Sort-Merge-Join in Spark

Akash Dwivedi · Follow

3 min read · Sep 14, 2020

Working with huge datasets and performing joins is always a bottleneck. Most of the time job gets failed due to `java.lang.OutOfMemoryError` . In this blog I am trying to provide a join method that is robust enough to handle huge datasets and if used smartly then fast enough to provide the desired results.

Sort-Merge-Bucket Join is a combination of two steps. In the first step it orders the joined data-sets. The second operation is the merge of sorted data into a single place by simply iterating over the elements and assembling the rows having the same value for the join key.

In SPARK Broadcast Hash join seems to be most performant join strategy , it is applicable to a small set of scenarios where datasets are able to fit in hash partition memory. Shuffle Hash Join & Sort Merge Join are the true work-horses of Spark SQL.

The property which leads to setting the Sort-Merge Join :

```
spark.sql.join.preferSortMergeJoin
```

The class involved in sort-merge join we should mention

```
org.apache.spark.sql.execution.joins.SortMergeJoinExec
```

Below is the simple script which shows you how Sort-Merge-Join works.

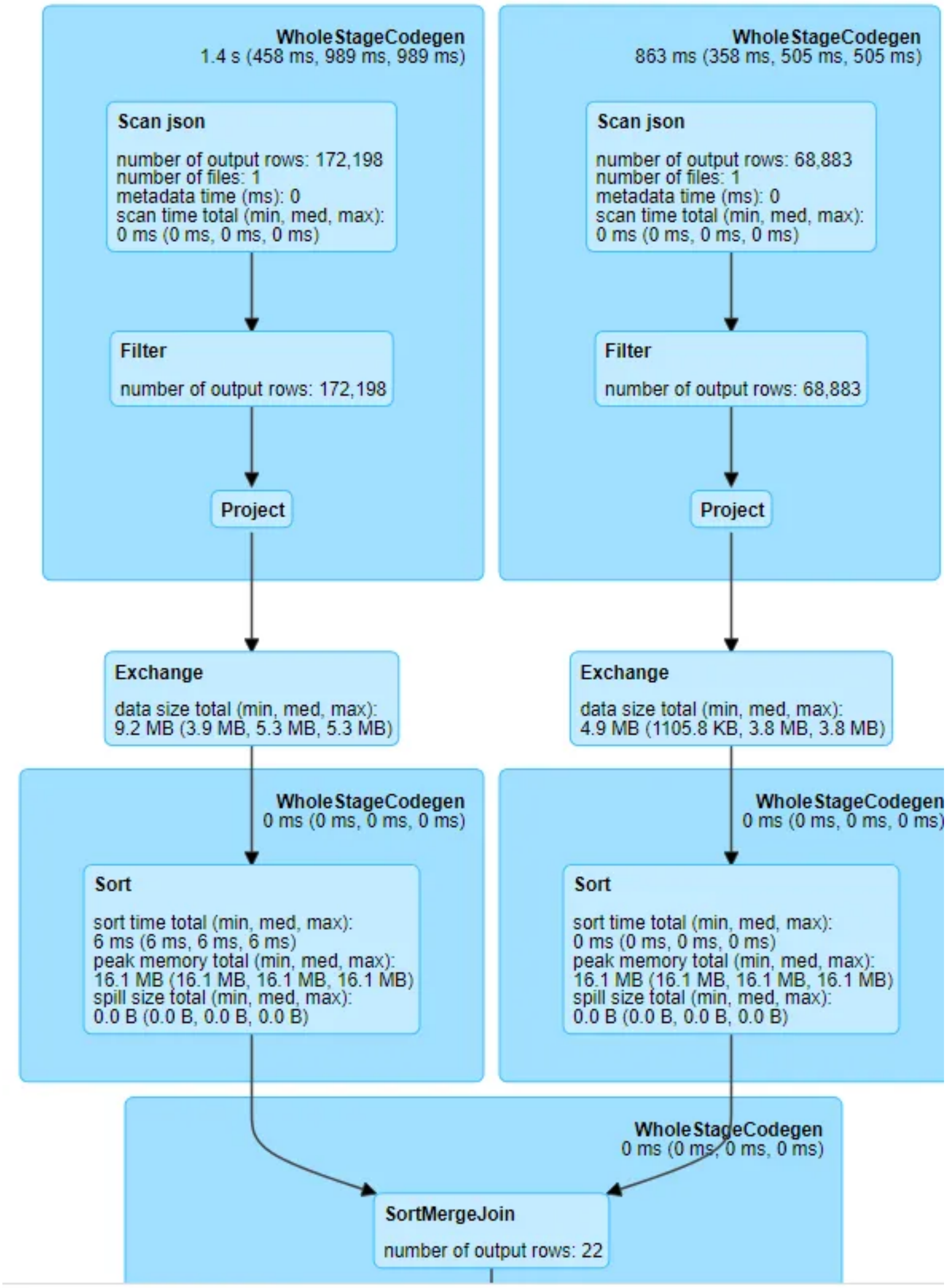
```
from pyspark.sql import SparkSession

 spark = SparkSession.builder\
         .appName("sort-merge-analysis")\
         .master("yarn")\
         .config("spark.sql.join.preferSortMergeJoin", "true")\
         .config("spark.sql.autoBroadcastJoinThreshold", "1")\
         .config("spark.sql.defaultSizeInBytes", "100000")\
         .enableHiveSupport()\
         .getOrCreate()

 orders = spark.read.json('retail_db_json/orders')
 order_item = spark.read.json('retail_db_json/order_items')
 wide_table = order_item.join(orders ,
orders.order_id==order_item.order_item_order_id)
 wide_table.show()
```

Here is the query Execution plan:

```
== Physical Plan ==
 *SortMergeJoin [order_item_order_id#26L], [order_id#10L], Inner
 :- *Sort [order_item_order_id#26L ASC NULLS FIRST], false, 0
 :  +- Exchange hashpartitioning(order_item_order_id#26L, 200)
 :     +- *Project [order_item_id#25L, order_item_order_id#26L,
order_item_product_id#27L, order_item_product_price#28,
order_item_quantity#29L, order_item_subtotal#30]
 :        +- *Filter isnotnull(order_item_order_id#26L)
 :           +- *FileScan json
```


WholeStageCodegen
1.4 s (458 ms, 989 ms, 989 ms)
Scan json
number of output rows: 172,198
number of files: 1
metadata time (ms): 0
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
Filter
number of output rows: 172,198
Project
WholeStageCodegen
863 ms (358 ms, 505 ms, 505 ms)
Scan json
number of output rows: 68,883
number of files: 1
metadata time (ms): 0
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
Filter
number of output rows: 68,883
Project
Exchange
data size total (min, med, max):
9.2 MB (3.9 MB, 5.3 MB, 5.3 MB)
Exchange
data size total (min, med, max):
4.9 MB (1105.8 KB, 3.8 MB, 3.8 MB)
WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)
Sort
sort time total (min, med, max):
6 ms (6 ms, 6 ms, 6 ms)
peak memory total (min, med, max):
16.1 MB (16.1 MB, 16.1 MB, 16.1 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)
Sort
sort time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
peak memory total (min, med, max):
16.1 MB (16.1 MB, 16.1 MB, 16.1 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)
SortMergeJoin
number of output rows: 22

Deep Dive

The Spark SQL planner chooses to implement the join operation using 'SortMergeJoin'. The precedence order for equi-join implementations (as in Spark 2.2.0) is as follows:

1. Broadcast Hash Join
2. Shuffle Hash Join: if the average size of a single partition is small enough to build a hash table.
3. Sort Merge: if the matching join keys are sortable.

Next thing which requires attention is Bucketing.

Bucketing is one of the famous optimization technique which is used to avoid data shuffle.

S-M-B joins gives its best performance when the datasets use buckets to store the data.

Bucketing is used exclusively in **FileSourceScanExec** physical operator (when it is requested for the input RDD and to determine the partitioning and ordering of the output).

In the above DAG if we inspect properly, we can clearly see there is an **Exchange** stage present. When we use **pre-shuffled bucketed tables** on join key, we take the advantage of offset provided by bucketing.

Below I will try to run the same operation and this time the two tables will be pre-shuffled bucketed on join key.

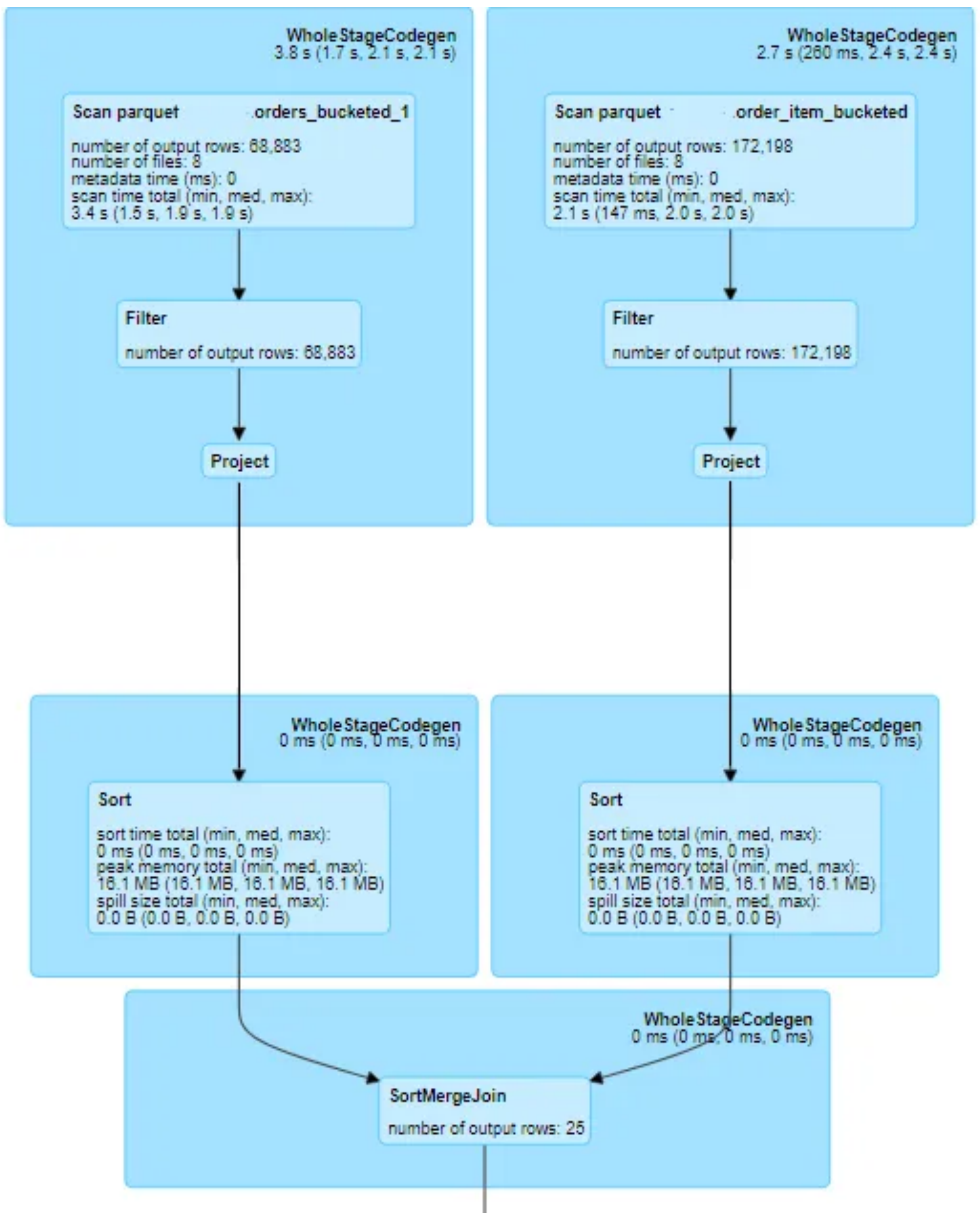
Here is the code snippet

```
from pyspark.sql import SparkSession

 spark = SparkSession.builder\
         .appName("sort-merge-bucket")\
         .master("yarn")\
         .config("spark.sql.join.preferSortMergeJoin", "true")\
         .config("spark.sql.autoBroadcastJoinThreshold", "1")\
         .config("spark.sql.defaultSizeInBytes", "100000")\
         .enableHiveSupport()\
         .getOrCreate()orders = spark.sql("select * from
orders_bucketed")
order_item = spark.sql("select * from
order_item_bucketed")wide_table = order_item.join(orders ,
orders.order_id==order_item.order_item_order_id)
```

```
wide_table.show()
```

DAG generated


WholeStageCodegen
3.8 s (1.7 s, 2.1 s, 2.1 s)
Scan parquet orders_bucketed_1
number of output rows: 68,883
number of files: 8
metadata time (ms): 0
scan time total (min, med, max):
3.4 s (1.5 s, 1.9 s, 1.9 s)
Filter
number of output rows: 68,883
Project
WholeStageCodegen
2.7 s (260 ms, 2.4 s, 2.4 s)
Scan parquet order_item_bucketed
number of output rows: 172,198
number of files: 8
metadata time (ms): 0
scan time total (min, med, max):
2.1 s (147 ms, 2.0 s, 2.0 s)
Filter
number of output rows: 172,198
Project
WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)
Sort
sort time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
peak memory total (min, med, max):
16.1 MB (16.1 MB, 16.1 MB, 16.1 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)
Sort
sort time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
peak memory total (min, med, max):
16.1 MB (16.1 MB, 16.1 MB, 16.1 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)
SortMergeJoin
number of output rows: 25

As we can clearly see there is no exchange step present in DAG

Few Observations:

- The number of buckets has to be between 0 and 100000 exclusive or Spark SQL throws an *AnalysisException*

- The number of partitions on both sides of a join must be exactly the same.
- As of Spark 2.4, Spark SQL supports **bucket pruning** to optimize filtering on bucketed column (by reducing the number of bucket files to scan).

Spark Data Engineering Big Data Big Data Analytics

Follow

Written by Akash Dwivedi

24 Followers

Data Engineer | Expedia Groups

More from Akash Dwivedi

Akash Dwivedi in Data Engineer Things

Consistency Model in Cassandra

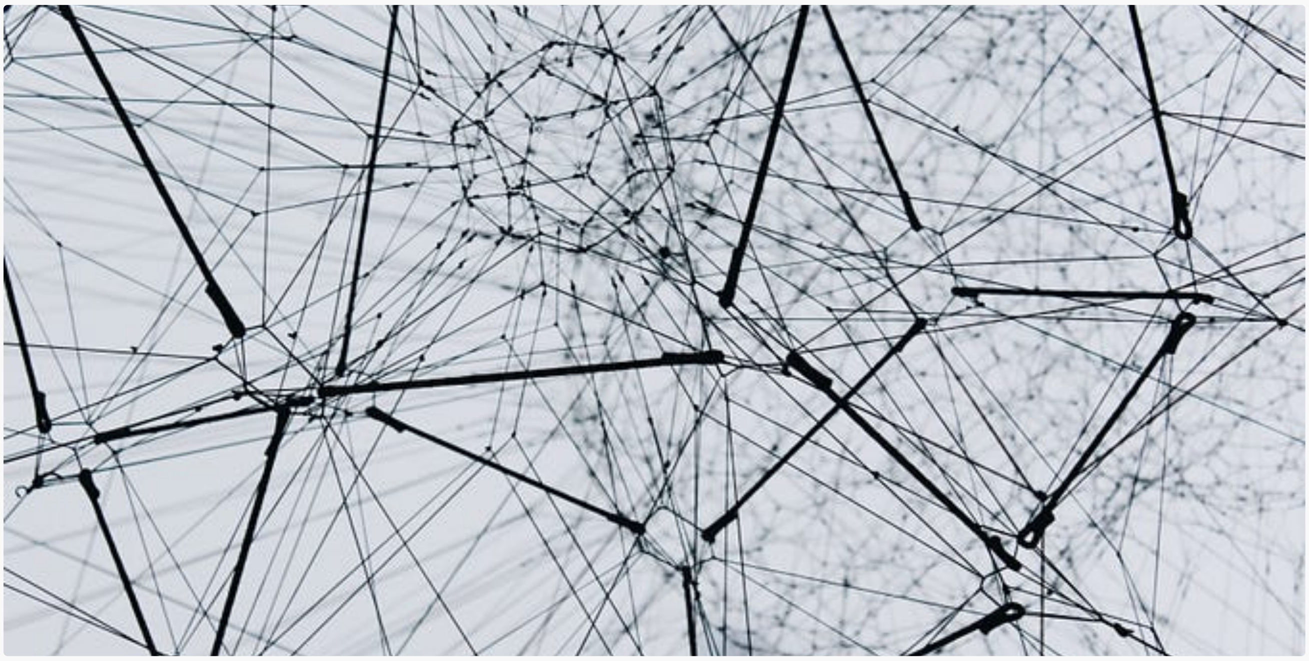
A guide to tune consistency in cassandra

7 min read · Mar 16, 2023

83 2

See all from Akash Dwivedi

Recommended from Medium



Subham Khandelwal in Dev Genius

PySpark — Optimize Joins in Spark

Shuffle Hash Join, Sort Merge Join, Broadcast joins and Bucketing for better Join Performance

149 1

Ashwin

Spark Repartition vs Coalesce

Are you struggling with optimizing the performance of your Spark application? If so, understanding the key differences between the...

· 6 min read · Dec 25, 2023

2

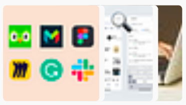
Lists

Natural Language Processing

1417 stories · 911 saves

Staff Picks

632 stories · 930 saves



Interesting Design Topics

257 stories · 498 saves

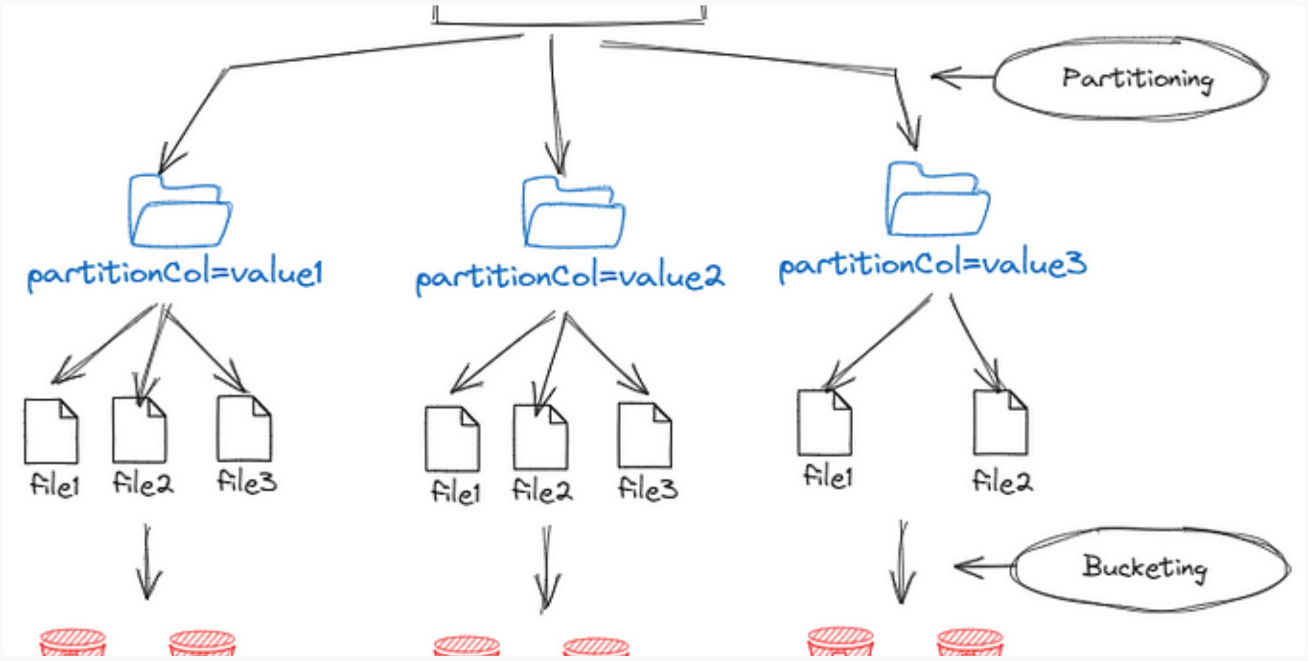
Sai Prabhanj Turaga

Spark — Handling Duplicates

Handling duplicate rows or specific column duplicates in Spark can be important when working with large datasets. You can identify and...

3 min read · Nov 6, 2023

90



Kerrache Massipssa in Data Engineer Things

Apache Spark Partitioning and Bucketing

Learn the Partitioning and Bucketing with Apache Spark (PySpark) and understand how and when to use each of them.

· 5 min read · Dec 14, 2023

178 1

Hari prasad

DropDuplicate, Distinct and GroupBy in Apache Spark | Efficiently Remove Duplicates/Redundant Data

How DropDuplicate, Distinct and GoupBy works in Apache Spark and Scenarios to use them to remove duplicates in data efficiently or...

5 min read · Jan 21, 2024

1

Michael Berk in Towards Data Science

1.5 Years of Spark Knowledge in 8 Tips

My learnings from Databricks customer engagements

8 min read · Dec 24, 2023

1.8K 11

See more recommendations