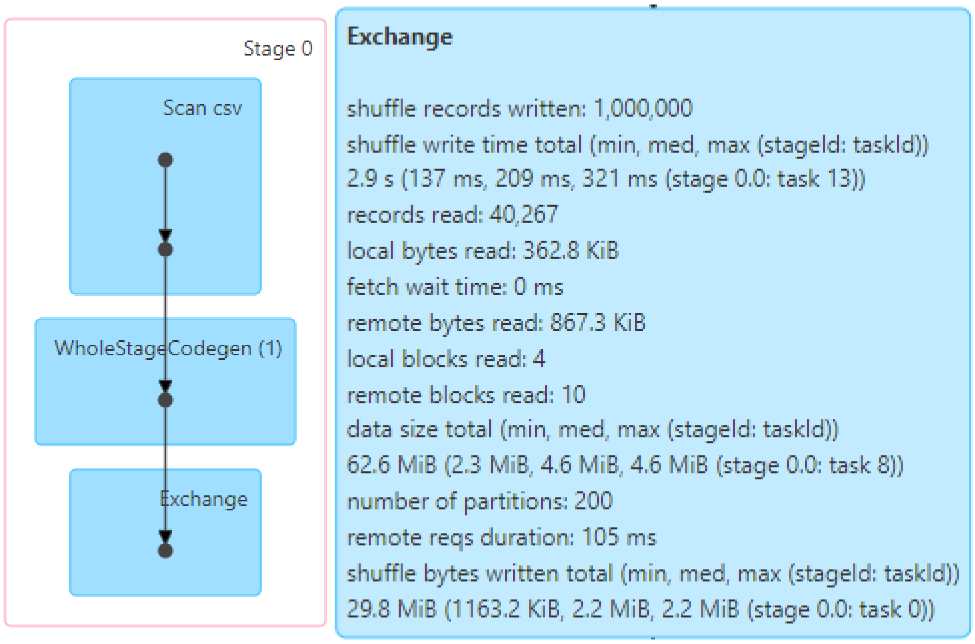
1. Calculate Daily Active Users (DAU) and Monthly Active Users (MAU) for the past year:
    a. Criteria: a user is considered active if they performed an action on a page, on a specific date;
    b. Solution that calculates DAU and MAU can be found in job: *spark_jobs/calculate_dau_mau.py;*
    c. While creating similar marketing metrics, we might face shuffling issues, due to transformations like *groupBy*, or *agg.* Depending on the column we group by, partition pruning might be inefficient as well; Filtering before transforming, may reduce the shuffling, as well as repartitioning on the groupBy column.
2. Calculate session-based metrics:
    a. User session duration is the elapsed time between the initiation of the first user action in a session and the completion of the last user action in the session. An action is considered as last in a session if the time difference between the end of the action and the beginning of the next one is greater than 30 minutes;
    b. Solution that calculates Average session duration and Actions per session can be found in job: *spark_jobs/calculate_session_metrics.py;*
3. Spark UI Analysis:
    a. The spark app CalculateSessionMetrics is split into 2 jobs:
        i. Read CSV: reads the CSV file (FileScanRDD), applies transformations, and prepares data for the next job. This job contains one stage that is split into 14 tasks (one for each date partition) with a duration of :

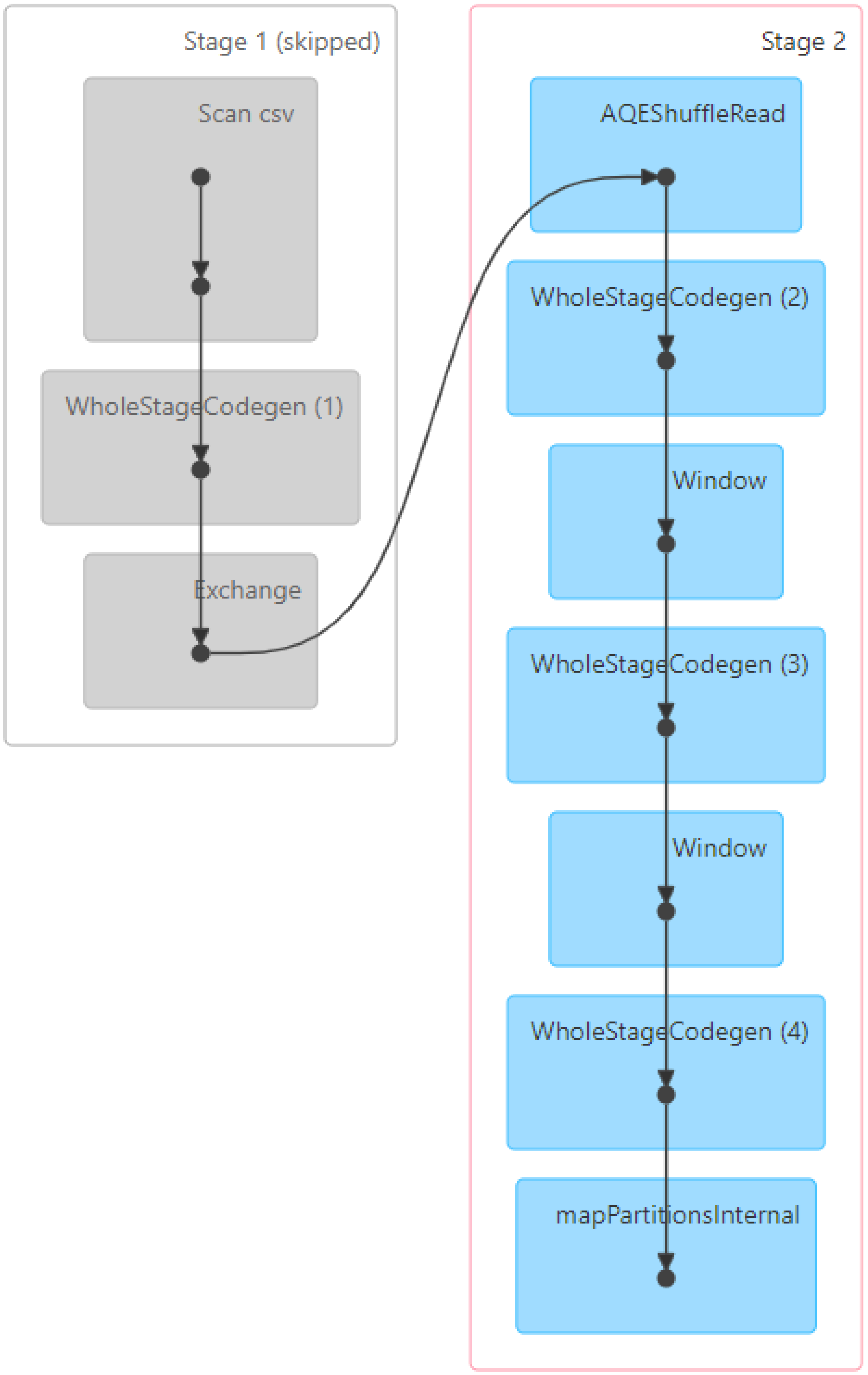


Exchange

```
shuffle records written: 1,000,000
shuffle write time total (min, med, max (stageId: taskId))
2.9 s (137 ms, 209 ms, 321 ms (stage 0.0: task 13))
records read: 40,267
local bytes read: 362.8 KiB
fetch wait time: 0 ms
remote bytes read: 867.3 KiB
local blocks read: 4
remote blocks read: 10
data size total (min, med, max (stageId: taskId))
62.6 MiB (2.3 MiB, 4.6 MiB, 4.6 MiB (stage 0.0: task 8))
number of partitions: 200
remote reqs duration: 105 ms
shuffle bytes written total (min, med, max (stageId: taskId))
29.8 MiB (1163.2 KiB, 2.2 MiB, 2.2 MiB (stage 0.0: task 0))
```

Details:
1. Scan CSV: File read operation from disk.
2. WholeStageCodegen: Optimized code generation for transformations.
3. Exchange: Spark is shuffle-exchanging data between executors in order to achieve optimal partitioning and it took around 2.9 seconds to write.
4. All 14 tasks had the same runtime (3s), meaning the data is equally-spread across partitions, reducing skewing;

ii. Calculate metrics: performs windowing operations, calculates time differences, assigns session IDs, and calculates metrics. This job contains one stage with one task with a duration of 2s



Details (can be observed in the Stages tab and SQL/DAtaFrame tab):

1. AQEShuffleRead: Adaptive Query Execution (AQE) optimization for coalescing shuffled data that was generated in the Exchange step previously (200 partitions), into partitions (25 partitions).
2. WholeStageCodegen: order by "timestamp" for the window functions and aggregations.

3. Window: Lag-based calculation of time_diff between consecutive events per user. Another window operation to calculate cumulative sum for session ID assignment (may be subject to optimization).
4. WholeStageCodegen: groupBy(“user_id) and calculate aggregations like sum, count and avg for session metrics.

Potential Bottlenecks:

- **Shuffling**: High shuffle overhead due to hash partitioning and sorting.
- **Window Functions**: Multiple window operations (lag and sum) can be resource-intensive.

b. Implement improvements:

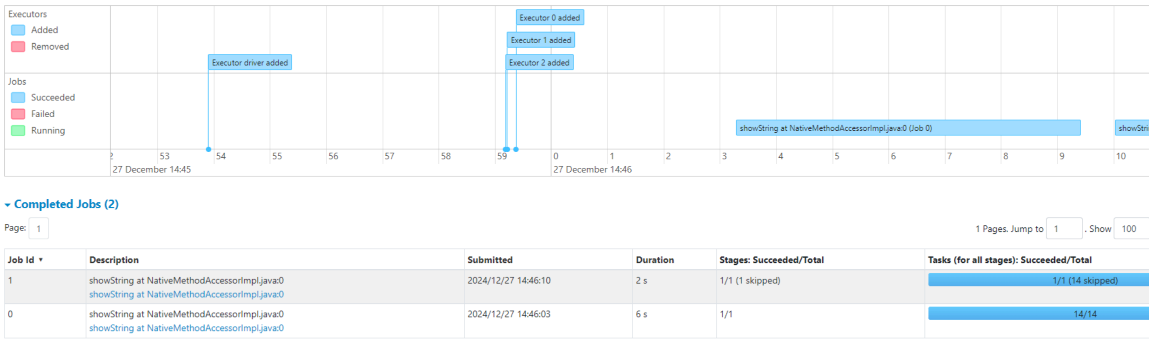
Optimized job: *spark_jobs/calculate_session_metrics_optimized.py.*

Improvements:

Converted data to parquet format (columnar) in order to improve read performance: achieved in a separate job called *spark_jobs/dataset_parquet.py* ;

Simplified sessionization logic with precomputed session-related features. Also, the precomputed data is cached after windowing.

Before:

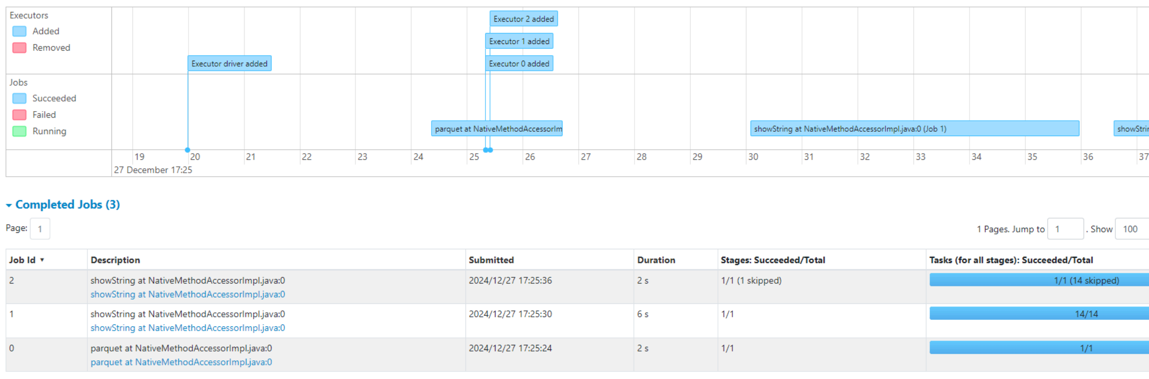


Completed Jobs (2)

Page: 1 1 Pages. Jump to 1 . Show 100

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/12/27 14:46:10	2 s	1/1 (1 skipped)	1/1 (14 skipped)
0	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/12/27 14:46:03	6 s	1/1	14/14

After:



Completed Jobs (3)

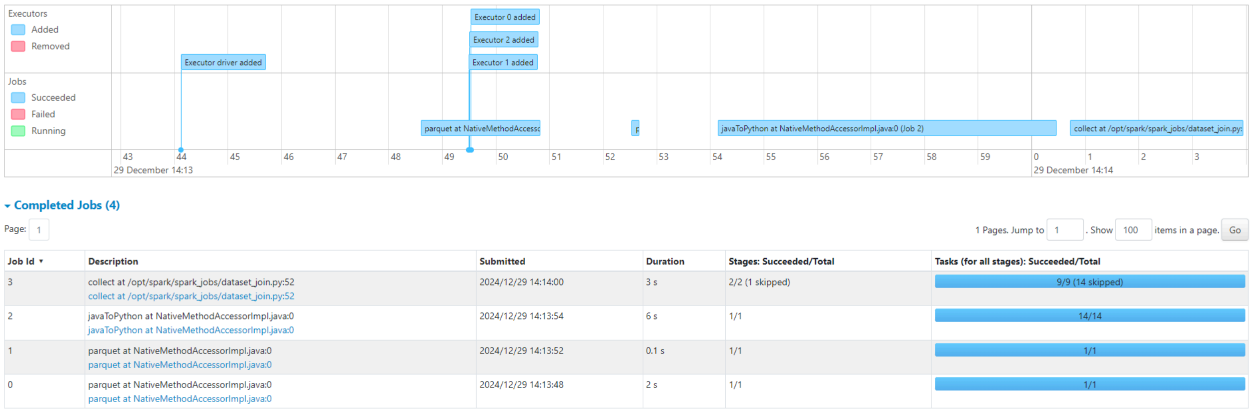
Page: 1 1 Pages. Jump to 1 . Show 100

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/12/27 17:25:36	2 s	1/1 (1 skipped)	1/1 (14 skipped)
1	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/12/27 17:25:30	6 s	1/1	14/14
0	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/12/27 17:25:24	2 s	1/1	1/1

Data reading from source takes less time now, using parquet format, than it did before with csv. Due to the nature of the transformations, there is now improvement in execution time for job 2.

c. Identified Bottlenecks:
    - **Shuffling**: High shuffle overhead due to hash partitioning and sorting. The high shuffle overhead is caused by the Exchange stage performing hash partitioning (hashpartitioning(user_id#0, 100)). This occurs because the query requires distributing data across partitions for operations like aggregations and window functions. Sorting operations (Sort [user_id#0 ASC, timestamp#1 ASC]) compound the shuffle overhead as they involve sorting within each partition.
      Proposed solution: analyze the dataset and adjust the partition count dynamically based on data size using Spark's spark.sql.shuffle.partitions configuration.
    - **Window Functions**: Multiple window operations (lag, sum) can be resource-intensive. The repeated sorting for each window definition ([user_id#0, timestamp#1 ASC]) exacerbates the issue.
      Proposed solution: combine multiple window operations into a single operation where possible.
    - **Aggregation Levels**: Nested aggregations increase execution time. The nested HashAggregate stages (13 to 16) introduce additional computation by processing partial aggregations at multiple levels. Aggregations on user_id#0 and session_id#42L might result in repetitive groupings.
      Proposed solution: consolidate aggregations wherever possible to avoid multiple aggregation stages.

4. Dataset Join(spark job *spark_jobs/dataset_join.py*):
    a. **Broadcast Join**: The User Metadata dataset is likely much smaller compared to the User Interactions dataset because metadata typically contains one row per user, while interactions may have multiple rows per user. Broadcasting the smaller dataset (User Metadata) avoids a costly shuffle operation on the larger dataset (User Interactions) by sending the smaller dataset to all executors.
    b. **Handling Data Skew**: Certain user_id values might be heavily skewed (e.g., users with high activity in User Interactions). To mitigate skew, we apply salting on the skewed user_id keys. This ensures that the data is distributed more evenly across partitions. The salt is added by concatenating a random integer (0–9) to the key.



Completed Jobs (4)

Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	collect at /opt/spark/spark_jobs/dataset_join.py:52 collect at /opt/spark/spark_jobs/dataset_join.py:52	2024/12/29 14:14:00	3 s	2/2 (1 skipped)	9/9 (14 skipped)
2	javaToPython at NativeMethodAccessorImpl.java:0 javaToPython at NativeMethodAccessorImpl.java:0	2024/12/29 14:13:54	6 s	1/1	14/14
1	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/12/29 14:13:52	0.1 s	1/1	1/1
0	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/12/29 14:13:48	2 s	1/1	1/1

5. Large-scale processing optimizations:

Bottlenecks Identified:

- **Shuffle and Broadcast Issues**: The `BroadcastExchange` (steps 9 and 18) indicates that a broadcast join is being performed. Although broadcast joins are efficient for small datasets, the `Statistics(sizeInBytes=18.0 MiB)` suggests a considerable dataset size. If the `BroadcastHashJoin` exceeds executor memory, it can cause performance degradation or OOM errors.
- **Skewed Keys Handling**: The addition of salted keys (steps 3 and 7) introduces extra computation (`rand()` to generate salts). If skewed keys are not well-distributed after salting, the shuffle and join stages (`BroadcastHashJoin`) can still be bottlenecked due to uneven partitioning.
- **Columnar to Row Conversion**: Steps 2 and 6 (`ColumnarToRow`) convert data formats for operations. These transformations can introduce computational overhead, especially when dealing with large datasets.
- **Scan and Projection**: Multiple `Scan parquet` (steps 1 and 5) and `Project` (steps 3, 7, 14, 16) stages indicate repeated I/O operations and transformations. This might be due to redundant logic in filtering and projection.

Shuffle Bottlenecks:

- **Join Operation**: The `BroadcastHashJoin` (steps 11 and 19) relies on `salted_user_id` for key matching. Uneven distribution of skewed keys across partitions after salting can still result in data skew during shuffle operations.
- **Filtering**: Filtering operations (`Filter` steps 4, 8, 15, and 17) are applied to large datasets before joining. If these filters are not selective enough, they can propagate large amounts of unnecessary data downstream.

Improvements to Address Bottlenecks:

- Use rand(seed) with a fixed seed to ensure repeatability during debugging and testing. Analyze the dataset skewed keys distribution and choose a salted threshold for the user_id column (5% of total).
- For large broadcast datasets, switching to a SortMergeJoin is the preferred solution (`.config("spark.sql.autoBroadcastJoinThreshold", -1) \ .config("spark.sql.join.preferSortMergeJoin", "true"`). This forces a SortMergeJoin and avoids executor memory pressure caused by broadcasting.
- Enabled the *bloomFilterIndex* in order to filter-out non-existing join keys, before performing the join;
- Repartitioned both datasets by the salted_user_id column in order to minimize shuffle and improve parallelism.

Memory Tuning Strategy:

1. **Driver Memory**:
    a. Manages job execution information, such as the DAG, task scheduling, and metadata.
    b. Sufficient memory is critical for handling large numbers of tasks or stages.
2. **Executor Memory**:
    a. Includes memory for:
        i. **Storage**: Cached data, broadcast variables, and shuffle files.
        ii. **Execution**: Task computation, intermediate results, and aggregations.
    b. Divided into execution, storage, and user memory (based on the Unified Memory Management model).
3. **Calculate Executor and Driver Memory**:
    a. executorMemory = heapMemory + overhead;
    b. Allocate approximately **75% of the total memory** to executor memory, leaving 25% for system processes and buffer;
    c. Divide executor memory into storage, execution, and user memory: Execution: ~60%, Storage: ~30%, User: ~10%;
    d. Driver memory depends on job complexity. Allocate additional memory for large datasets or high task parallelism.

Memory Allocation Calculation:

1. 100GB Data
    - Assumptions:
        - Input data size: 100GB.
        - Block size: 128MB → ~800 partitions.
    - Configuration:
        - Executors per node: 4 (8 cores per executor).
        - Executor memory per node: (128GB * 0.75) / 4 = 24GB.
        - Total executors: 4 * 16 = 64.
        - Memory per executor:

        - Heap = 24GB - 2GB (overhead) = 22GB
        - Execution memory = 22GB * 0.6 = 13.2GB
        - Storage memory = 22GB * 0.3 = 6.6GB
        - User memory = 22GB * 0.1 = 2.2GB

2. 500GB Data
    - Assumptions:
        - Input data size: 500GB.
        - Block size: 128MB → ~4000 partitions.
    - Configuration:
        - Executors per node: 4 (8 cores per executor).
        - Executor memory per node: (128GB * 0.75) / 4 = 24GB.
        - Total executors: 4 * 16 = 64.
        - Memory per executor:
            - Heap = 24GB - 2GB (overhead) = 22GB
            - Execution memory = 22GB * 0.6 = 13.2GB
            - Storage memory = 22GB * 0.3 = 6.6GB
            - User memory = 22GB * 0.1 = 2.2GB
        - Total partitions = 4000 (split across 64 executors → ~63 partitions per executor).
3. 1TB Data
    - Assumptions:
        - Input data size: 1TB.
        - Block size: 128MB → ~4800 partitions.
    - Configuration:
        - Executors per node: 4 (8 cores per executor).
        - Executor memory per node: (128GB * 0.75) / 4 = 24GB.
        - Total executors: 4 * 16 = 64.
        - Memory per executor:
            - Heap = 24GB - 2GB (overhead) = 22GB
            - Execution memory = 22GB * 0.6 = 13.2GB
            - Storage memory = 22GB * 0.3 = 6.6GB
            - User memory = 22GB * 0.1 = 2.2GB
        - Total partitions = 8000 (split across 64 executors → ~125 partitions per executor).

System Architecture Diagram

The Architecture Diagram can be found in ***Spark_Architecture_diagram.jpg***, explaining how the spark jobs can be deployed into a Kubernetes cluster, using spark-submit. This architecture showcases a distributed Spark application running on Kubernetes with a batch processing model. It incorporates the necessary components for processing, storage, orchestration, and monitoring, with the flexibility and scalability provided by Kubernetes.

Kubernetes manages the deployment, scaling, and orchestration of Spark clusters. If more worker nodes are required, Kubernetes can dynamically scale the number of pods running Spark Executors.

Kubernetes orchestrates the Spark jobs, schedules jobs, and manages containers. It can use Kubernetes CronJobs for periodic batch processing jobs or trigger them based on events.