Lab 1.2 — Benchmarking a Single-Node Pipeline to its Breaking Point

Session 1.2 — Distributed Programming with Scala
Hands-on lab
Tools: Scala 3 (`scala-cli`), SQLite (via JDBC), system monitor (`btop` / `htop` / Activity Monitor / Task Manager)

---

1. Objective

You will process the same dataset three ways — naive hand-rolled code, then a proper database (SQLite), then push both until they break. By scaling from 100K to 100M rows you will experience firsthand the progression from single-file processing to database to the point where even a database is not enough — and distributed processing becomes necessary.

2. Setup

Install `scala-cli` — a single binary that compiles and runs Scala files with no project setup:

```
curl -sSLf https://scala-cli.virtuslab.org/get | sh
```

Create a working directory for this lab. Each program is a standalone `.scala` file run with `scala-cli run .` or `scala-cli run MyFile.scala`.

To use SQLite from Scala, add a JDBC dependency directive at the top of your file:

```
//> using dep org.xerial:sqlite-jdbc:3.49.1.0
```

`scala-cli` resolves and downloads it automatically on first run.

2.1. IDE support

VS Code / Cursor — install the **Metals** extension (`scalameta.metals`). Open your lab folder, Metals will detect `scala-cli` directives and provide completions, go-to-definition, and inline errors. No additional configuration needed.

IntelliJ IDEA — install the **Scala** plugin (bundled with IDEA Ultimate, available in the marketplace for Community). Open the folder, then right-click a `.scala` file and select *Set up scala-cli project* — IDEA imports the dependency directives and enables full IDE support.

3. Data Model

Two tables:

customers

Column	Type	Notes
`customer_id`	UUID	primary key
`name`	TEXT	first + last
`city`	TEXT	from provided list

orders

Column	Type	Notes
`order_id`	UUID	primary key
`customer_id`	UUID	FK → customers
`amount`	INT	cents (e.g. 14999 = $149.99)
`ts`	TIMESTAMP	random within 2024

4. Material

Three text files are provided as a `lab-1.2-single-node-benchmark-assets.tar.gz` archive, downloadable from the course website. Extract it into your working directory. Each file contains one entry per line, sorted alphabetically:

- `cities.txt` — 150 world cities
- `first_names.txt` — 98 first names
- `last_names.txt` — 97 last names

With 98 × 97 = 9,506 unique full name combinations, you can fill 10,000 customers with minimal collisions. You are free to use your own lists instead.

5. Walkthrough

5.1. Step 1 — Data Generation

5.1.1. 1.1 — Customer generator

Write a program that produces `customers.csv` with 10,000 rows:

- `customer_id`: a random UUID v4 (use your language's standard library)
- `name`: a random first name + last name from the provided lists
- `city`: picked uniformly at random from `cities.txt`

Hints: `scala.io.Source.fromFile` reads text files into lines. `java.util.UUID.randomUUID()` generates UUIDs. `scala.util.Random.nextInt(n)` picks a random index. `java.io.PrintWriter` writes to a file.

Verify: `wc -l customers.csv` should print 10,000 (plus a header if you added one). Open it, spot-check a few rows.

5.1.2. 1.2 — Order generator

Write a program that produces `orders.csv` with a *configurable* row count (CLI argument or constant). Each row:

- `order_id`: a random UUID v4
- `customer_id`: a random UUID picked from the customer file you just generated
- `amount`: random integer in [100, 50000] (i.e. $1.00 to $500.00)
- `ts`: random timestamp within 2024 (uniform random second between `2024-01-01T00:00:00` and `2024-12-31T23:59:59`)

Hints: `args(0).toInt` reads a CLI argument. `java.time.LocalDateTime` and `java.time.temporal.ChronoUnit` handle timestamps. Load `customer_id` values from the CSV you generated in 1.1 — read the file, extract the first column into a `Vector`, and pick random entries.

Generate a first file with **100,000 rows**. Verify: `wc -l`, `head -5`, check file size on disk.

5.1.3. 1.3 — Sanity checks

- Count distinct `customer_id` values in `orders.csv` — should be close to 10,000.
- Check that `amount` values are in range.
- Note the file sizes. Predict: how big will 10M rows be? 100M?

5.2. Step 2 — Hand-Rolled Aggregate (100K rows)

5.2.1. 2.1 — Load customers into memory

Read `customers.csv` and build an in-memory lookup: `customer_id → city`. This is your "index."

Hint: parse each line into a `Map[String, String]` mapping customer ID to city. `System.nanoTime()` before and after gives you wall-clock time.

Measure: how long does this take? How much memory does it consume? (Check your system monitor — it should be negligible.)

5.2.2. 2.2 — Scan, filter, join, aggregate

Read `orders.csv` line by line. For each row:

1. Parse the line (split on delimiter, cast `amount` to int)
2. Filter: skip rows where `amount <= 5000` (i.e. $50.00)
3. Look up `city` from the customer hashmap using `customer_id`
4. Accumulate `sum(amount)` into a `city → total_revenue` hashmap

Hint: `Source.fromFile(...).getLines()` gives you a lazy iterator. You can chain `.map`, `.filter`, `.groupBy` from the collection API, or use a mutable `HashMap` with `updateWith` — both approaches work. `toSeq.sortBy(_._1)` sorts by city name.

Print the result: revenue per city, sorted by city name.

Measure and record wall-clock time. At 100K rows this should be sub-second.

5.2.3. 2.3 — Observe on your system monitor

Open your system monitor on a second terminal or screen. Run the program again. Note:

- **CPU:** one core briefly spikes
- **Memory:** barely moves
- **Disk I/O:** a small read burst, then nothing

Record these observations — you will compare them against later steps.

5.3. Step 3 — Same Query in SQLite (100K rows)

5.3.1. 3.1 — Create the database and tables

Create a SQLite database. Define the schema:

```
CREATE TABLE customers (
    customer_id TEXT PRIMARY KEY,
    name TEXT,
    city TEXT
);

CREATE TABLE orders (
    order_id TEXT PRIMARY KEY,
    customer_id TEXT,
    amount INTEGER,
    ts TEXT
);
```

5.3.2. 3.2 — Bulk load the CSVs

Import `customers.csv` and `orders.csv` into the tables using JDBC.

Hint: `java.sql.DriverManager.getConnection("jdbc:sqlite:lab.db")` opens (or creates) the database. Use a `PreparedStatement` with batched inserts (`addBatch` / `executeBatch`) inside a transaction for speed — row-by-row commits are orders of magnitude slower.

Measure how long the import takes. This is overhead your hand-rolled code does not pay — but it is a one-time cost.

5.3.3. 3.3 — Run the equivalent query

```
SELECT c.city, SUM(o.amount) AS total_revenue
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.amount > 5000
GROUP BY c.city
ORDER BY c.city;
```

Measure wall-clock time. Compare against step 2.2. At 100K rows, both should be fast.

5.3.4. 3.4 — Compare results

Verify that both approaches produce the same numbers. This confirms your hand-rolled code is correct and the comparison is fair.

5.4. Step 4 — Scale to 10M Rows

5.4.1. 4.1 — Regenerate orders

Run your order generator with **10,000,000 rows**. Note:

- How long does generation itself take?
- How big is the CSV on disk?

5.4.2. 4.2 — Rerun hand-rolled code

Run your step 2 code on the 10M-row file. Measure wall-clock time.

Observe on your system monitor:

- **CPU:** one core pinned at 100% for the entire duration

- **Memory:** the two hashmaps are tiny (10K customers, 150 cities), but the file read buffer may grow
- **Disk I/O:** sustained read throughput — how close to your disk's maximum?

Record the time.

5.4.3. 4.3 — Rerun in SQLite

Re-import into SQLite (drop and recreate, or use a fresh DB). Measure import time separately from query time.

Run the same SQL query. Measure.

Without an index: SQLite does a full table scan + hash join. It is faster than hand-rolled code because it avoids CSV re-parsing, but it is not spectacular.

5.4.4. 4.4 — Add an index

```
CREATE INDEX idx_orders_customer ON orders(customer_id);
```

Measure how long index creation takes. Then rerun the query. Compare against 4.3.

Key insight: the index is an up-front cost that pays off on every subsequent query. Your hand-rolled code has no equivalent — every run re-reads and re-parses the entire file.

5.4.5. 4.5 — Run a second query

Compute something different — top 10 customers by total spend:

```
SELECT c.name, SUM(o.amount) AS total
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
GROUP BY o.customer_id
ORDER BY total DESC
LIMIT 10;
```

In SQLite this reuses the loaded data and index. Near-instant.

With your hand-rolled code you would have to re-read and re-parse the entire CSV, build a different hashmap, then sort it. Write and run this code. Measure.

The gap becomes clear: SQLite pays the import cost once. Your code pays the full parsing cost on every single query.

5.5. Step 5 — Scale to 100M Rows (The Wall)

5.5.1. 5.1 — Regenerate orders

Run your order generator with **100,000,000 rows**. Note:

- Generation time
- File size on disk (expect 5–10 GB)
- Does the generator itself start struggling? (Watch memory.)

5.5.2. 5.2 — Rerun hand-rolled code

Run your step 2 code.

Observe on your system monitor:

- **CPU:** single core at 100% for tens of seconds
- **Disk:** sustained sequential read — the bottleneck may shift to I/O if the file does not fit in the OS page cache

- **Memory:** watch for growth if your code buffers lines or allocates intermediate strings

Record wall-clock time. This is now painful enough that you would not want to iterate on it.

5.5.3. 5.3 — Load into SQLite

Bulk-insert 100M rows. This takes minutes, but you only do it once.

Create the index. This also takes time (SQLite sorts 100M UUID strings for the B-tree).

5.5.4. 5.4 — Run queries in SQLite

Run the same two queries from steps 3.3 and 4.5. Measure.

SQLite handles them in seconds. Your hand-rolled code took minutes. The database wins by an order of magnitude or more.

5.5.5. 5.5 — Modify the query, run again

Change the filter from `amount > 5000` to `amount > 20000`. In SQLite: change one number, re-execute, seconds. With hand-rolled code: re-read the entire CSV, re-parse 100M lines, apply the new filter. Full cost again.

This is the moment: a database separates *storage* from *query*. Your hand-rolled code welds them together. Every new question means starting from scratch.

5.6. Step 6 — Push SQLite to its Limits (Foreshadowing)

5.6.1. 6.1 — Self-join: concurrent orders

Find pairs of orders from customers in the same city placed within 60 seconds of each other:

```
SELECT o1.order_id, o2.order_id, c1.city
FROM orders o1
JOIN orders o2 ON o1.order_id < o2.order_id
JOIN customers c1 ON o1.customer_id = c1.customer_id
JOIN customers c2 ON o2.customer_id = c2.customer_id
WHERE c1.city = c2.city
  AND ABS(unixepoch(o1.ts) - unixepoch(o2.ts)) < 60;
```

Even at 10M rows this query is extremely slow. At 100M it is hopeless.

5.6.2. 6.2 — Observe on your system monitor

- **CPU:** pegged at 100%
- **Disk:** massive write bursts as SQLite spills intermediate join results to temp files
- **Memory:** SQLite tries to stay within its cache limit, but temp file I/O dominates

Let it run for a minute or two, then cancel. The point is made.

5.6.3. 6.3 — Discussion

SQLite is single-threaded, single-node, and stores everything on one disk. For this query:

- The join is quadratic in the number of orders per city
- No amount of indexing fixes a quadratic join
- You would need to partition by city and parallelize across cores or machines
- This is exactly what distributed engines (Spark, Flink) do

This is the hook for the rest of the course.

6. Key Takeaways

1. **Don't roll your own database.** Hand-rolled CSV processing is a terrible, unindexed, optimizer-free database. SQLite — a single C library — demolishes it.
2. **Databases separate storage from queries.** Load once, query many times. Hand-rolled code pays the full parsing cost on every question.
3. **Indexes are the first superpower.** An up-front $O(n \log n)$ cost that turns every future lookup from $O(n)$ into $O(\log n)$.
4. **Even a great single-node database has limits.** Quadratic joins, data that exceeds one disk, queries that need more than one core — these are the problems the rest of this course addresses.
5. **Your system monitor tells you which wall you hit.** CPU, memory, or I/O — the bottleneck determines your next move.