Laboratory Report: OLAP Operations in HQL

Course: ISIT312/ISIT912 Big Data Management
Semester: Spring 2023
Laboratory: Lab 5 - OLAP Operations in Hive HQL

Executive Summary

This laboratory exercise explored advanced OLAP (Online Analytical Processing) operations in Hive HQL, focusing on data warehousing extensions including GROUP BY enhancements (ROLLUP, CUBE, GROUPING SETS), window functions, and analytical functions. The exercises demonstrated efficient multi-dimensional data analysis techniques on a sample ORDERS dataset representing a three-dimensional data cube.

---

1. Dataset Overview

1.1 Table Structure

The ORDERS table was created with the following schema:

```sql
CREATE TABLE ORDERS(
   part     CHAR(7),
   customer VARCHAR(30),
   amount   DECIMAL(8,2),
   oyear    DECIMAL(4),
   omonth   DECIMAL(2),
   oday     DECIMAL(2)
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
  STORED AS TEXTFILE;
```

1.2 Data Characteristics

- **Fact Measure:** amount (order amount)
- **Dimensions:**
  - part (product dimension)
  - customer (customer dimension)
  - time (oyear, omonth, oday - hierarchical time dimension)
- **Total Records:** 10 rows
- **Total Amount:** 1143.00

- **Parts:** bolt, screw, nut, washer
- **Customers:** James, Peter, Bob, Alice
- **Time Range:** 2016-2018

---

2. Basic Aggregations with GROUP BY

2.1 Simple Aggregations

Query 1: Count orders per part

```sql
SELECT part, COUNT(*)
FROM orders
GROUP BY part;
```

Results:

- bolt: 5 orders
- nut: 1 order
- screw: 2 orders
- washer: 2 orders

Query 2: Sum amounts per part

```sql
SELECT part, SUM(amount)
FROM orders
GROUP BY part;
```

Results:

- bolt: 900.00
- nut: 23.00
- screw: 75.00
- washer: 145.00

2.2 Multi-Dimensional Aggregation Challenge

The inefficient UNION-based approach demonstrated the problem of multiple table scans:

```sql
SELECT part, NULL, COUNT(*)
FROM orders GROUP BY part
UNION
SELECT NULL, customer, COUNT(*)
FROM orders GROUP BY customer
UNION
SELECT part, customer, COUNT(*)
FROM orders GROUP BY part, customer;
```

Execution Time: 103.33 seconds

Performance Issue: Table scanned 3 times

Result: 16 rows with aggregations at different dimensional levels

---

3. ROLLUP Operator

3.1 Hierarchical Aggregation

Query: Part and Customer with ROLLUP

```sql
SELECT part, customer, SUM(amount)
FROM orders
GROUP BY part, customer WITH ROLLUP;
```

Execution Time: 18.664 seconds (83% faster than UNION approach)

Key Results:

- Grand Total: 1143.00 (NULL, NULL)
- Per Part Subtotals: bolt (900.00), nut (23.00), screw (75.00), washer (145.00)
- Per Part-Customer Details: 9 detailed combinations

Advantage: Single table scan performs hierarchical aggregation efficiently.

3.2 Time Dimension ROLLUP

Query: Year and Month aggregation

```sql
```

```
SELECT oyear, omonth, SUM(amount)
FROM orders
GROUP BY oyear, omonth WITH ROLLUP;
```

Results Structure:

- Grand total across all years: 1143.00
- Yearly subtotals: 2016 (345.00), 2017 (120.00), 2018 (678.00)
- Monthly details within each year

---

4. CUBE Operator

4.1 All Dimensional Combinations

Query: Average amount with CUBE

```
sql

SELECT part, customer, AVG(amount)
FROM orders
GROUP BY part, customer WITH CUBE;
```

Execution Time: 17.994 seconds
Total Rows: 17 (all possible combinations)

Key Results:

- Overall average: 114.30
- Customer averages: Alice (39.00), Bob (300.00), James (116.25), Peter (100.00)
- Part averages: bolt (180.00), nut (23.00), screw (37.50), washer (72.50)
- Part-Customer combinations: 9 detailed averages

Verification:

- Overall average confirmed: 114.30
- Bolt average confirmed: 180.00

---

5. GROUPING SETS Operator

5.1 Custom Aggregation Combinations

Query: Specific grouping combinations

```sql
SELECT part, customer, oyear, COUNT(*)
FROM ORDERS
GROUP BY part, customer, oyear
GROUPING SETS ((part), (customer), (oyear, customer));
```

Execution Time: 17.318 seconds

Results: 16 rows with three specific grouping levels

Grouping Levels Produced:

1. By part only: 4 rows
2. By customer only: 4 rows
3. By year and customer combination: 8 rows

5.2 Equivalence with ROLLUP

Query demonstrating equivalence:

```sql
SELECT oyear, omonth, oday, SUM(amount)
FROM orders
GROUP BY oyear, omonth, oday
GROUPING SETS((oyear,omonth,oday), (oyear,omonth), (oyear), ());
```

This produced identical results to the ROLLUP version:

- 21 total rows
- Hierarchical aggregation from daily to yearly to grand total

---

6. Window Functions

6.1 Basic Windowing

Query: Sum with partitioning

```sql
```

```
SELECT part, SUM(amount) OVER (PARTITION BY part)
FROM orders;
```

Results:

- Each row shows its part and the total for that part
- bolt rows: all show 900.00
- nut rows: all show 23.00
- screw rows: all show 75.00
- washer rows: all show 145.00

With DISTINCT:

```
sql

SELECT DISTINCT part, SUM(amount) OVER (PARTITION BY part)
FROM orders;
```

Produces same result as GROUP BY but using window functions.

6.2 Multiple Aggregations in Windows

Query: MAX and SUM together

```
sql

SELECT part, customer, amount,
    MAX(amount) OVER (PARTITION BY part),
    SUM(amount) OVER (PARTITION BY part)
FROM orders;
```

Key Insight: Each row displays individual values alongside partition-level aggregates:

- bolt rows show MAX=300.00, SUM=900.00
- screw rows show MAX=55.00, SUM=75.00

---

7. Window Ordering

7.1 Running Totals

Query: Cumulative sum with ordering

```
sql
```

```
SELECT part, amount,
    SUM(amount) OVER (PARTITION BY part ORDER BY amount)
FROM orders;
```

Results for bolt:

- 100.00 → 200.00 (cumulative sum of two 100.00 values)
- 200.00 → 600.00 (adds two 200.00 values)
- 300.00 → 900.00 (final cumulative)

Issue: Rows with same amount are aggregated together.

7.2 Refined Ordering

Query: Row-by-row cumulative sum

```
sql

SELECT part, amount,
    SUM(amount) OVER (PARTITION BY part
                ORDER BY amount, oyear, omonth, oday)
FROM orders;
```

Results for bolt:

- 100.00 → 100.00
- 100.00 → 200.00
- 200.00 → 400.00
- 200.00 → 600.00
- 300.00 → 900.00

Improvement: More selective ordering key ensures row-by-row processing.

7.3 Walking Average

Query: Time-based walking average

```
sql

SELECT part, amount, oyear,
    AVG(amount) OVER (PARTITION BY part ORDER BY oyear)
FROM orders;
```

Results for bolt:

- 2016: 200.00 (AVG of one 200.00 order)
- 2017: 150.00 (AVG of 200 and 100)
- 2018: 180.00 (AVG of all five orders: 900/5)

---

8. Window Framing

8.1 Expanding Frame (Default)

Query: Default frame behavior

```sql
SELECT part, amount,
    AVG(amount) OVER (PARTITION BY part
                ORDER BY oyear, omonth, oday)
FROM orders;
```

Frame Behavior: Expands from first row to current row (unbounded preceding to current row).

8.2 Fixed-Size Frame

Query: Moving average with 1 preceding row

```sql
SELECT part, amount,
    AVG(amount) OVER (PARTITION BY part
                ORDER BY oyear, omonth, oday
                ROWS 1 PRECEDING)
FROM orders;
```

Results for bolt:

- Row 1: 200.00 (only current row)
- Row 2: 150.00 (AVG of 200 and 100)
- Row 3: 100.00 (AVG of 100 and 100)
- Row 4: 150.00 (AVG of 100 and 200)
- Row 5: 250.00 (AVG of 200 and 300)

Advantage: Creates a sliding window of fixed size for moving averages.

8.3 Frame Specification Options

Available frame specifications:

1. `ROWS BETWEEN 3 PRECEDING AND CURRENT ROW`
2. `ROWS BETWEEN UNBOUNDED PRECEDING AND 2 FOLLOWING`
3. `ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`
4. `ROWS BETWEEN 2 FOLLOWING AND UNBOUNDED FOLLOWING`

---

9. LEAD and LAG Functions

9.1 LEAD Function

Query: Access next row value

```sql
SELECT part, amount,
    LEAD(amount) OVER (PARTITION BY part
                ORDER BY oyear, omonth, oday)
FROM orders;
```

Results for bolt:

- 200.00 → 100.00 (next value)
- 100.00 → 100.00
- 100.00 → 200.00
- 200.00 → 300.00
- 300.00 → NULL (no next value)

9.2 LAG Function

Query: Access previous row value

```sql
SELECT part, amount,
    LAG(amount) OVER (PARTITION BY part
                ORDER BY oyear, omonth, oday)
FROM orders;
```

Results for bolt:

- 200.00 → NULL (no previous value)
- 100.00 → 200.00
- 100.00 → 100.00
- 200.00 → 100.00
- 300.00 → 200.00

9.3 Calculations with LAG

Query: Change from previous value

```sql
SELECT part, amount,
    amount - LAG(amount) OVER (PARTITION BY part
                               ORDER BY oyear, omonth, oday)
FROM orders;
```

Results for bolt:

- 200.00 → NULL
- 100.00 → -100.00 (decrease)
- 100.00 → 0.00 (no change)
- 200.00 → 100.00 (increase)
- 300.00 → 100.00 (increase)

9.4 Eliminating NULLs

Query: Using default value in LAG

```sql
SELECT part, amount,
    amount + LAG(amount, 1, 0) OVER (PARTITION BY part
                                     ORDER BY oyear, omonth, oday)
FROM orders;
```

Result: NULL values replaced with 0, enabling clean arithmetic operations.

---

10. Analytic Functions

10.1 RANK Function

Query:

```sql
SELECT part, amount,
    RANK() OVER (PARTITION BY part ORDER BY amount)
FROM orders;
```

Results for bolt:

- 100.00 → 1 (two rows, both rank 1)
- 100.00 → 1
- 200.00 → 3 (rank jumps by number of previous ties)
- 200.00 → 3
- 300.00 → 5

Behavior: Ranks with gaps when ties exist.

10.2 DENSE_RANK Function

Query:

```sql
SELECT part, amount,
    DENSE_RANK() OVER (PARTITION BY part ORDER BY amount)
FROM orders;
```

Results for bolt:

- 100.00 → 1
- 100.00 → 1
- 200.00 → 2 (no gap)
- 200.00 → 2
- 300.00 → 3

Behavior: Ranks without gaps, consecutive numbering.

10.3 CUME_DIST Function

Query:

```sql
SELECT part, amount,
    CUME_DIST() OVER (PARTITION BY part ORDER BY amount)
FROM orders;
```

Results for bolt:

- 100.00 → 0.4 (2 rows / 5 total)
- 100.00 → 0.4
- 200.00 → 0.8 (4 rows / 5 total)
- 200.00 → 0.8
- 300.00 → 1.0 (5 rows / 5 total)

Interpretation: Cumulative distribution - percentage of values ≤ current value.

10.4 PERCENT_RANK Function

Query:

```sql
SELECT part, amount,
    PERCENT_RANK() OVER (PARTITION BY part ORDER BY amount)
FROM orders;
```

Results for bolt:

- 100.00 → 0.0 (calculated as (1-1)/(5-1) = 0)
- 200.00 → 0.5 (calculated as (3-1)/(5-1) = 0.5)
- 300.00 → 1.0 (calculated as (5-1)/(5-1) = 1.0)

Formula: (rank - 1) / (total rows - 1)

10.5 NTILE Function

Query: Divide into 2 buckets

```sql
```

```
SELECT part, amount,
    NTILE(2) OVER (PARTITION BY part ORDER BY amount)
FROM orders;
```

Results for bolt (5 rows divided into 2 buckets):

- 100.00 → 1 (bucket 1)
- 100.00 → 1
- 200.00 → 1
- 200.00 → 2 (bucket 2)
- 300.00 → 2

Query: Divide into 5 buckets

sql

```
SELECT part, amount,
    NTILE(5) OVER (PARTITION BY part ORDER BY amount)
FROM orders;
```

Results for bolt: Each row gets its own bucket (1, 2, 3, 4, 5)

Use Case: Quartile/percentile analysis for data distribution.

10.6 ROW_NUMBER Function

Query:

sql

```
SELECT part, amount,
    ROW_NUMBER() OVER (PARTITION BY part ORDER BY amount)
FROM orders;
```

Results for bolt:

- 100.00 → 1
- 100.00 → 2 (unique sequential number despite tie)
- 200.00 → 3
- 200.00 → 4
- 300.00 → 5

Behavior: Assigns unique sequential numbers regardless of ties.

---

11. Performance Analysis

11.1 Execution Time Comparison

Operation	Approach	Execution Time	Table Scans
Multi-dimension aggregation	UNION	103.33s	3
Multi-dimension aggregation	ROLLUP	18.66s	1
Average calculation	CUBE	17.99s	1
Custom groupings	GROUPING SETS	17.32s	1
Window aggregation	OVER clause	~17-19s	1

Key Finding: Advanced GROUP BY operators (ROLLUP, CUBE, GROUPING SETS) provide 80-85% performance improvement over UNION-based approaches by eliminating redundant table scans.

11.2 Efficiency Insights

1. **Single Scan Advantage:** All advanced operators compute multiple aggregation levels in one pass
2. **Window Functions:** Enable row-level context with partition-level aggregates without GROUP BY
3. **Analytic Functions:** Provide ranking and distribution calculations efficiently within partitions

---

12. Key Learnings

12.1 Technical Concepts

1. **ROLLUP:** Creates hierarchical aggregations following dimension order
2. **CUBE:** Generates all possible dimensional combinations (power set)
3. **GROUPING SETS:** Allows explicit specification of desired grouping combinations
4. **Window Functions:** Enable calculations across row sets related to current row
5. **Frame Specification:** Controls which rows are included in window calculations
6. **Analytic Functions:** Provide ranking, distribution, and position-based calculations

12.2 Best Practices

1. Use ROLLUP for hierarchical dimensions (e.g., year → month → day)
2. Use CUBE when all dimensional combinations are needed
3. Use GROUPING SETS for specific, non-hierarchical grouping combinations

4. Add specific ORDER BY keys to window functions to avoid unintended tie-handling
5. Choose appropriate frame specifications based on analysis requirements
6. Select RANK vs DENSE_RANK vs ROW_NUMBER based on tie-handling needs

12.3 Practical Applications

1. **Time-series Analysis:** Walking averages, cumulative sums, period-over-period changes
2. **Ranking Systems:** Top-N queries, quartile analysis, percentile rankings
3. **Multi-dimensional Reporting:** Subtotals, grand totals, drill-down capabilities
4. **Comparative Analysis:** Current vs previous/next values, change detection
5. **Distribution Analysis:** Bucket assignments, cumulative distributions

---

13. Conclusion

This laboratory successfully demonstrated the power and efficiency of Hive HQL's OLAP extensions. The exercises showed how advanced GROUP BY operators (ROLLUP, CUBE, GROUPING SETS) dramatically improve query performance over traditional UNION approaches, while window and analytic functions provide sophisticated row-level calculations within partitioned data sets.

The hands-on experience with the ORDERS dataset illustrated real-world data warehousing scenarios, including hierarchical time dimensions, multi-dimensional analysis, and advanced analytics. These techniques are essential for building efficient big data analytics solutions on Hadoop/Hive platforms.

Key Achievement: Reduced query execution time by up to 83% while maintaining or enhancing analytical capabilities through proper use of OLAP operators.

Appendix: Window Clause Syntax

Named Window Definition:

```sql
SELECT part, SUM(amount) OVER w
FROM orders
WINDOW w AS (PARTITION BY part);
```

Advantage: Improves readability and reusability when multiple window functions share the same specification.