Checkpoint: Concurrency Control + Distributed DB (Banking Scenario)

This document contains: (1) clear explanations of the concurrency problems, (2) safe schedules, (3) SQL examples (naive vs pessimistic vs optimistic), (4) JavaScript/Node examples using SQL transactions, and (5) a high-level distributed database plan (fragmentation/replication/allocation).

Part 1 — Transaction Management

Scenario: Two transfers run concurrently and touch the same account.

1) What can go wrong: Lost Update

A **lost update** happens when two transactions read the same balance, compute new balances independently, then the later write overwrites the earlier write. Final balance becomes incorrect.

Step	T1: transfer 100 from A → B	T2: transfer 50 from A → C	Result
1	READ A = 1000		
2		READ A = 1000	both saw old value
3	WRITE A = 900		
4		WRITE A = 950	overwrites T1 ⇒ lost update

Correct final should be 850, but the unsafe schedule can end with 950. Therefore it is **not safe**.

2) Simple locking strategy

Shared lock (S) for reading (view balance) and **Exclusive lock (X)** for writing (transfer). For a transfer, take X-lock(s) on the affected account rows (source and destination) and hold them until COMMIT (strict two-phase locking behavior).

3) Pessimistic vs Optimistic locking

Pessimistic locking is recommended for transfers because correctness is critical and conflicts are plausible. It prevents problems by locking before updates. **Optimistic locking** is also correct, but it detects conflicts at write/commit time and requires retry logic.

4) Safe schedule with X-lock

Step	T1	T2	Safe?
1	X-LOCK(A)		
2	READ/UPDATE A		
3	COMMIT; UNLOCK(A)		
4		X-LOCK(A)	
5		READ/UPDATE A	
6		COMMIT; UNLOCK(A)	YES

SQL — Minimal Schema + Concurrency Examples

The SQL below is written in a PostgreSQL-friendly style. It focuses on demonstrating concurrency control rather than full production constraints.

Schema + fragmentation views

```
-- Core tables
CREATE TABLE customers (
  customer_id   UUID PRIMARY KEY,
  full_name     TEXT NOT NULL,
  phone         TEXT,
  address       TEXT,
  home_branch   TEXT NOT NULL CHECK (home_branch IN ('Tunis','Sousse','Sfax'))
);

-- Vertical fragmentation: auth separated
CREATE TABLE customer_auth (
  customer_id    UUID PRIMARY KEY REFERENCES customers(customer_id),
  email          TEXT UNIQUE NOT NULL,
  password_hash  TEXT NOT NULL,
  last_login_at  TIMESTAMP
);

CREATE TABLE accounts (
  account_id   UUID PRIMARY KEY,
  customer_id  UUID NOT NULL REFERENCES customers(customer_id),
  branch       TEXT NOT NULL CHECK (branch IN ('Tunis','Sousse','Sfax')),
  balance_cents BIGINT NOT NULL CHECK (balance_cents >= 0),
  version      BIGINT NOT NULL DEFAULT 1  -- for optimistic locking
);

CREATE TABLE ledger_transactions (
  tx_id           UUID PRIMARY KEY,
  created_at      TIMESTAMP NOT NULL DEFAULT NOW(),
  from_account_id UUID NOT NULL REFERENCES accounts(account_id),
  to_account_id   UUID NOT NULL REFERENCES accounts(account_id),
  amount_cents    BIGINT NOT NULL CHECK (amount_cents > 0),
  branch          TEXT NOT NULL CHECK (branch IN ('Tunis','Sousse','Sfax'))
);

-- Horizontal fragmentation (views per branch)
CREATE VIEW customers_tunis  AS SELECT * FROM customers WHERE home_branch='Tunis';
CREATE VIEW customers_sousse AS SELECT * FROM customers WHERE home_branch='Sousse';
CREATE VIEW customers_sfax   AS SELECT * FROM customers WHERE home_branch='Sfax';
```

Example 1 — Naive transfer (unsafe)

```
-- NAIVE TRANSFER (unsafe under concurrency)
-- Problem: lost update can happen if two transfers touch same account.
BEGIN;

-- read balance into app memory (or client)
SELECT balance_cents FROM accounts WHERE account_id = :from;

-- app computes: newBalance = oldBalance - amount
UPDATE accounts
SET balance_cents = :new_balance
WHERE account_id = :from;

UPDATE accounts
SET balance_cents = balance_cents + :amount
WHERE account_id = :to;

INSERT INTO ledger_transactions(tx_id, from_account_id, to_account_id, amount_cents, branch)
VALUES (:tx_id, :from, :to, :amount, :branch);
```

```
COMMIT;
```

Unsafe because two transactions can read the same old balance and overwrite each other.

Example 2 — Pessimistic locking (SELECT ... FOR UPDATE)

```
-- PESSIMISTIC LOCKING (safe): lock rows before updating
BEGIN;

-- Lock rows in consistent order to reduce deadlocks
-- (example: lock lower UUID first in application logic)
SELECT account_id, balance_cents
FROM accounts
WHERE account_id IN (:from, :to)
FOR UPDATE;

-- Withdraw
UPDATE accounts
SET balance_cents = balance_cents - :amount
WHERE account_id = :from AND balance_cents >= :amount;

-- Deposit
UPDATE accounts
SET balance_cents = balance_cents + :amount
WHERE account_id = :to;

INSERT INTO ledger_transactions(tx_id, from_account_id, to_account_id, amount_cents, branch)
VALUES (:tx_id, :from, :to, :amount, :branch);

COMMIT;
```

If another transfer touches the same row, it waits until COMMIT, preventing lost updates.

Example 3 — Optimistic locking (version column)

```
-- OPTIMISTIC LOCKING (safe): detect conflicts using version
-- If UPDATE affects 0 rows, someone changed the row first ⇒ retry.
BEGIN;

-- read
SELECT balance_cents, version
FROM accounts
WHERE account_id = :from;

-- attempt update only if version unchanged
UPDATE accounts
SET balance_cents = balance_cents - :amount,
    version = version + 1
WHERE account_id = :from
  AND version = :expected_version
  AND balance_cents >= :amount;

-- check rowcount in application:
-- if 0 => ROLLBACK and retry

UPDATE accounts
SET balance_cents = balance_cents + :amount,
    version = version + 1
WHERE account_id = :to
  AND version = :expected_version_to;

INSERT INTO ledger_transactions(tx_id, from_account_id, to_account_id, amount_cents, branch)
VALUES (:tx_id, :from, :to, :amount, :branch);

COMMIT;
```

Correct but needs retry logic; commonly used when conflicts are rare.

JavaScript (Node.js) — Minimal API Examples

Below are simplified Express + node-postgres examples to demonstrate the concepts.

Express API (pessimistic + optimistic examples)

```
// npm i express pg
import express from "express";
import { Pool } from "pg";
import crypto from "crypto";

const app = express();
app.use(express.json());

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
});

// Helper: run a function inside a DB transaction
async function withTx(fn) {
  const client = await pool.connect();
  try {
    await client.query("BEGIN");
    const res = await fn(client);
    await client.query("COMMIT");
    return res;
  } catch (err) {
    await client.query("ROLLBACK");
    throw err;
  } finally {
    client.release();
  }
}

/**
 * GET /balance/:accountId
 * Read-only: typically fine with normal isolation; for strictness you can use REPEATABLE READ.
 */
app.get("/balance/:accountId", async (req, res) => {
  const { accountId } = req.params;
  const r = await pool.query(
    "SELECT account_id, balance_cents FROM accounts WHERE account_id = $1",
    [accountId]
  );
  if (r.rowCount === 0) return res.status(404).json({ error: "ACCOUNT_NOT_FOUND" });
  res.json(r.rows[0]);
});

/**
 * POST /transfer (pessimistic locking)
 * body: { fromAccountId, toAccountId, amountCents, branch }
 */
app.post("/transfer", async (req, res) => {
  const { fromAccountId, toAccountId, amountCents, branch } = req.body;

  if (!fromAccountId || !toAccountId || !Number.isInteger(amountCents) || amountCents <= 0) {
    return res.status(400).json({ error: "INVALID_INPUT" });
  }

  try {
    const result = await withTx(async (client) => {
      // 1) Lock both account rows to prevent concurrent writes (lost update).
      // Lock order should be consistent to reduce deadlocks.
      const ids = [fromAccountId, toAccountId].sort(); // simplistic ordering
      await client.query(
        `SELECT account_id FROM accounts WHERE account_id = ANY($1::uuid[]) FOR UPDATE`,
        [ids]
      );
      // 2) Withdraw (with sufficient funds check)
      const w = await client.query(
        `UPDATE accounts
         SET balance_cents = balance_cents - $1
         WHERE account_id = $2 AND balance_cents >= $1
         RETURNING balance_cents`,
        [amountCents, fromAccountId]
      );
      if (w.rowCount === 0) {
        return { ok: false, error: "INSUFFICIENT_FUNDS" };
      }

      // 3) Deposit
      await client.query(
        `UPDATE accounts
         SET balance_cents = balance_cents + $1
         WHERE account_id = $2`,
        [amountCents, toAccountId]
      );

      // 4) Insert ledger record
      const txId = crypto.randomUUID();
      await client.query(
        `INSERT INTO ledger_transactions(tx_id, from_account_id, to_account_id, amount_cents, branch)
         VALUES ($1, $2, $3, $4, $5)`,
        [txId, fromAccountId, toAccountId, amountCents, branch ?? "Tunis"]
      );

      return { ok: true, txId };
    });

    if (!result.ok) return res.status(409).json(result);
    res.status(201).json(result);
  } catch (e) {
    console.error(e);
    res.status(500).json({ error: "INTERNAL_ERROR" });
  }
});

/**
 * POST /transfer-optimistic (optional): optimistic approach with retry.
 * This is just for learning; pessimistic is easier to justify for banking.
 */
app.post("/transfer-optimistic", async (req, res) => {
  const { fromAccountId, toAccountId, amountCents, branch } = req.body;

  const MAX_RETRIES = 3;
  for (let attempt = 1; attempt <= MAX_RETRIES; attempt++) {
    try {
      const out = await withTx(async (client) => {
        // read versions
        const fromRow = await client.query(
          "SELECT balance_cents, version FROM accounts WHERE account_id=$1",
          [fromAccountId]
        );
        const toRow = await client.query(
          "SELECT version FROM accounts WHERE account_id=$1",
          [toAccountId]
        );
        if (fromRow.rowCount === 0 || toRow.rowCount === 0) {
          return { ok: false, error: "ACCOUNT_NOT_FOUND" };
        }

        const { balance_cents: bal, version: vFrom } = fromRow.rows[0];
        const { version: vTo } = toRow.rows[0];
        if (bal < amountCents) return { ok: false, error: "INSUFFICIENT_FUNDS" };

        // conditional updates (detect conflict)
        const u1 = await client.query(
          `UPDATE accounts
           SET balance_cents = balance_cents - $1, version = version + 1
           WHERE account_id=$2 AND version=$3`,
          [amountCents, fromAccountId, vFrom]
        );
        const u2 = await client.query(
          `UPDATE accounts
           SET balance_cents = balance_cents + $1, version = version + 1
           WHERE account_id=$2 AND version=$3`,
          [amountCents, toAccountId, vTo]
        );

        if (u1.rowCount === 0 || u2.rowCount === 0) {
          // conflict => trigger retry
          throw new Error("OPTIMISTIC_CONFLICT");
        }

        const txId = crypto.randomUUID();
        await client.query(
          `INSERT INTO ledger_transactions(tx_id, from_account_id, to_account_id, amount_cents, branch)
           VALUES ($1, $2, $3, $4, $5)`,
          [txId, fromAccountId, toAccountId, amountCents, branch ?? "Tunis"]
        );

        return { ok: true, txId };
      });

      if (!out.ok) return res.status(409).json(out);
      return res.status(201).json({ ...out, attempt });
    } catch (e) {
      if (String(e?.message) === "OPTIMISTIC_CONFLICT" && attempt < MAX_RETRIES) {
        continue; // retry
      }
      return res.status(500).json({ error: "INTERNAL_ERROR", details: String(e?.message ?? e) });
    }
  }
});

app.listen(3000, () => console.log("API on http://localhost:3000"));
```

Part 2 — Distributed Database Planning

Branches: Tunis, Sousse, Sfax. The goal is fast local operations with correct global behavior.

Horizontal fragmentation (Customers by branch)

Store customer rows at the branch that owns them (home_branch). Each site holds only its fragment. Queries at a branch are mostly local, reducing network traffic.

Vertical fragmentation (separate auth column)

Keep sensitive login/auth columns in a separate table (CustomerAuth). This improves security and keeps typical branch queries lighter.

Replication across all branches (what + why)

Replicate **basic customer identifiers** (customer_id, name, status) so any branch can recognize a customer. Account balances may be replicated for availability, but balance updates must remain strongly consistent (e.g., primary-copy writes or synchronous replication). Transaction history is usually large, so full replication everywhere is costly.

Static vs dynamic allocation for transaction history

Dynamic allocation: store a transaction record at the branch where the transaction happens (local writes, good locality), then periodically consolidate to a central reporting/audit store. This reduces cross-branch write traffic and scales better.