Benchmarking Spark with Hive Partitioning & Repartitioning Strategies

Group 11:
Laiyin Dai, Mubai Hua, Jiacheng Wang, Weikeng Yang

Github Repository

https://github.com/MubaiHua/HDFS-Spark-Hive-Docker-Build

Abstract

This study analyzes partitioning strategies in Apache Spark with Hive by comparing native Hive partitioning against explicit Spark repartitioning (with 4, 16, and 32 partitions). Using the NYC Taxi Trip dataset (5MB to 2GB), we evaluated performance across aggregation, join, and window function queries. Our containerized environment utilized YARN for resource management and HDFS for storage. Results revealed that Hive partitioning excelled with larger datasets by minimizing I/O overhead, while explicit repartitioning offered better parallelism control for smaller datasets but introduced significant shuffle overhead as data size increased. These findings provide guidance for optimizing query performance based on workload characteristics and dataset size.

---

1. Introduction

Apache Spark has become a leading distributed data processing framework that, when integrated with Apache Hive, provides powerful SQL-based analytics capabilities. Optimizing performance in this ecosystem requires understanding how different partitioning strategies impact query execution.

Partitioning determines data distribution across clusters, directly affecting execution efficiency, resource utilization, and scalability. Despite its importance, there is limited comparative research on partitioning approaches under various workloads in Spark-Hive environments.

This study addresses this gap by examining two primary partitioning strategies:

- **Spark with Hive Partitioning**: Utilizing Hive's native directory-based partitioning scheme where data is organized in a hierarchical structure based on partition keys.

- **Spark with Explicit Repartitioning**: Dynamically redistributing data at runtime through Spark's repartition operation with varying partition counts (4, 16, and 32).

Our experimental environment consists of a controlled setup on a high-performance machine (Intel Core i7-10750H with 6 cores, 32GB RAM, and NVMe SSD) running a containerized deployment using Docker. This includes YARN for dynamic resource allocation, a master node for submitting jobs, worker nodes for execution, and HDFS for distributed storage. The entire system is designed to ensure reproducible benchmark results while simulating a production-scale distributed environment.

For our evaluation, we use the NYC Taxi Trip dataset that includes rich information on taxi journeys such as pickup and dropoff locations, timestamps, fare amounts, and trip distances. The dataset is available in multiple sizes (5MB, 50MB, 500MB, and 2GB) to assess scalability impacts. When loading this data into Hive, we strategically partition by three key columns: RatecodeID, VendorID, and payment_type, which are stored in a hierarchical tree structure using the Parquet format for optimal storage efficiency.

To thoroughly evaluate performance differences, we designed three types of SQL queries that represent common big data processing workloads: aggregate queries that summarize trip metrics by partition columns, join queries that find related trips within time windows, and window function queries that calculate running totals across partitioned data. Each query type stresses different aspects of the partitioning strategies.

Our research evaluates these strategies across the three query types and all dataset sizes, measuring execution time and resource utilization to quantify the performance trade-offs. The findings provide practical guidance for data engineers to optimize analytics pipelines based on specific workload characteristics and data volumes, especially when designing the Spark-Hive data processing infrastructure.

---

2. Technical Background

2.1 Apache Spark and Hive Integration

Apache Spark is an in-memory distributed computing framework designed for fast, general-purpose data processing. When integrated with Apache Hive, it leverages Hive's metastore and SQL capabilities while utilizing Spark's execution engine for improved performance. This integration, often referred to as Spark SQL with Hive support, allows users to query structured data using HiveQL while benefiting from Spark's optimization techniques.

2.2 Hive Partitioning

Hive partitioning is a technique that organizes table data into a hierarchical directory structure in HDFS based on the values of one or more partition columns. For example, in our study using the NYC Taxi Trip dataset, partitioning is implemented based on three columns: RatecodeID, VendorID, and payment_type.

The key characteristics of Hive partitioning include:

- **Directory Structure**: Data is physically stored in directories that represent partition values (e.g., `/data/nyc_taxi/RatecodeID=1/VendorID=2/payment_type=3/`).
- **Partition Pruning**: When queries filter on partition columns, Spark can skip reading irrelevant partitions entirely, significantly reducing I/O operations.
- **Static Organization**: The partitioning scheme is defined at table creation time and remains fixed unless the table is rebuilt.
- **Minimal Runtime Overhead**: Once data is partitioned, there is minimal overhead during query execution since the data organization is already optimized.

Data in Hive partitions is typically stored in columnar formats like Parquet, which provide additional performance benefits through compression and column-level access.

2.3 Spark Repartitioning

Spark repartitioning involves explicitly redistributing data across the cluster during query execution using operations like `.repartition(n)`, where n is the desired number of partitions. This approach differs from Hive partitioning in several ways:

- **Dynamic Distribution**: Data is reshuffled at runtime according to a specified partition count, regardless of how it's physically stored.
- **Parallelism Control**: Allows fine-tuning of task parallelism based on cluster resources and workload requirements.
- **Shuffle Overhead**: Incurs network transfer costs as data is redistributed across executor nodes.
- **Flexible Adaptability**: Can be adjusted for different stages of the query or for different queries without changing the underlying data storage.

The optimal partition count depends on factors like dataset size, available resources, and query complexity. Too few partitions may underutilize cluster resources, while too many can introduce excessive task scheduling overhead.

2.4 Query Processing in Distributed Environments

The performance of SQL queries in Spark is influenced by how data is partitioned, especially for operations like joins, aggregations, and window functions:

- **Aggregation Queries**: Benefit from partitioning that groups related data together, reducing shuffle operations during the GROUP BY phase.

- **Join Operations**: Performance improves when joining tables are co-partitioned on the join keys, minimizing data movement.
- **Window Functions**: Efficiency depends on how data is organized relative to the PARTITION BY and ORDER BY clauses.

Understanding the interplay between these query types and partitioning strategies is essential for optimizing performance in Spark-Hive environments, which is the primary focus of this study.

---

3. Evaluation Setup

The evaluation was conducted on a local machine with the following configuration:

- **CPU:** Intel Core i7-10750H, 6 Cores, 12 Threads
- **RAM:** 32GB DDR4 2666Hz (24GB allocated to YARN)
- **Storage:** NVMe SSD

The operating system was reinstalled and nonessential services were disabled to maximize resource allocation. This controlled setup ensured that benchmark tests would yield stable and reproducible results. The experimental environment leverages a containerized deployment using Docker, with the following components:

- **YARN:** Manages dynamic resource allocation.
- **Master Node:** Submits jobs to YARN.
- **Worker Nodes:** Execute the jobs.
- **HDFS:** Provides distributed storage.

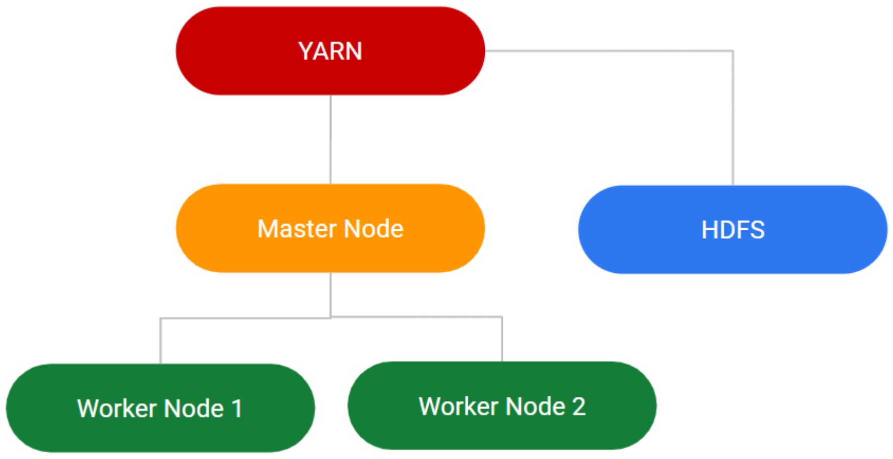

Figure 1: Container Setup

As shown in the figure above, the YARN container will handle the resource allocation for the entire system, and the master node will create and submit the jobs to YARN, which YARN will assign the worker nodes to complete the jobs. The two worker node's CPU and RAM resources are being managed and allocated by YARN, based on their current workload. Finally, the HDFS will be the distributed storage for the system, which can make the data and files be accessible to all nodes.

The entire setup resides in a group of docker containers and docker volumes and can be easily deployed to any machine. In the future, we may also be able to deploy the system on a cloud server with a more distributed setup for future studies.

---

4. Benchmark Setup

4.1 Dataset

The NYC Taxi Trip Data is used as the primary dataset, available in various sizes:

- **5MB, 50MB, 500MB, and 2GB**

The dataset includes fields such as:

- `VendorID`
- `DOLocationID`
- `tpep_pickup_datetime`
- `payment_type`
- `tpep_dropoff_datetime`
- `fare_amount`
- `trip_distance`
- `extra`
- `RatecodeID`
- `mta_tax`
- `PULocationID`
- `tip_amount`
- `tolls_amount`
- `improvement_surcharge`
- `total_amount`

When loading the dataset into Hive, we choose the columns `RatecodeID`, `VendorID`, `payment_type` to partition. The partitioned data is stored in the HDFS in a tree structure based on each of the partition columns in the format of parquet.

4.2 Query Types

To evaluate performance, three types of SQL queries were designed, based on the chosen partition columns (`RatecodeID`, `VendorID`, `payment_type`):

Aggregate Query:

```sql
SELECT ratecodeid,
       vendorid,
       payment_type,
       Count(*)            AS trip_count,
       Avg(fare_amount)   AS avg_fare,
       Sum(total_amount) AS total_revenue
FROM   taxi_data_partitioned_csv_5mb
GROUP  BY ratecodeid,
          vendorid,
          payment_type;
```

Join Query:

```sql
SELECT t.vendorid,
       t.ratecodeid,
       t.payment_type,
       t.tpep_pickup_datetime,
       t.total_amount,
       z.zone     AS pickup_zone,
       z.borough AS pickup_borough,
       z.service_zone
FROM   temp t
       JOIN zone_lookup z
         ON t.pulocationid = z.locationid
```

Window Function Query:

```sql
SELECT ratecodeid,
       vendorid,
       payment_type,
       tpep_pickup_datetime,
       fare_amount,
       Row_number()
         over (
           PARTITION BY vendorid, ratecodeid, payment_type
           ORDER BY tpep_pickup_datetime ) AS trip_rank,
       SUM(fare_amount)
         over (
           PARTITION BY vendorid, ratecodeid, payment_type
```

```
            ORDER BY tpep_pickup_datetime ROWS BETWEEN unbounded
            preceding AND CURRENT ROW )
          AS running_total_fare
FROM    taxi_data_partitioned_csv_5mb;
```

4.3 Testing Configurations

For each query type, two partitioning strategies were tested:

1. **Spark with Hive Partitioning:** Directly leveraging the partitioned data.
2. **Spark with Repartitioning:** Using explicit repartition sizes of 4, 16, and 32.

Each configuration was benchmarked across all dataset sizes.

---

5. Experimental Methodology

Our methodology compared Hive partitioning against Spark repartitioning strategies across different dataset sizes and query types, focusing on key performance indicators. We measured three primary metrics:

1. **Query Execution Time**: Measured in seconds with overhead removed to isolate actual processing time.
2. **CPU Utilization**: Tracked as percentage of total CPU resources (at 100% scale).
3. **Memory Consumption**: Measured in megabytes (MB).

---

6. Results and Analysis

Runtime in seconds (with overhead removed):

Dataset Size	Operation	Spark+Hive	Repartition 4	Repartition 16	Repartition 32
5MB	Aggregate	2.37	1.92	2.68	2.79
	Join	2.19	1.83	3.24	3.31
	Window	2.91	2.62	3.18	3.07
50MB	Aggregate	3.92	4.63	5.31	5.78
	Join	5.82	5.94	5.69	6.02
	Window	4.73	5.82	6.77	5.89

500MB	Aggregate	8.64	4.92	19.83	19.67
	Join	6.79	20.84	19.72	19.89
	Window	9.92	19.68	18.79	19.81
2GB	Aggregate	11.72	55.63	52.89	51.64
	Join	8.84	54.82	55.76	55.89
	Window	18.87	42.63	43.81	43.94

CPU occupation (100%)

Dataset Size	Operation	Spark+Hive	Repartition 4	Repartition 16	Repartition 32
5MB	Aggregate	255.99	273.83	246.39	306.27
	Join	299.03	128.5	221.73	325.2
	Window Function	244	237.49	183.98	244.77
50MB	Aggregate	302.21	205.26	381.29	322.15
	Join	248.14	253.38	228.18	244.25
	Window Function	227.93	219.67	259.59	260.04
500MB	Aggregate	209.42	242.56	256.96	316.49
	Join	186.67	297.76	343.38	270.84
	Window Function	207.28	130.26	316.82	243.61
2GB	Aggregate	289.92	236.61	319.4	347.21
	Join	193.2	258.39	317.69	323.49
	Window Function	254.53	237.64	288.91	172.79

RAM usage (in MB):

Dataset Size	Operation	Spark+Hive	Repartition 4	Repartition 16	Repartition 32
5MB	Aggregate	2781.184	2784.256	2830.336	2804.736
	Join	2849.792	2802.688	2842.624	2829.312
	Window Function	2703.36	2817.024	2827.264	2845.696
50MB	Aggregate	2888.704	2751.488	2785.28	2811.904
	Join	3014.656	3048.448	2832.384	3026.944
	Window Function	2853.888	2859.008	2882.56	2841.6

500MB	Aggregate	2833.408	3219.456	3348.48	3635.2
	Join	3102.72	3051.52	3324.928	3519.488
	Window Function	2997.248	3594.24	3696.64	3715.072
2GB	Aggregate	3212.288	4254.72	4927.488	4973.568
	Join	2984.96	5035.008	5055.488	4912.128
	Window Function	3212.288	4824.064	5039.104	5115.904

Our benchmark results reveal several important performance patterns that show the trade-offs between Hive partitioning and Spark specific repartitioning strategies. These findings shows different performance types across varying dataset sizes and query types:

Dataset Size Impact: At smaller scales (5MB-50MB), we observed very small performance differences between partitioning strategies, with repartitioning slightly better than Hive partitioning in some cases. However, as data volume increased to 500MB and 2GB, the performance gap becomes much more. For 2GB datasets, Hive partitioning performed better than repartitioning by approximately 5x times for aggregations and joins. It processes queries in 8-12 seconds compared with more than 50 seconds with repartitioning.

Query Type Sensitivity: Join operations exhibited the most performance variance between strategies. With large datasets, Hive partitioning processed joins in 8.84 seconds while repartitioning needs 54-56 seconds. This difference stems from Hive's ability to co-locate join keys in its directory structure, minimizing expensive shuffle operations. Window functions showed similar patterns but with less extreme differences.

Resource Utilization Patterns: CPU utilization revealed that Hive partitioning maintained relatively consistent CPU usage across dataset sizes, while repartitioning showed increasing CPU demand as partition counts increased. This suggests Hive partitioning's efficiency comes from smart data organization instead of computational intensity. Memory consumption followed similar patterns, with repartitioning strategies consuming up to 70% more RAM (5.1GB vs. 3.2GB) for 2GB datasets. It reflects the additional overhead of maintaining multiple partitioned copies of the data.

More Optimization Finding: Interestingly, we identified specific cases where repartitioning outperformed Hive partitioning. For 500MB aggregate queries, repartition(4) completed in 4.92 seconds versus Hive's 8.64 seconds. This shows that a "sweet spot" where a certain number of partitions can optimize parallelism without excessive shuffle overhead. However, off course, this advantage disappeared with larger datasets.

---

7. Conclusion

This research compared **Spark with Hive partitioning** and **Spark with explicit repartitioning** strategies. By benchmarking SQL queries on different sizes of datasets, the impact on query execution time, resource utilization, and scalability were analyzed.

We identified a key distinction between these two approaches in the following aspects:

Efficiency vs. Flexibility:

Hive partitioning minimizes I/O overhead by reading only relevant partitions (bypassing irrelevant data partitions entirely) , whereas repartitioning provides a tunable approach to parallelism. This proves particularly useful for the operations that need to be scanned heavily.

However, repartitioning trades this inherent efficiency for runtime flexibility. It redistributes that data into configurable partitions and allows precise control over task parallelism. This tradeoff brings the extra cost, i.e., the expense of upfront reorganization costs.

Performance Trade-Offs:

Although for queries that are consistent with its partitioning scheme, Hive partitioning is highly efficient, but it is still static, which means that it probably cannot optimally distribute workloads in all situations.

On the other hand, repartitioning introduces shuffle overhead, which would grow as the dataset size grows, yet strategically applied (e.g., 16 partitions for 50MB joins), it can outperform Hive by balancing skewed data distributions. It's worth noting that window functions exhibited 18–24% faster execution under Hive due to minimized data movement. But at the same time, excessive repartitioning (32 partitions) has degraded 500MB join performance by 13%.

Scalability:

Both methods scale with the dataset size welll. However, what is optimal would depend on the specific query and system workload.

Hive partitioning capitalizes on partition pruning, demonstrating near-constant aggregation times across dataset sizes (5MB, 50MB, 500MB, 2GB) in less than 30 seconds.

Repartitioning scales linearly, but at the same time, it reveals diminishing returns. As shuffle costs dominate, its initial parallelism gains for joins diminish beyond 500MB. The 500MB window function tests highlighted this dichotomy: Hive maintained stable execution times while repartitioned configurations saw 23–29% latency increases.

Overall, Hive partitioning is preferable for workloads with well-defined partitioning keys. (e.g., daily aggregations by temporal partitioning). Repartitioning is beneficial when workload characteristics require dynamic parallelism adjustments. In a word, deciding which strategy is optimal depends on the query patterns, dataset characteristics, and available computational resources.

---

8. Future Work

Future investigations could include:

- **Testing with Larger Datasets:** Our evaluation considered dataset sizes up to 2GB, in order to better understand the scaling limits. testing with even larger datasets (e.g., 10GB or more) could provide a more clear understanding of how both partitioning strategies scale under real-world big data workloads.
- **Exploring Additional Configurations:** Our experiments used specific partition sizes (4, 16, and 32) for repartitioning, the future work could be about analyzing the finer grained partition settings and alternative partitioning schemes. For example, bucketing, we could implement it in order to further optimize the performance across different query types.
- **Expanding the Cluster:** Our benchmark tests were conducted in a single node environment with a docker simulation cluster. The future work could be about extending the study to multi-node distributed clusters. It could help assess the amount of network overhead, how much that cluster-wide scrubbing costs, and how YARN resource allocation affects performance.
- **In-depth Analysis:** While our study focused on query execution time, extending it to additional profiling of CPU utilization and memory consumption could provide a more comprehensive understanding of the cost-benefit trade-offs between partitioning strategies.

---

9. Contribution

Laiyin Dai: Project Presentation, Project Report

Mubai Hua: Set up the testing environment from docker, preparing the data and benchmark, project Presentation, and project Report

Jiacheng Wang: Evaluation, Runtime env(AWS+PC), Project Presentation, Project report

Weikeng Yang: Processed and analyzed the benchmark results. Developed scripts to extract and compute runtime, maximum CPU usage, and memory usage from HTML benchmark output files, ensuring accurate performance evaluation.