Overview

Duration: 6 hours **Challenge:** Build a custom query engine to process a dataset of millions of taxi trip records. The engine must support filtering, projection, aggregation, and group-by operations without relying on external data processing engines (e.g., DuckDB, Polars, Pandas, Spark, Hadoop, etc.) for reading the JSON file.

> **Important:**
>
> - **File Reader Requirement:** The file reading code must be completely custom-built. While you can use external libraries for processing the data *after* the file is read, you are **not allowed** to use them for parsing or reading the JSON files.
> - **Dataset Download:** You can download the dataset from the following link (make sure to unzip it before use):
>   taxi-trips-data.json.zip

Programming Language Requirements

- Participants can use any programming language of their choice (Python, Java, C++, JavaScript, Go, Rust, etc.)
- Internet access will be available during the hackathon for reference and documentation lookup
- The only restriction is that the JSON file reading and parsing code must be written from scratch

Dataset Schema

The dataset is based on publicly available Taxi Trip Records and is provided to you as a newline separated JSON with ~20 million rows. For the purpose of this challenge, it contains the following columns:

- **tpep*pickup*datetime:** The date and time when the meter was engaged.
- **tpep*dropoff*datetime:** The date and time when the meter was disengaged.
- **VendorID:** Numeric code indicating the taxi provider (e.g., 1 for Creative Mobile Technologies, LLC; 2 for VeriFone Inc.).
- **Passenger_count:** The number of passengers in the vehicle.
- **Trip_distance:** The distance of the trip (in miles) as reported by the taximeter.
- **Payment_type:** Numeric code for the payment method (1= Credit card, 2= Cash, 3= No

charge, 4= Dispute, 5= Unknown, 6= Voided trip).
- **Fare_amount:** The fare amount (in USD) calculated by the meter.
- **Tip_amount:** The tip amount (in USD) (populated for credit card tips; cash tips are not included).
- **Store*and*fwd_flag:** Indicates whether the trip record was stored in the vehicle memory before sending (values: 'Y' for store-and-forward, 'N' otherwise).

Query Operations

Your query engine should implement the following operations on the dataset:

1. **Filtering**: Select records that match specific conditions on any column
2. **Projection**: Select specific columns from the dataset
3. **Aggregation**: Compute aggregated values (count, sum, average, min, max) over the dataset
4. **Group-By**: Group records by specific columns and perform aggregations within groups

Queries

Your solution should implement four predefined queries that demonstrate these operations. Your program should accept a query name parameter (`query1` , `query2` , `query3` , or `query4`) to execute the corresponding query.

Query 1: Overall Record Count

Objective:
Count the total number of records in the dataset.

Query:

```
SELECT COUNT(*) AS total_trips
FROM dataset;
```

Output Format:

```
┌──────────────────┐
│   total_trips    │
│      int64       │
├──────────────────┤
│    20332093      │
│ (20.33 million)  │
└──────────────────┘
```

Query 2: Trip Distance Filter with Payment Type Grouping

Objective:
Filter trips with a `Trip_distance` greater than 5 miles. Group by `Payment_type` and compute:

- Count of trips per payment type
- Average fare amount per payment type
- Total tip amount per payment type

Query:

```
SELECT
    Payment_type,
    COUNT(*) AS num_trips,
    AVG(Fare_amount) AS avg_fare,
    SUM(Tip_amount) AS total_tip
FROM dataset
WHERE Trip_distance > 5
GROUP BY Payment_type;
```

Output Format:

```
┌──────────────┬───────────┬────────────────────┬──────────────────────┐
│ payment_type │ num_trips │      avg_fare      │      total_tip       │
│    int64     │   int64   │       double       │        double        │
├──────────────┼───────────┼────────────────────┼──────────────────────┤
│            0 │    325550 │  35.56091902933489 │  621999.0699999994 │
│            1 │   2460202 │ 50.042180841247884 │ 24342962.609993026 │
│            2 │    460863 │ 48.709813241679306 │              3311.95 │
│            3 │     16865 │ 17.337380966498664 │               857.78 │
│            4 │     61820 │ 3.2400266903914594 │              7660.67 │
└──────────────┴───────────┴────────────────────┴──────────────────────┘
```

Note: Precision of float and double values up to at least 2 decimal places is expected.

Query 3: Store-and-Forward Flag and Date Filter with Vendor Grouping

Objective:
Filter trips with:

- `Store_and_fwd_flag` equal to `'Y'` (store and forward trips)
- Pickup date in January 2024 (between '2024-01-01' and '2024-01-31' inclusive)

Group by `VendorID` and compute:

- Count of trips per vendor
- Average passenger count per vendor

Query:

```
SELECT
    VendorID,
    COUNT(*) AS trips,
    AVG(Passenger_count) AS avg_passengers
FROM dataset
WHERE Store_and_fwd_flag = 'Y' AND tpep_pickup_datetime >= '2024-01-01'
  AND tpep_pickup_datetime < '2024-02-01'
GROUP BY VendorID;
```

Output Format:

```
┌──────────┬─────────┬──────────────────────┐
│ VendorID │ trips   │  avg_passengers      │
│  int64   │ int64   │      double          │
├──────────┼─────────┼──────────────────────┤
│        1 │ 10104   │  1.21229216152019    │
│        2 │  1232   │ 1.224025974025974    │
└──────────┴─────────┴──────────────────────┘
```

Query 4: Daily Statistics for January 2024

Objective:
Extract the date from `tpep_pickup_datetime` and for January 2024:

- Group trips by day
- For each day, compute:
  - Total number of trips
  - Average number of passengers
  - Average trip distance
  - Average fare amount
  - Total tip amount

Query:

```
SELECT
    CAST(tpep_pickup_datetime AS DATE) AS trip_date,
    COUNT(*) AS total_trips,
    AVG(Passenger_count) AS avg_passengers,
    AVG(Trip_distance) AS avg_distance,
    AVG(Fare_amount) AS avg_fare,
    SUM(Tip_amount) AS total_tip
FROM dataset
WHERE tpep_pickup_datetime >= '2024-01-01'
  AND tpep_pickup_datetime < '2024-02-01'
GROUP BY CAST(tpep_pickup_datetime AS DATE)
ORDER BY trip_date ASC;
```

Output Format:

trip_date date	total_trips int64	…	avg_fare double	total_tip double
2024-01-01	81013	…	21.788021058348768	264284.6399999976
2024-01-02	75519	…	20.967073451714114	265859.3800000038
2024-01-03	82427	…	19.66402258968533	277524.6
2024-01-04	102901	…	18.42131349549564	339759.10000000935
2024-01-05	103178	…	17.83245468995312	333356.7199999961
2024-01-06	97117	…	17.310473449550663	295446.2000000026
2024-01-07	67543	…	19.452978546999567	236380.09000000532
2024-01-08	80034	…	18.698506009945884	273058.9000000012
2024-01-09	93962	…	16.898261531257415	286757.1600000021
2024-01-10	95000	…	17.92064168421048	323644.9799999951
2024-01-11	105010	…	18.556641938862928	370943.91000000853
2024-01-12	103655	…	18.548339973952142	360971.7599999962
2024-01-13	104758	…	17.27619885832127	337550.1799999966
2024-01-14	94420	…	17.870277165854763	308173.5400000066
2024-01-15	77033	…	19.298005789726666	279363.86999999697
2024-01-16	93057	…	19.172022523829376	332362.8399999965
2024-01-17	110365	…	18.203946359806057	369867.4299999938
2024-01-18	110358	…	17.895973558781506	370351.3899999922
2024-01-19	95951	…	17.221129951746214	311721.26000000676
2024-01-20	108768	…	16.534302092527188	318073.76000000094
2024-01-21	84502	…	18.430728385127043	282291.9899999996
2024-01-22	85650	…	18.535876474022135	295781.6300000113
2024-01-23	99891	…	17.677200548597973	335802.6799999926
2024-01-24	105120	…	17.716179509132402	360508.6900000098
2024-01-25	110318	…	18.263443046465635	383378.97999999655
2024-01-26	105804	…	17.815411326603893	354378.40999999776
2024-01-27	110515	…	16.816794824231966	342094.0500000055
2024-01-28	92837	…	18.069742344108526	297860.55000001454
2024-01-29	84547	…	18.543849338237887	296439.159999995
2024-01-30	101233	…	17.334327640196282	339472.65000000736
2024-01-31	102131	…	17.52606094134025	346143.6299999983

31 rows 6 columns (4 shown)

Note: The output will include a line for each day in January 2024 (31 rows).

Evaluation Criteria

Participants' submissions will be evaluated based on the following:

1. **Completion:**

- Successfully implementing all four queries with correct results.
- Properly handling the core operations (filtering, projection, aggregation, group-by).

2. **Performance Optimizations:**
    - Efficient memory management and faster execution of queries to process ~20 million records.
3. **Resilience & Edge Cases:**
    - Handling outliar values.
    - Ensuring that the file reader is built from scratch without using external libraries for parsing or reading JSON files.
4. **Execution Time:**
    - The time taken to process each query will be measured by the `run.sh` script and will factor into your evaluation.

Submission Instructions

1. Create a query engine that implements the four specified queries
2. Your solution must include a custom file reader implementation for the JSON data
3. Modify the provided `run.sh` script to execute your program with the appropriate command
4. The `run.sh` script will be used to evaluate your solution during judging

How to Use the `run.sh` Script

We will provide a `run.sh` script template that you need to modify. This script will:

1. Accept a query name as input (`query1` , `query2` , `query3` , `query4`)
2. Execute your program to run that specific query
3. Measure execution time
4. Output the query results

Simply replace the placeholder section in the script with the command that runs your program. For example:

- Python: `python3 your_solution.py "$1"`
- Java: `java -jar YourSolution.jar "$1"`
- Node.js: `node your_solution.js "$1"`
- Go: `./your_solution "$1"`
- etc.

Where `$1` represents the query name passed to the script.

You can then run any of the predefined queries:

```
./run.sh query1  # For the record count query
./run.sh query2  # For the payment type aggregation
./run.sh query3  # For the vendor analysis with date filter
./run.sh query4  # For the daily analysis for January 2024
```

Conclusion

This challenge focuses on your ability to build an efficient, custom query engine that can process large datasets with optimal performance. The emphasis is on implementing a custom JSON reader and efficient data processing techniques. By completing this challenge, you'll demonstrate your skills in parallelization, memory management, and algorithm design for handling real-world data processing requirements.

Good luck, and we look forward to seeing your innovative solutions at the hackathon!