optimistic locking: no lock at all

Pessimistic locking (`FOR UPDATE`) assumes conflict and blocks. **Optimistic locking** assumes conflict is rare: read a version number with the row, and make the write conditional on it.

```
SELECT id, price, version FROM products WHERE id = 7;      -- version = 3

UPDATE products
SET price = 549, version = version + 1
WHERE id = 7 AND version = 3;          -- 1 row changed: you won
-- 0 rows changed: someone updated it first — reload and retry, or tell the us
```

No lock is held while the user edits the form, and a lost update becomes impossible. ORMs implement exactly this (JPA's `@Version`, Rails' `lock_version`). Pessimistic wins for hot rows with frequent conflicts; optimistic wins for long think-times and rare conflicts.

Practise

The exercises analyse a `lock_waits` table — a snapshot of who waits for whom, as `sys.innodb_lock_waits` would show it. The first finds pairs of transactions waiting on each other and names the victim InnoDB would choose; the second uses a recursive CTE to find every transaction caught in a cycle of any length.

EXERCISE 1

Two-way deadlocks and their victim

`lock_waits` is a snapshot of which transaction waits for which (one row per waited-for lock), and `trx` holds how many rows each transaction has modified. Find every pair of transactions that **wait for each other** — a two-way deadlock — and the victim InnoDB would roll back: the one with **fewer** `rows_modified`, or, on a tie, the one with the larger id.

`lock_waits`:

waiting_txn	blocking_txn	resource
101	102	accounts:id=2
102	101	accounts:id=1
103	101	accounts:id=1
104	105	orders:id=77
105	106	orders:id=78
106	104	orders:id=76

`trx`: `(101, 4)`, `(102, 1)`, `(103, 2)`, `(104, 9)`, `(105, 3)`, `(106, 6)`.

Output columns: `txn_a` (the smaller id), `txn_b`, `victim` — one row per pair — ordered by `txn_a`, then `txn_b`. (The three-way cycle 104 → 105 → 106 → 104 is not a pair; the next exercise finds it.)

```
txn_a | txn_b | victim
101 | 102 | 102
```

INPUT 1

```
CREATE TABLE lock_waits (
  waiting_txn INT NOT NULL,
  blocking_txn INT NOT NULL,
  resource VARCHAR(40) NOT NULL
);
INSERT INTO lock_waits VALUES
  (101, 102, 'accounts:id=2'),
  (102, 101, 'accounts:id=1'),
  (103, 101, 'accounts:id=1'),
  (104, 105, 'orders:id=77'),
  (105, 106, 'orders:id=78'),
  (106, 104, 'orders:id=76');
CREATE TABLE trx (txn_id INT PRIMARY
KEY, rows_modified INT NOT NULL);
INSERT INTO trx VALUES
  (101, 4), (102, 1), (103, 2), (104,
9), (105, 3), (106, 6);
```

OUTPUT 1

```
txn_a | txn_b | victim
101 | 102 | 102
```

REFERENCE SOLUTION

```sql
SELECT DISTINCT
       w1.waiting_txn AS txn_a,
       w1.blocking_txn AS txn_b,
       CASE
         WHEN ta.rows_modified < tb.rows_modified THEN ta.txn_id
         ELSE tb.txn_id
       END AS victim
FROM lock_waits w1
JOIN lock_waits w2
  ON w2.waiting_txn = w1.blocking_txn
 AND w2.blocking_txn = w1.waiting_txn
JOIN trx ta ON ta.txn_id = w1.waiting_txn
JOIN trx tb ON tb.txn_id = w1.blocking_txn
WHERE w1.waiting_txn < w1.blocking_txn
ORDER BY txn_a, txn_b;
```

EXERCISE 2

Every transaction in a cycle

Deadlocks can involve more than two transactions. Using the same `lock_waits` table, follow the wait-for edges with a **recursive CTE** and list every transaction that can reach itself — it is part of a deadlock cycle — with the length of the shortest cycle it is in (the number of wait edges).

Output columns: `txn`, `cycle_length`, ordered by `txn`. With the data of the previous exercise:

```
txn | cycle_length
101 | 2
102 | 2
104 | 3
105 | 3
106 | 3
```

Transaction 103 waits for a deadlocked transaction but is not in a cycle itself. Stop the recursion at a depth of 10.

INPUT 1

```
CREATE TABLE lock_waits (
  waiting_txn INT NOT NULL,
  blocking_txn INT NOT NULL,
  resource VARCHAR(40) NOT NULL
);
INSERT INTO lock_waits VALUES
  (101, 102, 'accounts:id=2'),
  (102, 101, 'accounts:id=1'),
  (103, 101, 'accounts:id=1'),
  (104, 105, 'orders:id=77'),
  (105, 106, 'orders:id=78'),
  (106, 104, 'orders:id=76');
CREATE TABLE trx (txn_id INT PRIMARY
KEY, rows_modified INT NOT NULL);
INSERT INTO trx VALUES
  (101, 4), (102, 1), (103, 2), (104,
9), (105, 3), (106, 6);
```

OUTPUT 1

```
txn | cycle_length
101 | 2
102 | 2
104 | 3
105 | 3
106 | 3
```

REFERENCE SOLUTION

```sql
WITH RECURSIVE chain (start_txn, current_txn, depth) AS (
  SELECT waiting_txn, blocking_txn, 1
  FROM lock_waits
  UNION ALL
  SELECT c.start_txn, w.blocking_txn, c.depth + 1
  FROM chain c
  JOIN lock_waits w ON w.waiting_txn = c.current_txn
  WHERE c.depth < 10
)
SELECT start_txn AS txn, MIN(depth) AS cycle_length
FROM chain
WHERE current_txn = start_txn
GROUP BY start_txn
ORDER BY txn;
```

QUIZ

1. What does `SELECT balance FROM accounts WHERE id = 1 FOR UPDATE` do inside a transaction?

A. Reads the snapshot without locking

B. Reads the latest committed row and holds an exclusive lock on it until the transaction ends

C. Updates the row

D. Locks the whole table

2. Under REPEATABLE READ, `SELECT … WHERE age BETWEEN 25 AND 35 FOR UPDATE` uses an index on `age`. Why does another session's `INSERT` of age 30 wait?

A. The table is locked

B. Next-key locks cover the gaps in the scanned range, blocking inserts into it

C. INSERT always waits for SELECT

D. Because of MVCC

3. How does InnoDB resolve a deadlock?

A. Waits for `innodb_lock_wait_timeout` and then fails both

B. Detects the cycle at once and rolls back one transaction — usually the one that modified fewer rows — with error 1213

C. Kills the connection that started first

D. Converts the locks to shared locks

4. Which practice prevents the classic two-account transfer deadlock?

A. Use READ UNCOMMITTED

B. Always lock (update) the accounts in the same order, such as lower id first

C. Add more indexes

D. Use autocommit

5. Several workers poll a `jobs` table for pending work. Which clause lets each grab a different job without waiting on the others?

A. `FOR SHARE`

B. `FOR UPDATE NOWAIT`

C. `FOR UPDATE SKIP LOCKED`

D. `LOCK IN SHARE MODE`

ANSWER KEY · LOCKS AND DEADLOCKS

1. B A locking read takes an X lock on the matched index records; other transactions that try to change or lock row 1 wait. With a primary-key lookup only that one record is locked.

2. B InnoDB locks the scanned records and the gaps between them so no phantom can appear in the range. Under READ COMMITTED these gap locks are not taken, and the insert would proceed.

3. B With `innodb_deadlock_detect` on (the default), InnoDB checks the wait-for graph on every new wait and rolls back the smaller transaction entirely. The timeout (error 1205) is for plain waits, not cycles.

4. B A cycle needs two transactions acquiring the same locks in opposite orders. A global order makes the second transaction wait for the first instead of forming a cycle.

5. C `SKIP LOCKED` (MySQL 8) skips rows another transaction has locked, so each worker takes the next free job. `NOWAIT` would fail instead of moving on.

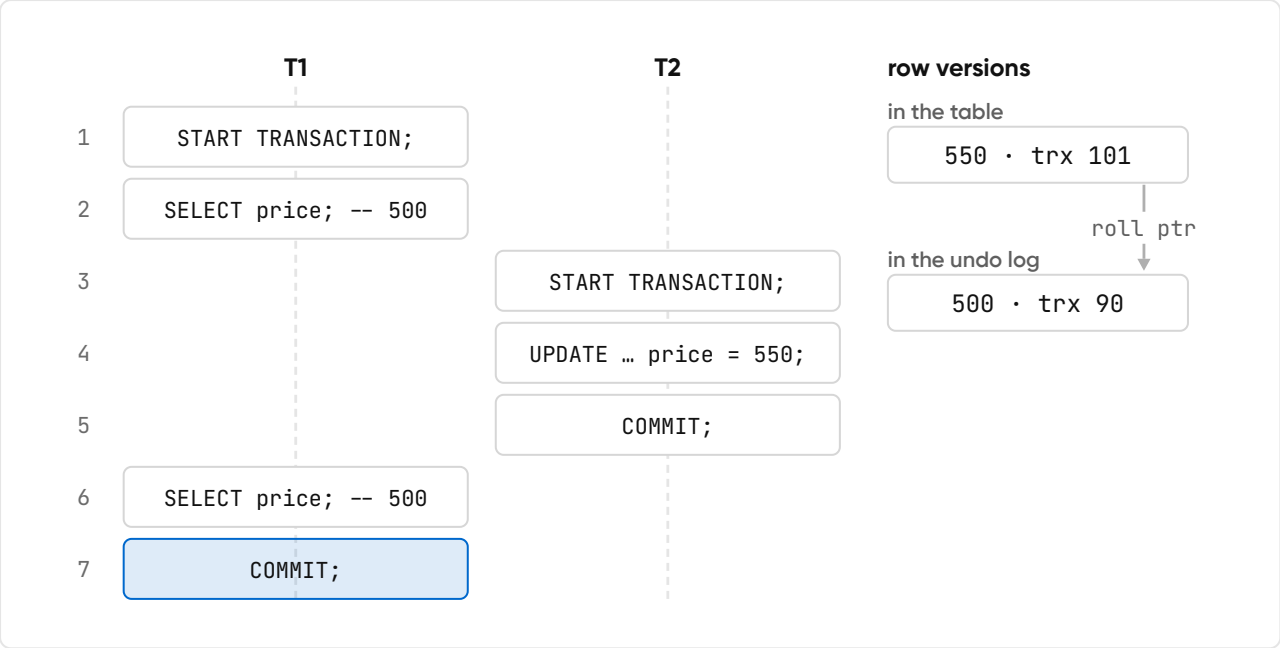
LESSON 4 · 15 MIN

MVCC — how readers and writers avoid each other

If every write took locks that blocked readers, a busy table would spend its life waiting: a report scanning the orders table would stall every checkout, and every checkout would stall the report. Modern databases avoid that with **multi-version concurrency control (MVCC)**: a change creates a new version of the row while the old one is kept for transactions that should still see it. Readers read a **snapshot**; writers write new versions; neither waits for the other. This lesson explains how InnoDB stores versions, how a read decides which version it may see, how isolation levels map onto snapshots, and why long transactions hurt.

The idea in one picture

MVCC: a reader keeps its snapshot while a writer commits



1. T1 starts and is given transaction id 100. The price row holds one version, 500, written by transaction 90, which committed long ago.
2. T1's first read makes its read view: transaction 90 had committed, 100 is itself, and anything numbered from 101 on did not exist yet. The newest version passes, so T1 reads 500.
3. T2 starts with id 101. Nothing has changed for T1: no lock was taken by its read, and its view is already fixed.
4. T2's UPDATE writes a new version, 550 by transaction 101, into the table, and keeps 500 in the undo log behind the roll pointer. Readers are not blocked; they can still reach the old version.
5. T2 commits. 550 is now the latest committed price — but T1's read view was made before 101 committed, and a view never changes.
6. T1 reads again. The newest version is by 101, not in its view, so it follows the roll pointer to 500 by transaction 90, which its view does include, and reads 500 again. Neither transaction ever waited for the other.
7. T1 commits and its view is dropped. A transaction reading from now on would see 550; once no read view needs 500, InnoDB's purge discards it from the undo log.

T1 never waits for T2's lock and never sees T2's change: it keeps reading version 1, because version 2 did not exist — committed — when T1's snapshot was taken. T2 never waits for T1 either. Only two **writers** to the same row conflict, and they are ordered by locks.

How InnoDB stores versions

Each row in the clustered index carries hidden system columns:

- `DB_TRX_ID` (6 bytes) — the id of the transaction that last inserted or updated it;
- `DB_ROLL_PTR` (7 bytes) — a pointer to an **undo log record** holding what the row looked like before that change;
- `DB_ROW_ID` — the hidden row id, used only when the table has no suitable key.

An `UPDATE` changes the row **in place** and writes the previous values to the undo log, linked by the roll pointer. Repeated updates form a **version chain**: newest in the table, older ones reachable record by record through the undo log. A `DELETE` does not remove the row at once; it sets a delete mark, and the row is physically removed later.

The same undo records serve two purposes: `ROLLBACK` uses them to restore rows (atomicity), and consistent reads use them to rebuild old versions (isolation).

Read views: which version may I see?

A consistent read (any plain `SELECT` under `READ COMMITTED` or `REPEATABLE READ`) uses a **read view**, created from the state of the transaction system at that moment:

- the list of transaction ids that are **active** (started, not committed);
- the lowest active id, and the next id to be assigned.

To read a row, InnoDB compares the row's `DB_TRX_ID` with the view:

1. created by this transaction itself → visible;
2. created by a transaction that had **committed** before the view was made → visible;
3. created by a transaction still active at that time, or started after → **not visible**: follow `DB_ROLL_PTR` to the previous version and test again.

If the walk reaches a version that did not exist yet (the row was inserted after the snapshot), the row is simply absent from this reader's result. A delete-marked version that is visible means the row is deleted for this reader.

Isolation levels are snapshot timing

MVCC is how InnoDB implements the two levels you meet most:

Level	When the read view is created
`READ COMMITTED`	at the start of **every statement** — each sees the latest committed data
`REPEATABLE READ`	at the **first consistent read** of the transaction, then reused until commit

That is the whole difference between them for plain reads. `START TRANSACTION WITH CONSISTENT SNAPSHOT` creates the view immediately rather than at the first read — `mysqldump`

`--single-transaction` uses it to take a consistent backup of a live database without locking it.

Remember the boundary from the isolation lesson: MVCC covers **plain** reads. `UPDATE`, `DELETE` and `SELECT … FOR UPDATE` read the **latest committed** version and lock it, because changing an old version would lose someone's committed write.

Purge: cleaning up old versions

Old versions cannot be kept forever. InnoDB's background **purge** threads discard undo records — and physically remove delete-marked rows — once **no read view could still need them**: when the version has been superseded by a newer one committed before the **oldest active read view** was created.

That rule is why one idle transaction can hurt a whole server:

```sql
-- session A, at 09:00
START TRANSACTION;
SELECT COUNT(*) FROM orders;      -- read view created
-- …the developer goes to lunch, the session stays open
```

Until session A ends, purge cannot remove any version newer than its view. Every update made since 09:00 leaves undo records behind; the **history list length** (shown in `SHOW ENGINE INNODB STATUS`) climbs; queries that need an old snapshot walk ever-longer chains; and the undo tablespaces grow. Monitoring old entries in `information_schema.INNODB_TRX` and killing forgotten sessions is a routine DBA task.

PostgreSQL does it differently

PostgreSQL also uses MVCC, with a different layout:

- an `UPDATE` writes a **new tuple** in the table and marks the old one dead, using hidden `xmin` / `xmax` transaction ids on each tuple;
- old tuples stay in the table pages until **VACUUM** (usually autovacuum) reclaims them — the counterpart of InnoDB's purge;
- a table with heavy updates and lagging vacuum suffers **bloat**, the counterpart of a long undo history;

- `REPEATABLE READ` there is snapshot isolation: a concurrent update to a row you also update raises a serialization error rather than waiting and applying on the newer version as InnoDB does.

Oracle keeps old versions in undo segments, closer to InnoDB; a query that needs a version already overwritten fails with the famous "snapshot too old" error.

What MVCC does not solve

- **Write–write conflicts** still need locks: two transactions updating the same row are serialised.
- **Lost updates** in read-modify-write code: both read version 1 and both write — use an atomic update, `FOR UPDATE` or a version column.
- **Write skew**: two snapshots, two different rows written, an invariant broken.

MVCC makes reads cheap and non-blocking; correctness for writes still comes from locks, constraints and careful transactions.

Practise

The exercises model InnoDB's bookkeeping in tables. The first computes what each snapshot sees — the newest version of each row created by a transaction committed before the snapshot, with deleted rows left out. The second plays purge: given the oldest active snapshot, it lists the old versions no reader can still need.

EXERCISE 1

What each snapshot sees

Model InnoDB's version chains. `row_versions` holds every version of every row (`version` increases per row, `trx_id` created it, `deleted` = 1 marks a delete). `commits` lists committed transactions in commit order (`commit_seq`); a transaction absent from it has not committed. A snapshot taken at `as_of_seq` sees a version only if its transaction committed with `commit_seq ⩽ as_of_seq`, and for each row it reads the **newest** such version — and the row is absent if that version is a delete.

`row_versions`:

row_id	version	trx_id	val	deleted
1	1	100	Pune	0
1	2	103	Mumbai	0
2	1	100	red	0
2	2	105	blue	0
3	1	102	draft	0
3	2	104	draft	1
4	1	106	new	0

`commits`: `(100, 1)`, `(102, 2)`, `(103, 3)`, `(104, 4)`, `(106, 5)` — 105 is still running. `snapshots` (`reader`, `as_of_seq`): `('r1', 2)`, `('r2', 4)`, `('r3', 5)`.

Output columns: `reader`, `row_id`, `val`, ordered by `reader`, then `row_id`.

```
reader | row_id | val
r1 | 1 | Pune
r1 | 2 | red
r1 | 3 | draft
r2 | 1 | Mumbai
r2 | 2 | red
r3 | 1 | Mumbai
r3 | 2 | red
r3 | 4 | new
```

INPUT 1

```
CREATE TABLE row_versions (
  row_id INT NOT NULL,
  version INT NOT NULL,
  trx_id INT NOT NULL,
  val VARCHAR(20),
  deleted INT NOT NULL
);
INSERT INTO row_versions VALUES
  (1, 1, 100, 'Pune', 0),
  (1, 2, 103, 'Mumbai', 0),
  (2, 1, 100, 'red', 0),
  (2, 2, 105, 'blue', 0),
  (3, 1, 102, 'draft', 0),
  (3, 2, 104, 'draft', 1),
  (4, 1, 106, 'new', 0);
CREATE TABLE commits (trx_id INT PRIMARY
KEY, commit_seq INT NOT NULL);
INSERT INTO commits VALUES (100, 1),
(102, 2), (103, 3), (104, 4), (106, 5);
CREATE TABLE snapshots (reader
VARCHAR(5) NOT NULL, as_of_seq INT NOT
NULL);
INSERT INTO snapshots VALUES ('r1', 2),
('r2', 4), ('r3', 5);
```

OUTPUT 1

```
reader | row_id | val
r1 | 1 | Pune
r1 | 2 | red
r1 | 3 | draft
r2 | 1 | Mumbai
r2 | 2 | red
r3 | 1 | Mumbai
r3 | 2 | red
r3 | 4 | new
```

REFERENCE SOLUTION

```
WITH visible AS (
  SELECT s.reader,
         v.row_id,
         v.val,
         v.deleted,
         ROW_NUMBER() OVER (PARTITION BY s.reader, v.row_id ORDER BY v.version DESC)
  FROM snapshots s
  CROSS JOIN row_versions v
  JOIN commits c ON c.trx_id = v.trx_id
  WHERE c.commit_seq ⩽ s.as_of_seq
)
SELECT reader, row_id, val
FROM visible
WHERE rn = 1
  AND deleted = 0
ORDER BY reader, row_id;
```

EXERCISE 2

What purge may remove

InnoDB's purge may discard an old version once **no active read view can need it**: when a **newer** version of the same row was committed at or before the **oldest** active view's `as_of_seq`. If no view is active, the oldest view counts as the newest commit (`MAX(commit_seq)`). List the purgeable versions.

`row_versions` (`row_id`, `version`, `trx_id`, `val`, `deleted`):

row_id	version	trx_id	val	deleted
1	1	100	Pune	0
1	2	103	Mumbai	0
1	3	107	Delhi	0
2	1	100	red	0
2	2	105	blue	0
3	1	102	draft	0
3	2	104	draft	1

`commits`: `(100, 1)`, `(102, 2)`, `(103, 3)`, `(104, 4)`, `(107, 6)` — 105 is still running.
`active_views` (`reader`, `as_of_seq`): `('r7', 5)`, `('r2', 3)`, `('r8', 6)` — so the oldest view is at 3.

Output columns: `row_id`, `version`, ordered by `row_id`, then `version`.

```
row_id | version
1 | 1
```

Row 1's version 1 was superseded by version 2, committed at 3 — every view sees version 2 or later. Row 3's version 1 must stay: its delete committed at 4, after the view `r2` was taken.

INPUT 1

```
CREATE TABLE row_versions (
  row_id INT NOT NULL,
  version INT NOT NULL,
  trx_id INT NOT NULL,
  val VARCHAR(20),
  deleted INT NOT NULL
);
INSERT INTO row_versions VALUES
  (1, 1, 100, 'Pune', 0),
  (1, 2, 103, 'Mumbai', 0),
  (1, 3, 107, 'Delhi', 0),
  (2, 1, 100, 'red', 0),
  (2, 2, 105, 'blue', 0),
  (3, 1, 102, 'draft', 0),
  (3, 2, 104, 'draft', 1);
CREATE TABLE commits (trx_id INT PRIMARY
KEY, commit_seq INT NOT NULL);
INSERT INTO commits VALUES (100, 1),
(102, 2), (103, 3), (104, 4), (107, 6);
CREATE TABLE active_views (reader
VARCHAR(5) NOT NULL, as_of_seq INT NOT
NULL);
INSERT INTO active_views VALUES ('r7',
5), ('r2', 3), ('r8', 6);
```

OUTPUT 1

```
row_id | version
1 | 1
```

REFERENCE SOLUTION

```
SELECT v.row_id, v.version
FROM row_versions v
WHERE EXISTS (
  SELECT 1
  FROM row_versions n
  JOIN commits c ON c.trx_id = n.trx_id
  WHERE n.row_id = v.row_id
    AND n.version > v.version
    AND c.commit_seq ⩽ COALESCE((SELECT MIN(as_of_seq) FROM active_views),
                                (SELECT MAX(commit_seq) FROM commits))
)
ORDER BY v.row_id, v.version;
```

QUIZ

1. What is the main benefit of MVCC?

A. Writes never conflict

B. Readers see a consistent snapshot without blocking writers, and writers do not block readers

C. No locks are ever taken

D. Rows are stored once only

2. Where does InnoDB keep the previous versions of an updated row?

A. As new rows in the same table

B. In the undo log, linked from the row by its roll pointer

C. In the redo log

D. In the binary log

3. A session opened a transaction at 09:00, ran one SELECT and has been idle since. What grows while it stays open?

A. Nothing — idle sessions cost nothing

B. The undo history: purge cannot remove versions its read view may still need

C. The redo log only

D. The binary log

4. Under REPEATABLE READ, T1 reads a row with a plain SELECT, T2 updates it and commits, then T1 runs `UPDATE … WHERE id = that row`. Which version does T1's UPDATE act on?

A. T1's snapshot version

B. The latest committed version, written by T2

C. Neither: it fails with a serialization error

D. A random one

ANSWER KEY · MVCC — HOW READERS AND WRITERS AVOID EACH OTHER

1. B Plain reads use old versions instead of locks. Two writers to the same row still conflict and are ordered by locks.

2. B InnoDB updates the row in place and stores the old values in undo records reached through `DB_ROLL_PTR`. PostgreSQL is the one that writes a new tuple into the table.

3. B Purge removes only versions older than every active read view. One forgotten transaction keeps every later version's undo alive, raising the history list length and slowing reads.

4. B Writes and locking reads in InnoDB use the latest committed version; only plain SELECTs read the snapshot. PostgreSQL's REPEATABLE READ would instead raise a serialization error here.

LESSON 5 · 13 MIN

Savepoints — partial rollback inside a transaction

A transaction is all or nothing — until it is not convenient to be. A batch import where one record out of hundreds is bad, a checkout where the optional gift-wrap step fails, a framework that lets one service method call another inside the same transaction: in each case you want to undo **part** of the work and keep going. **Savepoints** give a transaction internal checkpoints. This lesson covers the three statements, how savepoints stack, how they interact with locks and errors in MySQL and PostgreSQL, and where they are used for real.

The three statements

```
START TRANSACTION;

INSERT INTO order_lines (order_id, sku, qty) VALUES (501, 'PEN-BLK', 10);
SAVEPOINT after_pens;

INSERT INTO order_lines (order_id, sku, qty) VALUES (501, 'INK-BLU', 500);
ROLLBACK TO SAVEPOINT after_pens;      -- undo the ink line only

INSERT INTO order_lines (order_id, sku, qty) VALUES (501, 'INK-BLU', 5);
COMMIT;                                -- pens and 5 ink cartridges are saved
```

- `SAVEPOINT name` marks the current point. Reusing a name moves the mark: the old savepoint of that name is replaced.
- `ROLLBACK TO [SAVEPOINT] name` undoes every change made after the mark. The transaction stays open, and the savepoint stays too, so you can roll back to it again.
- `RELEASE SAVEPOINT name` removes the mark and keeps the changes. It is bookkeeping, not a commit: nothing is durable until the transaction's `COMMIT`.

`COMMIT` and a full `ROLLBACK` end the transaction and delete every savepoint.

Savepoints stack

Savepoints form a stack in the order they were created:

```sql
START TRANSACTION;
UPDATE stock SET qty = qty - 1 WHERE sku = 'A';    -- change 1
SAVEPOINT s1;
UPDATE stock SET qty = qty - 1 WHERE sku = 'B';    -- change 2
SAVEPOINT s2;
UPDATE stock SET qty = qty - 1 WHERE sku = 'C';    -- change 3

ROLLBACK TO SAVEPOINT s1;   -- undoes changes 2 and 3; s2 no longer exists
-- ROLLBACK TO SAVEPOINT s2;   would now fail: SAVEPOINT s2 does not exist
COMMIT;                     -- keeps change 1 only
```

Rolling back to an earlier savepoint destroys every later one. `RELEASE SAVEPOINT s1` would likewise remove `s1` and `s2` — but keep all three changes.

A rollback to a savepoint is not a full rollback

Three behaviours to know for MySQL's InnoDB:

- **Locks are kept.** Row locks taken after the savepoint stay until the transaction ends, even though the changes are undone (locks on rows *inserted* after the savepoint disappear with the undo). Another session waiting on those rows keeps waiting.
- **The transaction is still yours to finish.** Forgetting the final `COMMIT` loses everything, including the work before the savepoint.
- **Implicit commits erase savepoints.** A DDL statement such as `CREATE TABLE` commits the transaction and with it every savepoint.

Savepoints and errors

How an error inside a transaction behaves is where MySQL and PostgreSQL differ most, and where savepoints matter most.

MySQL. A failing statement — a duplicate key, a `CHECK` violation — is rolled back on its own; the transaction stays open with its earlier changes, and the application decides what to do next. A savepoint lets you undo a whole *group* of statements that turned out wrong, not just the last one.

PostgreSQL. Any error **aborts** the transaction: every further statement fails with `current transaction is aborted, commands ignored until end of transaction block`. The only

ways out are `ROLLBACK`, or `ROLLBACK TO SAVEPOINT` if one was set before the error. That is why PostgreSQL drivers offer an "autosave" mode that silently sets a savepoint before each statement, and why bulk loaders there set one per record:

```
-- PostgreSQL, per-record recovery
BEGIN;
SAVEPOINT rec;
INSERT INTO customers VALUES (…);      -- fails on a duplicate
ROLLBACK TO SAVEPOINT rec;             -- the transaction is usable again
SAVEPOINT rec;
INSERT INTO customers VALUES (…);      -- the next record
…
COMMIT;
```

Where savepoints are used

- **Nested transactions in frameworks.** Databases do not nest transactions. When a service method marked transactional calls another inside an existing transaction with "nested" propagation — Spring's `Propagation.NESTED`, Django's nested `atomic()` blocks, SQLAlchemy's `begin_nested()` — the framework issues `SAVEPOINT`, then `RELEASE SAVEPOINT` on success or `ROLLBACK TO SAVEPOINT` on an exception. The inner failure is contained; the outer work survives.
- **Batch jobs.** Process records in a transaction, setting a savepoint before each chunk; a bad chunk is rolled back and logged without losing the good ones.
- **Optional steps.** Apply a coupon or reserve a gift wrap after a savepoint; if the step is refused, roll back to it and complete the order without it.
- **Testing.** Test suites wrap each test in a transaction, and savepoints let a test that itself uses transactions run inside it.

Cost and limits

Savepoints are cheap — a marker in the undo log — but not free: a transaction with thousands of open savepoints carries their bookkeeping until it ends, so release the ones you are done with in long loops. And a savepoint never shortens a transaction: everything from `START TRANSACTION` onward still holds its locks and its read view until the end.

Practise

Both exercises are scripts with a mistake in the middle that a savepoint undoes. The first fixes one wrong order line without losing the good ones; the second applies stock adjustments behind savepoints, rolls back to an earlier one — discarding the later savepoint with it — and records the adjustments that survived.

EXERCISE 1

Fix one line, keep the rest

Enter order **501**'s lines in one transaction, using a savepoint to recover from a mistake:

1. `START TRANSACTION`; insert `(501, 'PEN-BLK', 10)` and `(501, 'NB-A5', 4)`;
2. `SAVEPOINT after_basics`;
3. insert `(501, 'INK-BLU', 500)` — a typo;
4. `ROLLBACK TO SAVEPOINT after_basics`;
5. insert `(501, 'INK-BLU', 5)` and `COMMIT`.

Then show every line (the SELECT is in the starter).

`order_lines` (`order_id`, `sku`, `qty`) already holds `(500, 'PEN-RED', 2)`.

Output columns: `order_id`, `sku`, `qty`, ordered by `order_id`, then `sku`.

```
order_id | sku | qty
500 | PEN-RED | 2
501 | INK-BLU | 5
501 | NB-A5 | 4
501 | PEN-BLK | 10
```

INPUT 1

```
CREATE TABLE order_lines (
  order_id INT NOT NULL,
  sku VARCHAR(12) NOT NULL,
  qty INT NOT NULL
);
INSERT INTO order_lines VALUES (500,
'PEN-RED', 2);
```

OUTPUT 1

```
order_id | sku | qty
500 | PEN-RED | 2
501 | INK-BLU | 5
501 | NB-A5 | 4
501 | PEN-BLK | 10
```

REFERENCE SOLUTION

```sql
START TRANSACTION;

INSERT INTO order_lines (order_id, sku, qty) VALUES (501, 'PEN-BLK', 10);
INSERT INTO order_lines (order_id, sku, qty) VALUES (501, 'NB-A5', 4);
SAVEPOINT after_basics;

INSERT INTO order_lines (order_id, sku, qty) VALUES (501, 'INK-BLU', 500);
ROLLBACK TO SAVEPOINT after_basics;

INSERT INTO order_lines (order_id, sku, qty) VALUES (501, 'INK-BLU', 5);
COMMIT;

SELECT order_id, sku, qty FROM order_lines ORDER BY order_id, sku;
```

EXERCISE 2

Rolling back past a later savepoint

Apply stock adjustments, each logged in `adjustments`, with two savepoints — then roll back to the **first**, which also discards the second:

1. `START TRANSACTION`; subtract 2 from `'A'` and log `('A', -2)`;
2. `SAVEPOINT s1`; subtract 5 from `'B'` and log `('B', -5)`;
3. `SAVEPOINT s2`; add 10 to `'C'` and log `('C', 10)`;
4. `ROLLBACK TO SAVEPOINT s1` — the B and C adjustments were a mistake;
5. subtract 1 from `'D'`, log `('D', -1)`, and `COMMIT`.

Then show the stock and the log (both SELECTs are in the starter).

`stock` (`sku`, `qty`): `('A', 10)`, `('B', 20)`, `('C', 30)`, `('D', 40)`. `adjustments` (`sku`, `delta`) starts empty.

Output: `sku`, `qty` ordered by `sku`; then `sku`, `delta` ordered by `sku`.

```
sku | qty
A | 8
B | 20
C | 30
D | 39

sku | delta
A | -2
D | -1
```

INPUT 1

```
CREATE TABLE stock (sku VARCHAR(8)
PRIMARY KEY, qty INT NOT NULL);
INSERT INTO stock VALUES ('A', 10),
('B', 20), ('C', 30), ('D', 40);
CREATE TABLE adjustments (sku VARCHAR(8)
NOT NULL, delta INT NOT NULL);
```

OUTPUT 1

```
sku | qty
A | 8
B | 20
C | 30
D | 39

sku | delta
A | -2
D | -1
```

REFERENCE SOLUTION

```sql
START TRANSACTION;

UPDATE stock SET qty = qty - 2 WHERE sku = 'A';
INSERT INTO adjustments (sku, delta) VALUES ('A', -2);
SAVEPOINT s1;

UPDATE stock SET qty = qty - 5 WHERE sku = 'B';
INSERT INTO adjustments (sku, delta) VALUES ('B', -5);
SAVEPOINT s2;

UPDATE stock SET qty = qty + 10 WHERE sku = 'C';
INSERT INTO adjustments (sku, delta) VALUES ('C', 10);

ROLLBACK TO SAVEPOINT s1;

UPDATE stock SET qty = qty - 1 WHERE sku = 'D';
INSERT INTO adjustments (sku, delta) VALUES ('D', -1);
COMMIT;

SELECT sku, qty FROM stock ORDER BY sku;
SELECT sku, delta FROM adjustments ORDER BY sku;
```

QUIZ

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

A. Ends the transaction and undoes everything

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

C. Commits everything before `s1`

D. Deletes savepoint `s1` and keeps all changes

2. Savepoints `s1` then `s2` are set. After `ROLLBACK TO SAVEPOINT s1`, what happens to `ROLLBACK TO SAVEPOINT s2`?

A. It works as before

B. It fails: `s2` was discarded with the rollback to the earlier `s1`

C. It rolls back to `s1` again

D. It commits

3. Does `RELEASE SAVEPOINT s1` make the changes before it durable?

A. Yes, it is a partial commit

B. No — it only removes the marker; nothing is durable until the transaction commits

C. Yes, but only for inserts

D. It rolls them back

4. Why do PostgreSQL bulk loaders often set a savepoint before each record?

A. Savepoints make inserts faster

B. In PostgreSQL any error aborts the transaction; rolling back to a savepoint is the only way to continue it

C. To release locks after each record

D. PostgreSQL requires a savepoint per statement

ANSWER KEY · SAVEPOINTS — PARTIAL ROLLBACK INSIDE A TRANSACTION

1. B It is a partial undo: the work before `s1` remains pending, the transaction continues, and `s1` still exists. The last option describes `RELEASE SAVEPOINT`.

2. B Savepoints stack. Rolling back to an earlier one removes every savepoint created after it, together with the changes made after it.

3. B Releasing is bookkeeping. If the transaction is later rolled back, everything — including the work before the released savepoint — is undone.

4. B After an error PostgreSQL refuses further statements until `ROLLBACK` or `ROLLBACK TO SAVEPOINT`. MySQL rolls back only the failing statement and keeps the transaction usable.

CHECKPOINT · 25 MIN

Checkpoint — Transactions and concurrency

This checkpoint covers the whole module: transactions and the four ACID properties with the InnoDB machinery behind each, autocommit and implicit commits, the isolation levels and the dirty, non-repeatable and phantom reads they allow, lost updates and write skew, shared and exclusive row locks, locking reads, gap and next-key locks, lock waits and deadlocks, optimistic locking, MVCC with undo logs, read views and purge, and savepoints.

How it works. Twelve questions and three SQL tasks. You need 70% on the questions and every task accepted to clear the module. You can retake it as often as you like; your best score counts.

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

- What does each letter of ACID guarantee, and which InnoDB log provides atomicity and which durability?
- Why does `CREATE TABLE` inside a MySQL transaction make a later `ROLLBACK` useless?
- Draw the isolation-level table: four levels, three anomalies.
- What is InnoDB's default isolation level, and PostgreSQL's?
- What do `SELECT … FOR UPDATE`, `FOR SHARE` and `SKIP LOCKED` do?
- What is a next-key lock, and which anomaly does it prevent?
- How does InnoDB break a deadlock, and what should the application do about it?
- When does a `REPEATABLE READ` transaction create its read view, and why does a forgotten open transaction grow the undo history?
- What happens to later savepoints when you roll back to an earlier one?

The three tasks: a transfer that also charges a fee, where the fee is undone with a savepoint before the commit; an optimistic-locking update in which a stale second writer changes nothing; and a lock-wait report that ranks the blocking transactions by how many sessions wait on them and for how long.

EXERCISE 1

A transfer with a waived fee

In one transaction, move **300** from account **1** to account **2**, logging both movements in `ledger` with reason `'transfer'`. Then `SAVEPOINT before_fee`, charge account 1 a fee of **10** (balance and a `'fee'` ledger row) — and, because the fee is waived for this customer, `ROLLBACK TO SAVEPOINT before_fee` and `COMMIT`. Show the balances and the ledger (both SELECTs are in the starter).

`accounts`: `(1, 'Asha', 1000)`, `(2, 'Ravi', 200)`. `ledger` (`account_id`, `delta`, `reason`) starts empty.

Output: `id`, `owner`, `balance` ordered by `id`; then `account_id`, `delta`, `reason` ordered by `account_id`, then `delta`.

```
id | owner | balance
1 | Asha | 700
2 | Ravi | 500

account_id | delta | reason
1 | -300 | transfer
2 | 300 | transfer
```

INPUT 1

```
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  owner VARCHAR(40) NOT NULL,
  balance INT NOT NULL
);
INSERT INTO accounts VALUES (1, 'Asha',
1000), (2, 'Ravi', 200);
CREATE TABLE ledger (
  account_id INT NOT NULL,
  delta INT NOT NULL,
  reason VARCHAR(20) NOT NULL
);
```

OUTPUT 1

```
id | owner | balance
1 | Asha | 700
2 | Ravi | 500

account_id | delta | reason
1 | -300 | transfer
2 | 300 | transfer
```

REFERENCE SOLUTION

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 300 WHERE id = 1;
UPDATE accounts SET balance = balance + 300 WHERE id = 2;
INSERT INTO ledger (account_id, delta, reason) VALUES (1, -300, 'transfer'), (2, 300

SAVEPOINT before_fee;
UPDATE accounts SET balance = balance - 10 WHERE id = 1;
INSERT INTO ledger (account_id, delta, reason) VALUES (1, -10, 'fee');
ROLLBACK TO SAVEPOINT before_fee;

COMMIT;

SELECT id, owner, balance FROM accounts ORDER BY id;
SELECT account_id, delta, reason FROM ledger ORDER BY account_id, delta;
```

EXERCISE 2

Optimistic locking with a version column

Two editors opened product **7** when its `version` was **3**. Editor A saves first: set `price` to **549** and increment `version`, **only if** the version is still 3. Editor B saves second with the same stale version: set `price` to **499** and increment `version`, only if the version is still 3. Write both conditional updates in that order, then show the products (the SELECT is in the starter).

`products` (`id`, `name`, `price`, `version`): `(7, 'Desk lamp', 599, 3)`, `(8, 'Chair', 2499, 1)`.

Output columns: `id`, `price`, `version`, ordered by `id`.

```
id | price | version
7 | 549 | 4
8 | 2499 | 1
```

Editor B's update changes no row — in an application, "0 rows affected" is the signal to reload and retry.

INPUT 1

```
CREATE TABLE products (
  id INT PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  price INT NOT NULL,
  version INT NOT NULL
);
INSERT INTO products VALUES (7, 'Desk
lamp', 599, 3), (8, 'Chair', 2499, 1);
```

OUTPUT 1

```
id | price | version
7 | 549 | 4
8 | 2499 | 1
```

REFERENCE SOLUTION

```sql
UPDATE products
SET price = 549, version = version + 1
WHERE id = 7 AND version = 3;

UPDATE products
SET price = 499, version = version + 1
WHERE id = 7 AND version = 3;

SELECT id, price, version FROM products ORDER BY id;
```

EXERCISE 3

Who is blocking everyone?

`lock_waits` is a snapshot of current waits (`waiting_txn` waits for `blocking_txn`, for `wait_secs` so far). For each blocking transaction, report how many transactions wait on it, the longest wait, and whether it is a **root** blocker — not itself waiting for anyone (`'yes'` / `'no'`). Root blockers are the ones a DBA looks at first.

`lock_waits`:

waiting_txn	blocking_txn	wait_secs
12	10	41
13	10	8
14	12	30
15	11	5
16	12	2

Output columns: `blocking_txn`, `waiters`, `longest_wait`, `is_root`, ordered by `waiters` descending, then `longest_wait` descending, then `blocking_txn`.

```
blocking_txn | waiters | longest_wait | is_root
10 | 2 | 41 | yes
12 | 2 | 30 | no
11 | 1 | 5 | yes
```

INPUT 1

```
CREATE TABLE lock_waits (
  waiting_txn INT NOT NULL,
  blocking_txn INT NOT NULL,
  wait_secs INT NOT NULL
);
INSERT INTO lock_waits VALUES (12, 10,
41), (13, 10, 8), (14, 12, 30), (15, 11,
5), (16, 12, 2);
```

OUTPUT 1

```
blocking_txn | waiters | longest_wait |
is_root
10 | 2 | 41 | yes
12 | 2 | 30 | no
11 | 1 | 5 | yes
```

REFERENCE SOLUTION

```
SELECT b.blocking_txn,
       COUNT(*) AS waiters,
       MAX(b.wait_secs) AS longest_wait,
       CASE WHEN EXISTS (SELECT 1 FROM lock_waits w WHERE w.waiting_txn = b.blocking
            THEN 'no' ELSE 'yes' END AS is_root
FROM lock_waits b
GROUP BY b.blocking_txn
ORDER BY waiters DESC, longest_wait DESC, b.blocking_txn;
```

CHECKPOINT QUESTIONS

1. Which pairing of ACID property and InnoDB mechanism is correct?

A. Atomicity — redo log

B. Durability — redo log flushed at commit; atomicity — undo log

C. Isolation — binary log

D. Consistency — doublewrite buffer

2. Which anomalies does READ COMMITTED still allow?

A. Dirty reads only

B. Non-repeatable reads and phantoms

C. Phantoms only

D. None

3. In InnoDB, what does a plain `SELECT` (no `FOR UPDATE`) lock under REPEATABLE READ?

A. Every row it reads, shared

B. Nothing — it reads a consistent snapshot

C. The whole table

D. The gaps only

4. Why can an `UPDATE … WHERE customer_email = ?` lock far more rows than it changes?

A. UPDATE always locks the table

B. Without an index on `customer_email` it scans, and InnoDB locks every index record it scans

C. Because of autocommit

D. MVCC copies every row

5. Error 1205 `Lock wait timeout exceeded` means…

A. A deadlock was detected

B. A statement waited longer than `innodb_lock_wait_timeout` for a lock

C. The connection timed out

D. The redo log is full

6. When does optimistic locking fit better than `SELECT … FOR UPDATE`?

A. When the same hot row is updated constantly

B. When conflicts are rare and the user may take minutes between reading and saving

C. When the table has no primary key

D. Never — it is unsafe

7. A deadlock victim was rolled back with error 1213. What should the application do?

A. Report data corruption

B. Retry the whole transaction

C. Retry only the last statement

D. Raise the isolation level

8. Under READ COMMITTED, when does InnoDB create a read view?

A. Once per transaction

B. At the start of each consistent-read statement

C. Only on COMMIT

D. Never

9. In PostgreSQL, two REPEATABLE READ transactions update the same row; the first commits. What happens to the second?

A. It overwrites the first's change

B. It fails with a serialization error and must be retried

C. It waits forever

D. It silently does nothing

10. What does `RELEASE SAVEPOINT s1` do?

A. Rolls back to `s1`

B. Removes `s1` (and later savepoints) while keeping all the changes

C. Commits the transaction

D. Releases the row locks taken after `s1`

11. Which statement about savepoints and locks in InnoDB is true?

A. ROLLBACK TO SAVEPOINT releases every lock taken after the savepoint

B. Row locks taken after the savepoint are generally kept until the transaction ends

C. Savepoints prevent deadlocks

D. A savepoint locks the whole table

12. Which tool shows which transaction is blocking which in MySQL 8?

A. `SHOW TABLES`

B. `sys.innodb_lock_waits` (over `performance_schema.data_lock_waits`)

C. `EXPLAIN ANALYZE`

D. `information_schema.COLUMNS`

ANSWER KEY · CHECKPOINT — TRANSACTIONS AND CONCURRENCY

1. B The redo log makes committed work survive crashes; the undo log lets uncommitted work be rolled back and also feeds MVCC snapshots. Consistency comes from constraints and correct transactions.

2. B Each statement sees the latest committed data, so a repeated read can see a changed row or new rows. Only uncommitted data is hidden.

3. B Consistent reads use MVCC and take no row locks. Only under SERIALIZABLE with autocommit off do plain SELECTs become `FOR SHARE`.

4. B Locks are placed on the records a statement examines. An index on the filtered column limits the scan — and the locks — to the matching rows.

5. B 1205 is a plain wait that ran out of time (50 s by default), usually behind a transaction someone left open. A deadlock is error 1213 and is detected immediately.

6. B Holding a lock while a person edits a form blocks everyone else; a version check costs nothing until a conflict actually happens. For hot rows with frequent conflicts, retries pile up and pessimistic locking wins.

7. B The entire transaction was rolled back cleanly, so re-running it from the start is safe; retrying only the failed statement would miss the earlier, undone work.

8. B A fresh view per statement is why READ COMMITTED sees other transactions' commits mid-transaction. REPEATABLE READ creates one per transaction.

9. B PostgreSQL's REPEATABLE READ is snapshot isolation and refuses to update a row changed since the snapshot. InnoDB instead applies the update to the latest committed version.

10. B Release is bookkeeping only; the changes stay pending until the transaction commits or rolls back, and locks are held until then.

11. B InnoDB undoes the changes but keeps the locks it stored in memory after the savepoint; only locks on rows inserted after it go away with the undo.

12. B `performance_schema.data_locks` and `data_lock_waits` expose the locks; the `sys.innodb_lock_waits` view pairs each waiting transaction with its blocker and their queries. `SHOW ENGINE INNODB STATUS` shows the latest deadlock.

CodeKairo · SQL study plan · Module 14: Transactions and concurrency. Solve the exercises online to track your progress and earn XP.