Database Programming in SQL – Part I/II

Introduction to SQL (2×45 min)

Michael Emmerich

February 3, 2026

Roadmap

Part I (45 min)

- What is SQL? (declarative)
- SQL vs. relational model
- Basic statement rules
- Querying one table: SELECT / FROM / WHERE
- Sorting and limiting results
- NULL basics

Part II (45 min)

- Querying multiple tables (joins)
- IN and EXISTS subqueries
- Explicit JOIN syntax
- Set operations: UNION
- Aggregation: SUM, COUNT, MIN/MAX/AVG
- Grouping and HAVING

Part I

SQL basics & single-table queries

What is SQL?

- SQL (Structured Query Language) is a standardized language for interacting with relational databases.
- **Declarative**: you describe *what* you want; the DBMS decides *how* to compute it.

```sql
SELECT *
FROM table;
```

SQL vs. imperative code (intuition)

SQL (declarative)

```sql
SELECT *
FROM table;
```

Imperative (e.g., C#)

```csharp
for (int i = 0; i < table.Length; i++) {
  System.Console.WriteLine(table[i]);
}
```

In SQL, the optimizer can change the evaluation strategy without changing the result.

Relational model & SQL terminology

Relational model	**SQL**
Relation	Table
Attribute	Column
Tuple	Row

- SQL is often called *relationally complete*: relational algebra operations (e.g., cartesian product, set union, join and others) can be expressed in SQL.
- Practical note: unless constrained (e.g., by a PRIMARY KEY), tables may contain duplicate rows.

SQL dialects & sublanguages

- Different DBMSs implement (parts of) the standard with small differences: **SQL dialects**.
- Four common sublanguages:
  - **DML** (Data Manipulation Language): SELECT, INSERT, UPDATE, DELETE
  - **DDL** (Data Definition Language): CREATE, ALTER, DROP
  - **DCL** (Data Control Language): GRANT, REVOKE
  - **TxCL** (Transaction Control): BEGIN, COMMIT, ROLLBACK

SQL statements: basic rules

- Line breaks do not matter.
- Statements typically end with a semicolon `;`
- Keywords are case-insensitive: `SELECT` = `select`
- Identifier rules (typical): letters, digits, underscores; no spaces; do not start with a digit.
- Avoid using reserved keywords as table/column names.

CUSTOMER–INVOICE–PRODUCT: schema + foreign keys + table representation

Relations (PK underlined)

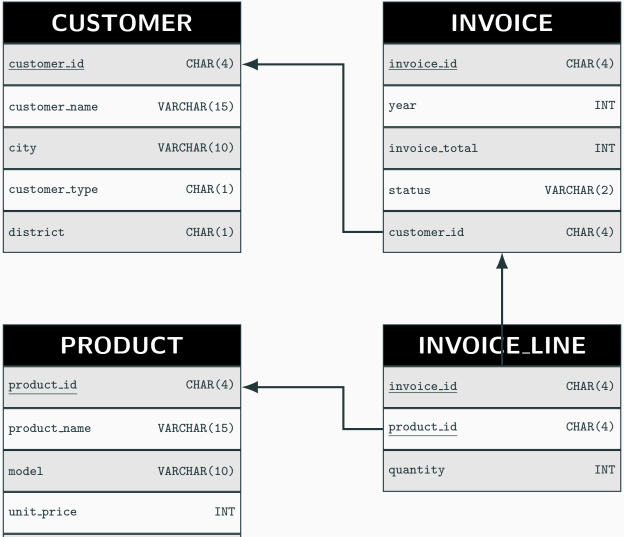
- CUSTOMER(customer_id, customer_name, city, customer_type, district)
- INVOICE(invoice_id, year, invoice_total, status, customer_id)
- PRODUCT(product_id, product_name, model, unit_price, color)
- INVOICE_LINE(invoice_id, product_id, quantity)

Foreign keys

- INVOICE.customer_id → CUSTOMER.customer_id
- INVOICE_LINE.invoice_id → INVOICE.invoice_id
- INVOICE_LINE.product_id → PRODUCT.product_id

Meaning (very short)

- A customer has invoices; each invoice has line items; each line item references a product.



Running example: Finnish names (blue) and English names (black)

We use a small sales database:

customers have **invoices**; each invoice consists of **invoice lines**; each line references one **product**.

In the rest of the lecture, we use the **English** names consistently.

Table (relation) names

Finnish	English
ASIAKAS	CUSTOMER
LASKU	INVOICE
TUOTE	PRODUCT
LASKU_RIVI	INVOICE_LINE

Attributes grouped by table

Table	Attributes (Finnish / English)
ASIAKAS	astun /customer_id, asnimi /customer_name, kaup /city, tyyppi /customer_type, mpiiri /district
LASKU	laskuno /invoice_id, vuosi /year, lask_summa /invoice_total, tila /status, astun /customer_id
TUOTE	tuotetun /product_id, tuotennimi /product_name, malli /model, ahinta /unit_price, vari /color
LASKU_RIVI	laskuno /invoice_id, tuotetun /product_id, maara /quantity

Foreign keys (arrows only here)

INVOICE.customer_id → CUSTOMER.customer_id

INVOICE_LINE.invoice_id → INVOICE.invoice_id

INVOICE_LINE.product_id → PRODUCT.product_id

SQL building blocks: DDL vs. DML (SQLite)

- We start from a **relational schema** (structure) and then write SQL.
- **DDL (Data Definition Language)** defines the *structure*: tables, attributes, primary keys, foreign keys, constraints.
- **DML (Data Manipulation Language)** changes the *data inside* tables: inserting, updating, deleting tuples.

Running example schema (lecture notation)

CUSTOMER(customer_id, customer_name, city, customer_type, district)

INVOICE(invoice_id, year, invoice_total, status, customer_id)

PRODUCT(product_id, product_name, model, unit_price, color)

INVOICE_LINE(invoice_id, product_id, quantity)

(relationship table)

Typical commands

```sql
-- DDL: define structure
CREATE TABLE ... ;
-- DML: insert data (tuples)
INSERT INTO ... VALUES ... ;
```

DDL in SQLite: CREATE TABLE and primary keys

- PRIMARY KEY enforces **uniqueness** and (in SQLite) implies **NOT NULL** for the key columns.
- In this running example: customer_id identifies a customer, product_id identifies a product.

DDL example: base tables

```
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 PRODUCT(
  product_id   CHAR(4)     PRIMARY KEY,
  product_name VARCHAR(15) NOT NULL,
  model        VARCHAR(10),
  unit_price   INT         NOT NULL,
  color        VARCHAR(10)
);
```

Foreign keys + inserts: INVOICE_LINE (SQLite)

- **Foreign keys (FK)** enforce **referential integrity**: e.g., INVOICE.customer_id must exist in CUSTOMER(customer_id).
- Composite PK (invoice_id, product_id) prevents product appearing twice in the same invoice.

```
PRAGMA foreign_keys = ON;
CREATE TABLE INVOICE_LINE(
  invoice_id  CHAR(4),
  product_id  CHAR(4),
  quantity    INT,
  PRIMARY KEY (invoice_id, product_id),
  FOREIGN KEY (invoice_id) REFERENCES INVOICE(invoice_id),
  FOREIGN KEY (product_id) REFERENCES PRODUCT(product_id)
);
INSERT INTO CUSTOMER(customer_id, customer_name, city, customer_type, district)
VALUES ('C001', 'Ada', 'Jyvaskyla', 'A', '1');
INSERT INTO PRODUCT(product_id, product_name, model, unit_price, color)
VALUES ('P010', 'Database Book', '2nd', 45, 'Blue');
INSERT INTO INVOICE(invoice_id, year, invoice_total, status, customer_id)
VALUES ('I100', 2026, 45, 'OK', 'C001');
INSERT INTO INVOICE_LINE(invoice_id, product_id, quantity)
VALUES ('I100', 'P010', 1);
```

General form of a query (one table)

```
SELECT column1, column2, ...
FROM   table;
```

- SELECT: which columns (projection)
- FROM: which table(s)

Selecting columns vs. selecting all

Pick specific columns

```sql
SELECT customer_id, customer_name, city,
    customer_type, district
FROM  customer;
```

Pick all columns

```sql
SELECT *
FROM  customer;
```

Prefer explicit column lists in real applications (clarity, stability, performance).

Filtering rows with WHERE

```sql
SELECT *
FROM   product
WHERE unit_price > 100
  AND unit_price < 2000;
```

- Comparison operators: =, <, <=, >, >=, <> (or !=)
- Combine conditions with AND, OR, and NOT

Operator precedence & parentheses

- Like arithmetic, parentheses control evaluation order.
- In many SQL dialects, AND binds tighter than OR.
- When in doubt: add parentheses.

```
SELECT product_id, product_name, unit_price
FROM   product
WHERE (product_name LIKE 't%' OR product_name LIKE 's%')
  AND (unit_price > 200 OR unit_price < 20);
```

Strings and pattern matching with LIKE

Wildcards (common):

- % matches any sequence of characters (including empty)
- _ matches exactly one character

```
-- Names starting with K, or ending in 'Ltd'
SELECT customer_name, district, customer_type
FROM   customer
WHERE  customer_name LIKE 'K%'
   OR  customer_name LIKE '%Ltd';
```

Other handy predicates: IN and BETWEEN

Membership

```
SELECT *
FROM   customer
WHERE district IN ('I', 'L');
```

Range (inclusive)

```
SELECT *
FROM   product
WHERE unit_price BETWEEN 100 AND 1000;
```

NULL values (missing/unknown)

- NULL represents "no value" (unknown, missing, not applicable).
- Comparing with NULL using = does *not* work as you might expect.
- Use IS NULL / IS NOT NULL.

```
SELECT *
FROM   product
WHERE unit_price IS NULL;
```

Ordering and limiting results

Order by one or more columns

```sql
SELECT customer_id, customer_type, district
FROM   customer
WHERE  customer_name <> 'Kajo'
ORDER BY customer_type, customer_name;
```

Limit output size

```sql
SELECT invoice_id, year, status
FROM   invoice
ORDER BY year DESC
LIMIT  1;
```

End of Part I

Next: multi-table queries, joins, set operations, aggregation.

Part II

Multi-table queries, joins & aggregation

Why multiple tables?

- Real databases split information across tables (normalization, clarity, integrity).
- Queries often need to combine rows from multiple tables based on a **join condition**.
- SQL offers several equivalent ways to express joins.

Join idea (conceptually)

- A join condition matches related rows (often via key/foreign key).
- Typical pattern: `table1.key = table2.foreign_key`
- The DBMS chooses an efficient algorithm (nested-loop, hash join, sort-merge, . . .).

Joining via a subquery: IN

```sql
-- Products that have been billed at least once
SELECT product_name
FROM   product
WHERE product_id IN (
  SELECT product_id
  FROM   invoice_line
);
```

Joining via a correlated subquery: EXISTS

```
SELECT p.product_name
FROM   product p
WHERE  EXISTS (
  SELECT *
  FROM   invoice_line il
  WHERE  p.product_id = il.product_id
);
```

EXISTS returns true if the subquery finds at least one matching row.

Joining with comparison operators (classic style)

```
SELECT DISTINCT p.product_name
FROM   product p, invoice_line il
WHERE  p.product_id = il.product_id;
```

- Works, but can become hard to read for many tables.
- Use DISTINCT if duplicates can appear.

Explicit JOIN syntax (recommended)

```
SELECT DISTINCT p.product_name
FROM   product p
JOIN   invoice_line il
       ON p.product_id = il.product_id;
```

SQL also defines other join types (e.g., LEFT OUTER JOIN); we focus on inner joins here.

Set operations: UNION

- Combine results of multiple queries into one result table.
- The queries must return the same number of columns (compatible types).

```
SELECT model AS models_and_product_names
FROM   product
UNION
SELECT product_name
FROM   product;
```

Aggregate functions (analytics)

- Aggregates compute one value from many rows.
- Common aggregates: SUM, COUNT, MIN, MAX, AVG

```
-- Sum of unit prices
SELECT SUM(unit_price) AS total_unit_price
FROM   product;
```

Counting rows and distinct values

Count rows

```
SELECT COUNT(*) AS customer_count
FROM   customer;
```

Count distinct values

```
SELECT COUNT(DISTINCT city) AS
    city_count
FROM   customer;
```

MIN, MAX, AVG

```
SELECT MAX(unit_price) - MIN(unit_price)
    AS price_range
FROM  product;
```

```
SELECT AVG(unit_price) AS avg_unit_price
FROM  product;
```

Grouping with GROUP BY

- Group rows by one (or more) columns, then aggregate per group.

```
-- Sum of unit prices by color
SELECT SUM(unit_price) AS total_price, color
FROM  product
GROUP BY color;
```

Filtering groups with HAVING

- WHERE filters rows *before* grouping.
- HAVING filters groups *after* grouping.

```
SELECT COUNT(DISTINCT p.product_id) AS product_count, p.color
FROM  product p, invoice_line il
WHERE p.product_id = il.product_id
GROUP BY p.color
HAVING product_count > 2
ORDER BY product_count DESC;
```

Example: GROUP BY (average grade per subject)

Table: `Enrolled`

Name	Subject	Grade
Anna	Topology	4
Bernd	Music	5
Corina	Music	4
Donald	Topology	1
Emmi	Databases	5

SQL query

```
SELECT Subject, AVG(Grade) AS avgGrade
FROM Enrolled
WHERE Name <> 'Emmi'
GROUP BY Subject;
```

After WHERE (rows removed before grouping)

Name	Subject	Grade
Anna	Topology	4
Donald	Topology	1
Bernd	Music	5
Corina	Music	4

Result after GROUP BY

Subject	AvgGrade
Topology	2.5
Music	4.5

Example: HAVING (filter groups after aggregation)

- WHERE filters **rows before** grouping.
- HAVING filters **groups after** GROUP BY (often using aggregates like AVG, COUNT).

SQL query with HAVING

```
SELECT Subject, AVG(Grade) AS avgGrade
FROM Enrolled
WHERE Name <> 'Emmi'
GROUP BY Subject
HAVING AVG(Grade) >= 3;
```

Groups and averages (from previous slide)

Subject	AvgGrade
Topology	2.5
Music	4.5

After HAVING (keep only avgGrade >= 3)

Subject	AvgGrade
Music	4.5

(Optional) Finding "missing" matches: NOT IN / NOT EXISTS

- Useful pattern: find rows in one table that have *no related row* in another table.

```
-- Customers who have never been billed in year 2011
SELECT customer_id, customer_name
FROM  customer
WHERE customer_id NOT IN (
  SELECT customer_id
  FROM  invoice
  WHERE year = 2011
);
```

Summary

- Single-table queries: SELECT / FROM / WHERE + ORDER BY + LIMIT
- Be careful with NULL: use IS NULL / IS NOT NULL
- Multi-table queries: joins via IN, EXISTS, comparisons, or explicit JOIN
- Set operation: UNION
- Aggregation: SUM, COUNT, MIN/MAX/AVG with GROUP BY and HAVING

Next up in the course

- Data definition (tables, keys, constraints)
- Updates and transactions
- Views and access control