Distributed SQL Query Engine over a Column Store

(Simplified, Sorted Storage Edition)

---

1. Overview

In this project, you will design and implement a **distributed SQL query engine** operating over a **custom-built, column-oriented storage layer**.

Your system will:

- Accept simple SQL analytical queries
- Parse, plan, and execute them **across multiple worker nodes**
- Aggregate and return final results to the user

This project models the core ideas behind analytical data engines such as **Google BigQuery**, **Amazon Redshift**, **Apache Drill**, and **Snowflake** — but in a simplified, educational form.

You will build:

- A **columnar storage layer** that stores each column in its own file, sorted by a key column
- A **local query executor** that performs scans, filters, and aggregations
- A **distributed query coordinator** that divides queries among workers and merges results

You may use any programming language (Python, Go, Java, Rust, C++, etc.) and standard libraries for networking and parsing — but **you may not use existing DBMSs** (e.g., SQLite, DuckDB, PostgreSQL).

---

2. Learning Objectives

By completing this project, you will:

- Understand distributed query execution and result aggregation
- Design and implement a **columnar storage model** optimized for analytical scans

- Implement a **SQL-like execution pipeline** (parse → plan → execute)
- Build a **coordinator–worker architecture** for distributed analytics
- Experiment with **predicate pushdown and partial aggregation**
- Learn how **sorted data** replaces indexing for efficient range filtering

---

3. System Architecture

(a) Coordinator Node

- Accepts queries (SQL)
- Parses and validates them
- Generates a logical plan and distributes subqueries to workers
- Merges partial results into a final output

(b) Worker Nodes

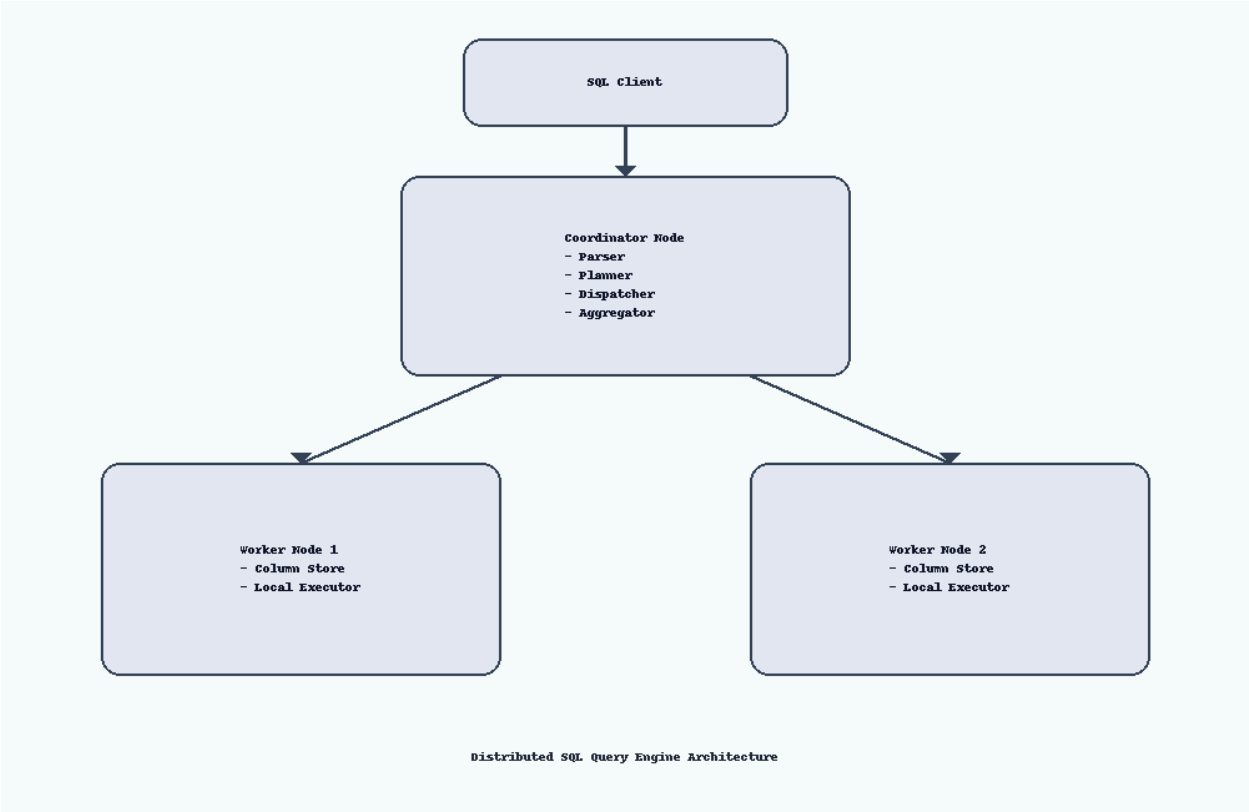
- Store table partitions (segments) as **sorted column files**
- Execute local operations:
    - Scan and filter rows by predicate
    - Perform local aggregations (`SUM`, `AVG`, `COUNT`, etc.)
- Return results to the coordinator

(c) Storage Layer (Column Store)

- Each table is stored as a directory
- Each column is a separate file (binary or text)
- **Rows in each segment must be sorted** by a key column defined in the schema
- This sorted property replaces indexes — predicates can prune entire data ranges

---

Example Architecture Diagram



---

4. Functional Requirements

Core Features

Your system must support:

- `SELECT` with projections and aggregates
- `FROM` a single table
- `WHERE` with simple predicates (`=`, `<`, `>`, `BETWEEN`, `AND`)
- `GROUP BY` with `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`
- Execution across multiple workers

Optional (Extra Credit)

- Joins between two tables
- Column compression (e.g., run-length encoding)

- Fault tolerance (retry or skip failed workers)
- Cost-based query optimization

---

5. Creating Tables and Loading Data

5.1 Schema Definition (Simple Schema Format)

Each table directory contains a text file `_schema.ssf`.
Each line defines a column:

```
column_name: type [nullable] [key]
```

Supported types:
`int32`, `int64`, `float64`, `bool`, `string`, `date`, `timestamp(ms)`

Example (`sales` table):

```
id: int64 key
region: string
amount: float64
ts: timestamp(ms) nullable
```

- The column marked **key** defines the **sort order** for all data in the table.
- All segments must remain sorted by this key.

5.2 Table Layout

```
data/sales/
  _schema.ssf
  _table.txt
  seg-000001/
    id.bin
    region.bin
    amount.bin
    ts.bin
  seg-000002/
    ...
```

`_table.txt` (simple metadata):

```
version=1
block_rows=65536
segment_target_rows=1000000
endianness=little
```

Each column file stores binary data for that column in **sorted row order**. Each segment is dedicated to one worker.

5.3 CLI Commands

```
# 1) Initialize a new table
minidist init data/sales --schema schema.ssf

# 2) Load CSV data (sorted by key)
minidist load data/sales --csv sales.csv --sort-key id --segments n

# 3) Inspect schema and metadata
minidist schema show data/sales
minidist info data/sales
```

Ingest rules:

- Data must be **sorted by the key column**.
- If unsorted, the loader sorts before writing.
- Missing values in nullable columns → stored as NULLs.

---

6. Implementation Layers

6.1 Storage Layer

Responsibilities

- Maintain sorted column files.

- Each segment is meant for a different worker

6.2 Worker Nodes

Responsibilities

- Execute local queries on sorted column data.
- Perform filters and local aggregates.
- Return partial aggregates to the coordinator.

Example Query

```
SELECT SUM(amount), COUNT(*)
FROM sales
WHERE id BETWEEN 100 AND 200;
```

Example response:

```
sum_amount | count
-----------|------
123456.0   | 789

Execution Details:
Rows scanned:       200000
Segments skipped:   4
Execution time:     350 ms
```

6.3 Coordinator Node

Responsibilities

- Accept queries (SQL)
- Parse → plan → distribute → merge results

Coordinator flow:

1. Parse query and identify predicates, aggregates, and columns.
2. Send subqueries to each worker.
3. Merge results:
    - `SUM`, `COUNT`: add partials

- `MIN`, `MAX`: take global min/max
- `AVG`: sum(partial_sum) / sum(partial_count)

4. Return final results to client.

Example CLI

```
python coordinator.py --port 8080 --workers 9001,9002
curl -X POST localhost:8080/query -q QUERY
```

---

7. Implementation Stages and Grading

Stage	Description	Difficulty	Weight
1. Column Store	Implement sorted columnar storage and CSV ingestion.	🟢 Easy–Medium	25%
2. Local Query Engine	Scan, filter, and aggregate on one node.	🟡 Medium	25%
3. Distributed Layer	Add coordinator and multiple workers. Execute distributed queries.	🟠 Medium–Hard	25%
4. Optimizations	Projection & filter pushdown, partial aggregation.	🔴 Hard	25%
5. Advanced Features (Bonus)	Joins, compression, or fault tolerance.	🟣 Bonus	+15%

---

8. Deliverables

1. **Source Code Repository**
   Modular, documented code (with `storage/`, `worker/`, `coordinator/` folders)
2. **README / User Guide**
   Build, run, and query instructions
3. **Design Report (≈10 pages)**
   Architecture diagrams, data model, execution flow, optimizations, trade-offs, performance notes

---

9. Example End-to-End Run

Data (`sales.csv`):

```
id,region,amount
1,EU,100
2,US,200
3,EU,50
4,APAC,300
```

Schema (`_schema.ssf`):

```
id: int64 key
region: string
amount: float64
```

Query:

```
SELECT region, SUM(amount)
FROM sales
WHERE amount > 100
GROUP BY region;
```

Execution Flow:

1. Coordinator identifies `region`, `amount` columns.
2. Sends subqueries to each worker.
3. Workers compute local sums per region.
4. Coordinator merges:

region	sum(amount)
US	200
APAC	300

---

10. References (Optional Reading)

- Stonebraker et al., *C-Store: A Column-oriented DBMS*, VLDB 2005
- Dean & Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004
- Abadi et al., *Column-Stores vs Row-Stores: How Different Are They Really?*, SIGMOD 2008
- Apache Drill and DuckDB docs (for inspiration only — do not reuse code)

---

11. Example dataset sources

- **NYC TLC Trip Record Data** (CSV & Parquet; includes schema and data dictionaries). New York City Government+1
  Cloud mirrors exist if you prefer object storage. Registry of Open Data+1
- **MovieLens** (100K "latest-small" or larger). GroupLens
- **TPC-H** (synthetic; generate CSVs with `dbgen`, available from TPC after free registration, or community mirrors of `dbgen` source). tpc.org+2tpc.org+2

You do not need to use this data but may be useful. You may have to sort some of the files.

12. Deliverables

- All files for your project need to be zipped in one single zip file. The file name should be in the following format: Lastname_Firstname_Project.zip

- Please upload your submission to eClass by the deadline. There are absolutely no exceptions and no extensions.

- Inside the zip file, please include a) your source code b) any results that are part of the project c) a README text file that explains how to compile (if needed) and run your code and d) a short and concise report (in PDF - no more than 10 pages) describing your work and your approach to solving the project. Please do not include binaries, nor the original or any derived data in the zip file.I will compile your code and run it myself.

13. Important Notes

- Please check the eClass Website regularly for announcements and/or clarifications to the project. Announcements will show up there and will also be sent to the mailing list.

- You are free to make any assumptions you may need to along the way as long as you document them in your report.

- The project is meant to be worked on by the student submitting and only that student. In the event that there is a submission not worked on by the student, then the student fails the class.

- Copying code from existing sources (e.g., the Web, github, ChatGPT, etc.) is strictly not allowed.

- You are responsible to safeguard your code, so if you are using a code repository service (e.g., github) keep your repository private. Cases when code was cloned and was also submitted by someone else cannot be reliably traced and result in everyone involved (regardless of who is at fault or who copied from whom) failing the class.

- Although students are expected (and encouraged) to chat among them on potential solutions and approaches, sharing code and solutions is strictly not allowed. In the event that two or more students provide submissions that have common pieces, everyone involved (regardless of who is at fault or who copied from whom) fails the class.

✅ End of Handout

Good luck!
Start small, test often, and let sorted data do the heavy lifting.
Build a clean design, not just a working prototype.
