Database Programming in SQL – Part II (continued)

DDL, Access Control (DCL) & Transactions (TxCL)   (90 min)

Michael Emmerich

February 10, 2026

Roadmap (today)

1) DDL: schema & integrity (30 min)

- Data types in practice
- Constraints: PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE, CHECK, DEFAULT
- ALTER TABLE and schema evolution
- Views and indexes (what/why)

2) DCL & TxCL (60 min)

- Access rights: roles, GRANT, REVOKE
- Transactions: BEGIN/COMMIT/ROLLBACK
- ACID, isolation anomalies, isolation levels
- Locking, 2PL, deadlocks (conceptual)

Connection to last lecture (2 min recap)

- Last time: querying data (SELECT/WHERE), joins, set operations, aggregation.
- Today: how we *define* the structure, protect data, and keep it correct under concurrency.

Mental model

$\Rightarrow$**Schema (DDL)** says what data *may exist*.

$\Rightarrow$**Rights (DCL)** say who *may do what*.

$\Rightarrow$**Transactions (TxCL)** say how concurrent changes stay *correct*.

Part A

Defining database structure (DDL) and Data Updates (DML)

SQL data types you will see often

SQL is strongly typed: each column has a declared type.

Type family	Examples	Typical use
Strings	`CHAR(n)`, `VARCHAR(n)`, `TEXT`	IDs, names, free text
Integers	`INT`, `INTEGER`	counts, years, quantities
Decimals	`NUMERIC(p,s)`, `DECIMAL(p,s)`	money, measurements
Booleans	`BOOLEAN`	flags, on/off states
Dates/times	`DATE`, `TIMESTAMP`	timestamps, events

Practical note: exact type behavior depends on DBMS; always check your product's documentation.

DDL recap: CREATE TABLE (structure + rules)

- Tables define columns, data types, and constraints.
- Constraints are the **first line of defense** for data quality.

```
CREATE TABLE CUSTOMER(
  customer_id  CHAR(4)     PRIMARY KEY,
  customer_name VARCHAR(15) NOT NULL,
  city         VARCHAR(10),
  customer_type CHAR(1),
  district     CHAR(1)
);

CREATE TABLE INVOICE(
  invoice_id    CHAR(4)    PRIMARY KEY,
  year          INT,
  invoice_total INT        CHECK (invoice_total >= 0),
  status        VARCHAR(2) DEFAULT 'OK',
  customer_id   CHAR(4)    NOT NULL,
  FOREIGN KEY(customer_id) REFERENCES CUSTOMER(customer_id)
);
```

Keys and referential integrity: PRIMARY KEY + FOREIGN KEY

Composite primary key (relationship table)

```
CREATE TABLE INVOICE_LINE(
  invoice_id  CHAR(4),
  product_id  CHAR(4),
  quantity    INT CHECK (quantity > 0),
  PRIMARY KEY (invoice_id, product_id),
  FOREIGN KEY (invoice_id) REFERENCES
        INVOICE(invoice_id) ON DELETE CASCADE,
  FOREIGN KEY (product_id) REFERENCES
        PRODUCT(product_id) ON DELETE SET NULL
);
```

Referential actions (idea)

- RESTRICT: deny update/delete in parent
- ON DELETE SET NULL: null Foreign Key in child
- ON DELETE CASCADE: propagate change/delete to child
- ON DELETE SET DEFAULT: assign default value

Exact availability and defaults depend on the DBMS.

Column constraints beyond keys

- DEFAULT sets a value if none is provided.
- NOT NULL forbids missing values.
- UNIQUE forbids duplicates.
- CHECK enforces domain/business rules.

```sql
CREATE TABLE CUSTOMER(
  customer_id  CHAR(4)     PRIMARY KEY,
  customer_name VARCHAR(15) NOT NULL,
  email        VARCHAR(50) UNIQUE,
  customer_type CHAR(1)    DEFAULT 'A',
  district     CHAR(1)     CHECK (district IN ('I','L','1','2','3'))
);
```

Rule of thumb: encode the *stable* rules in constraints; keep volatile rules in application logic.

Schema evolution: ALTER TABLE

- Databases evolve: new requirements, new columns, stricter rules.
- ALTER TABLE supports adding/dropping columns and constraints (syntax differs across DBMS).

```sql
-- Start minimal, then add constraints and columns
CREATE TABLE INVOICE_LINE(
  invoice_id CHAR(4),
  product_id CHAR(4)
);

ALTER TABLE INVOICE_LINE
  ADD COLUMN quantity INT;

-- In many DBMS (e.g., PostgreSQL) you can add constraints like this:
ALTER TABLE INVOICE_LINE
  ADD PRIMARY KEY (invoice_id, product_id);
```

Practical note: SQLite has limitations on ALTER TABLE compared to PostgreSQL/MySQL.

Views and indexes: two important DB objects

View = stored query (no data stored)

- Simplifies common queries
- Helps define **fine-grained rights** (bridge to DCL)

```sql
CREATE VIEW v_east_customers AS
SELECT customer_name, city, customer_type
FROM CUSTOMER
WHERE district = 'I';

SELECT * FROM v_east_customers;
```

Index = faster search structure

- Speeds up WHERE/JOIN/ORDER BY patterns
- Costs extra space and slows inserts/updates

```sql
CREATE INDEX customer_name_idx
ON CUSTOMER(customer_name);

DROP INDEX customer_name_idx;
```

Views: stored queries for reuse, rights, and derived data

View = stored query (typically no data stored)

- Simplifies common queries and hides complexity
- Helps define **fine-grained access rights** (expose only selected columns/rows)
- Can provide **derived attributes** and **unit conversions**

Example (SQLite): derived attributes + unit conversion

```sql
-- Assume table CITIZEN(citizen_id, name, city, size_cm, birthdate)

CREATE VIEW v_citizen_profile AS
SELECT
  citizen_id,
  city,
  ROUND(size_cm / 30.48, 2) AS size_ft, -- cm -> feet
  CAST((julianday('now') - julianday(birthdate)) / 365.25 AS INT) AS age_years
FROM CITIZEN;

SELECT * FROM v_citizen_profile;
```

Privacy idea: aggregates make individual data disappear (public health style)

Indexes: faster search structures (with trade-offs)

Index = auxiliary structure to speed up lookups

- Speeds up WHERE / JOIN / ORDER BY
- Costs extra space and can slow INSERT/UPDATE/DELETE

Common index types (concept)

- **Sorted list by attribute values**: fast search, but updates can be expensive
- **B+ tree**: keeps keys sorted, supports efficient updates, and supports **range queries**
- **Hash index**: use a hash function to map key $\rightarrow$ bucket/page; great for equality lookups, but not for ranges

Example: range query (benefits from B+ tree-like indexing)

```sql
CREATE INDEX citizen_birthdate_idx
ON CITIZEN(birthdate);

-- Range query: citizens born between 1973 and 1993
SELECT citizen_id, birthdate
FROM CITIZEN
WHERE birthdate BETWEEN '1973-01-01' AND '1993-12-31';
```

From DDL to DML: changing the data

After CREATE TABLE, we usually start manipulating rows using **DML**:

- INSERT = add new rows
- UPDATE = modify existing rows
- DELETE = remove rows

CRUD mapping (common in apps)

Create (rows) → INSERT Read → SELECT Update → UPDATE Delete → DELETE

INSERT: adding rows

- Always prefer the **explicit column list** (robust to schema changes).
- Columns not mentioned become NULL or their DEFAULT, if defined.

```
-- Insert one customer
INSERT INTO CUSTOMER (customer_id, customer_name, city, customer_type, district)
VALUES ('C001', 'Anna', 'Jyvaskyla', 'A', 'I');

-- Insert an invoice (status uses DEFAULT 'OK')
INSERT INTO INVOICE (invoice_id, year, invoice_total, customer_id)
VALUES ('I001', 2025, 120, 'C001');

-- Insert multiple rows (supported in most DBMS incl. PostgreSQL, MySQL, SQLite)
INSERT INTO CUSTOMER (customer_id, customer_name, city, customer_type, district)
VALUES
  ('C002', 'Ben', 'Turku',   'A', 'L'),
  ('C003', 'Cleo', 'Helsinki', 'B', '2');
```

Typical failure modes: duplicate PRIMARY KEY, FOREIGN KEY missing parent, CHECK violated.

UPDATE: changing existing rows

- UPDATE changes **all rows that match** the WHERE condition.
- **Without WHERE**: you update *every row* (common accident).

```sql
-- Mark an invoice as paid
UPDATE INVOICE
SET status = 'PA'
WHERE invoice_id = 'I001';

-- Correct a total (also demonstrates an expression)
UPDATE INVOICE
SET invoice_total = invoice_total + 10
WHERE year = 2025
  AND status = 'OK';

-- Update multiple columns at once
UPDATE CUSTOMER
SET city = 'Tampere',
    district = '1'
WHERE customer_id = 'C002';
```

Tip: do a SELECT ... WHERE ... first to verify which rows will be affected.

DELETE: removing rows

- DELETE removes **rows**, not table structure (that would be DROP TABLE).
- **Without WHERE**: you delete *every row*.
- Foreign keys may block deletion (e.g., RESTRICT) or propagate it (CASCADE).

```sql
-- Delete one invoice (only works if no referencing rows, or CASCADE is set)
DELETE FROM INVOICE
WHERE invoice_id = 'I001';

-- Delete all "test" customers (example condition)
DELETE FROM CUSTOMER
WHERE customer_name LIKE 'Test%';

-- Delete ALL rows (table remains!)
DELETE FROM INVOICE;
```

In practice: deletions are often done inside transactions, and sometimes replaced by "soft delete" (e.g., a column is_active).

Part B

Defining access rights (DCL)

Roles, users, and privileges (concept)

- SQL standard abstracts users/groups as **roles**.
- Every DB object has an **owner** (typically the creator).
- Owners can grant/revoke privileges to other roles.

Typical privileges

Privilege	**Allows**
`CONNECT`	connect to database
`SELECT`	read data
`INSERT`	insert rows
`UPDATE`	modify rows/columns
`DELETE`	delete rows
`EXECUTE`	execute routines
`ALL PRIVILEGES`	all above except `CONNECT`

GRANT: giving rights (with delegation option)

General form

```
GRANT privilege[, privilege]*
ON object
TO role[, role]*
[WITH GRANT OPTION];
```

Example: allow reading and updating, and allow re-granting

```
GRANT SELECT, UPDATE
ON INVOICE
TO analyst_role
WITH GRANT OPTION;
```

Column-level grants (fine-grained)

```
GRANT UPDATE (invoice_total, status)
ON INVOICE
TO billing_clerk;
```

Tip: combine with **views** to expose only selected columns/rows.

Roles can be granted to roles (hierarchies)

- Role hierarchies help manage many users: one grant updates many accounts.
- Creating users/roles is DBMS-specific (example shown in PostgreSQL style).

```
-- PostgreSQL-like example:
CREATE ROLE masa
  WITH PASSWORD 'jkl0088'
  IN ROLE varastotyontekijat
  LOGIN
  VALID UNTIL 'May 8 11:30:00 2016';

-- Grant role to role (inherit privileges)
GRANT analyst_role TO intern_role;
```

`varastotyontekijat`[1] = `warehouse_workers`.

[1] Finnish (standard spelling): *varastotyöntekijät* = "warehouse workers". Parts: *varasto* = warehouse; *työntekijä* = worker; *-t* = plural.

REVOKE: removing rights (and what happens downstream)

General form

```
REVOKE [GRANT OPTION FOR] privilege[,
    privilege]*
ON object
FROM role[, role]*;
```

Remove only delegation right

```
REVOKE GRANT OPTION FOR SELECT
ON INVOICE
FROM analyst_role;
```

Cascading effect (idea)

- If A granted to B *with grant option*
- and B granted further to C,
- then revoking from B typically also removes the derived right from C.

Best practice

- least privilege
- prefer views for fine-grained exposure

Part C

Transaction control (TxCL)

Transactions in SQL: the 3 core commands

- A transaction groups 1..n operations into one logical unit of work.
- SQL transaction control uses:

```sql
BEGIN [TRANSACTION];
-- ... your SELECT/INSERT/UPDATE/DELETE ...
COMMIT;  -- make changes permanent and visible
-- or
ROLLBACK; -- cancel changes of this transaction
```

Many systems also have *autocommit* mode: each statement becomes its own transaction unless you BEGIN explicitly.

ACID: what DBMSs try to guarantee

A – Atomicity
Either all operations succeed or none (failure triggers rollback).

C – Consistency
Database moves from one consistent state to another (constraints + rules hold).

I – Isolation
Concurrent transactions behave as if executed in some serial order.

D – Durability
After commit, results persist even after crashes (within practical limits).

Example (concept): guard a business rule with rollback

Business rule: bank account balance must not go below 0.

```
BEGIN TRANSACTION;
SELECT saldo
FROM tili
WHERE tilinro = :account_no;

-- if account not found: ROLLBACK;

UPDATE tili
SET saldo = saldo - :amount
WHERE tilinro = :account_no;

SELECT saldo
FROM tili
WHERE tilinro = :account_no;

-- if saldo < 0: ROLLBACK;

COMMIT;
```

tili[2] = account, tilinro[3] = account_no, saldo[4] = balance.

Isolation: what can go wrong with concurrency?

If transactions run in parallel, anomalies may violate the “as-if-serial” idea.

Classic anomalies

- **Lost update**: two writers overwrite each other (final state misses one update).
- **Dirty read**: a transaction reads uncommitted changes from another transaction.
- **Non-repeatable read**: re-reading a row yields a different value (someone committed an update).
- **Phantom read**: re-running a range query yields different *set of rows* (someone inserted/deleted in the range).

Lost update (simple example)

Two transactions both add 20 to the same balance, but one update is lost.

Time	T1	T2
t_1	Read balance $=$ 100	—
t_2	—	Read balance $=$ 100
t_3	Write balance $=$ 120	—
t_4	—	Write balance $=$ 120

Observation
Parallel result: 120. Serial result First T1 then T2: 140. Isolation aims to prevent this mismatch.

Isolation levels (SQL standard idea)

Looser isolation can be faster but allows more anomalies.

Isolation level	**Dirty reads**	**Non-repeatable**	**Phantoms**
`SERIALIZABLE`	no	no	no
`REPEATABLE READ`	no	no	yes
`READ COMMITTED`	no	yes	yes
`READ UNCOMMITTED`	yes	yes	yes

Exact behavior can differ by DBMS; the table captures the *standard* intent.

Concurrency control: how isolation is implemented (concept)

- A **schedule** is the interleaving of operations of concurrent transactions.
- A schedule is **serializable** if it is equivalent to some serial order.
- DBMSs use techniques such as:
  - **Locking** (common in classic RDBMS)
  - **Timestamp ordering** (also common in some settings)
  - **Optimistic vs pessimistic** strategies (high-level view)

Two-phase locking (2PL): the key protocol

2PL structures each transaction into two phases:

Expanding phase
Acquire all needed locks (and promotions). Do *not* release locks yet.

Shrinking phase
Release locks when they are no longer needed. Do *not* acquire new locks.

- 2PL + proper lock types aims for serializability.
- Stronger variants:
    - **Conservative 2PL**: acquire all locks before doing anything. $\Rightarrow$ Avoids deadlocks
    - **<u>Strict 2PL</u>**: keep all write locks until commit/rollback. $\Rightarrow$ Avoids cascading rollback.
    - **Strong strict 2PL**: keep all locks until commit/rollback.

Papadimitriou, C. H. (1979). The serializability of concurrent database updates. Journal of the ACM (JACM), 26(4), 631-653.

Strict 2PL example: lock upgrade causes waiting

Notation: `S(X)` = shared (read) lock on item X, `X(X)` = exclusive (write) lock, `R(X)`, `W(X)` = read/write. **Strict 2PL:** locks are released only at COMMIT/ROLLBACK.

Step	T1	T2	Lock state / comment
1	`S(A)`		`S(A)` held by T1
2	`R(A)`		read ok
3	`X(A)`		upgrade ok (no other holders)
4	`W(A)`		`X(A)` held by T1
5		`S(B)`	`S(B)` held by T2
6		`R(B)`	read ok
7	`S(B)`		compatible: T1 also gets `S(B)`
8	`R(B)`		read ok
9	`X(B)`		**WAIT**: cannot upgrade while T2 holds `S(B)`
10		COMMIT	strict 2PL: T2 releases `S(B)` only now
11	`X(B)`		upgrade granted
12	`W(B)`		write ok
13	COMMIT		strict 2PL: T1 releases `X(A)`, `X(B)`

Takeaway: even a *reader* (T2) can block a *writer* (T1) under strict 2PL when the writer needs an S$\rightarrow$X upgrade.

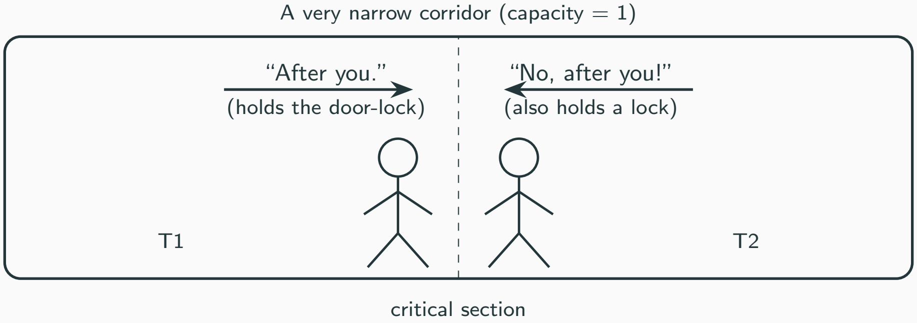
Deadlocks: circular waiting for locks

- Two-phase locking (2PL) helps isolation/serializability, but it can create **deadlocks**.
- A **deadlock** is a **cycle of waiting**: each transaction waits for a lock held by another.
- The situation does *not* resolve by itself: the DBMS must abort (rollback) one transaction.

Step	T1	T2
1	X-lock(A) granted	
2		X-lock(B) granted
3	request X-lock(B) *(blocked)*	
4		request X-lock(A) *(blocked)*
	Now: T1 waits for B (held by T2) and T2 waits for A (held by T1) $\Rightarrow$ deadlock.	

Key idea: the *wait-for* relation forms a cycle (T1 waits for T2, and T2 waits for T1).

Deadlock (human edition)



DEADLOCK

Both are polite. Neither moves.
The DBMS solution: "Sorry—one of you has to `ROLLBACK`."

Deadlocks: typical handling strategies

1) Prevention (avoid cycles)

- **Lock ordering**: always request locks in a fixed global order
  (e.g., by primary key, table order, or object id).
- **Timestamp priority protocols**:
  - **wait-die**: *older may wait*; *younger aborts* if it would wait for older.
  - **wound-wait**: *older aborts (wounds) younger*; *younger may wait* for older.

2) Detection (allow, then resolve)

- DBMS maintains a **wait-for graph**.
- If there is a **cycle**, a deadlock is detected.
- DBMS chooses a **victim**, does ROLLBACK, releases locks.

3) Timeout (simple fallback)

- If a lock wait exceeds a threshold: abort + retry.

Practical takeaway: deadlocks are normal in concurrent systems; robust applications handle "transaction aborted" by retrying.

In SQL: setting isolation level (typical pattern)

Syntax varies by DBMS, but the idea is:

```sql
BEGIN;
-- (DBMS-specific) set isolation for this transaction:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- do work safely
SELECT ...;
UPDATE ...;

COMMIT;
```

Takeaway: you choose an isolation level to trade off speed vs anomalies.

Summary

- **DDL**: define structure + constraints; evolve schema via ALTER; use **views** and **indexes**.
- **DCL**: manage access via roles and GRANT/REVOKE; prefer least privilege and views for fine-grained rights.
- **TxCL**: group operations into transactions; ACID; isolation levels; locking/2PL; deadlocks exist and are handled by the DBMS.

A friendly SQL bar joke summarizing the lecture

- **DDL** walks into a bar and says: “I would like to `CREATE TABLE`.”
- **DCL** replies (smiling): “Only if I `GRANT` you permission.”
- **DML** jumps in: “Great, I will `INSERT` some customers, `UPDATE` the bill, and `DELETE` the evidence.”

A QUERY squints at the menu and says:

```sql
SELECT beer
FROM fridge
WHERE cold = TRUE
ORDER BY foam DESC
LIMIT 1;
```

TxCL sighs (kindly): “BEGIN... and if this gets messy, we ROLLBACK.”

Kind reminder: be nice to your future self – use transactions, and always leave the database consistent.

Next up in the course

- More query and data definition patterns and performance intuitions
- Interfacing a SQL database management system with a host programming language
- Practical exercises on your DBMS