

datalex



experts in data, cloud  
architecture & SaaS—  
data lover

[Follow publication](#)

# Spark's Logical and Physical plans ... When, Why, How and Beyond.



Laurent Leturgez

Follow

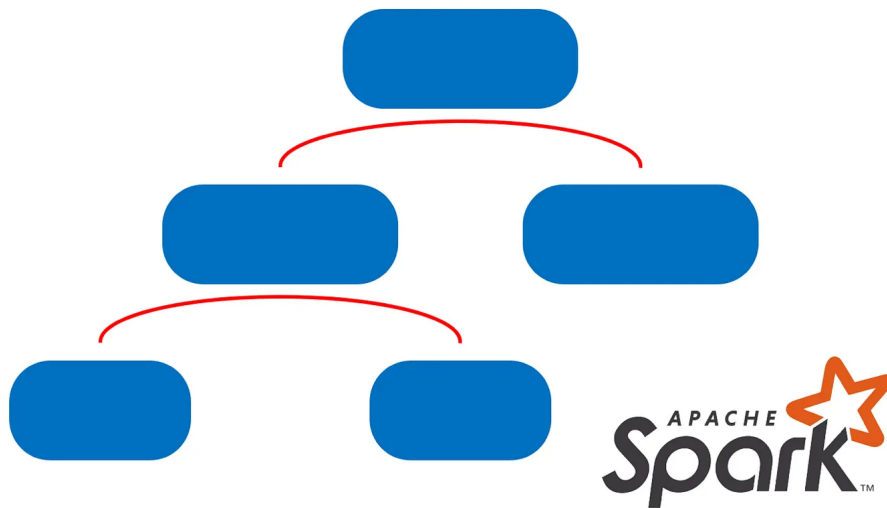
7 min read · Jul 23, 2020



324



7



There's a long time I didn't write something in a blog since I worked with Cloud technologies and specially Apache Spark (My old blog was dedicated to Data engineering and architecting Oracle databases ... here: <https://laurent-leturgez.com>)

Now I've stepped to BigData technologies, I've decided to write some posts on Medium and my first post is about a topic that is quite close to an Oracle database topic ... Apache Spark's execution plan.

## Execution plan aim

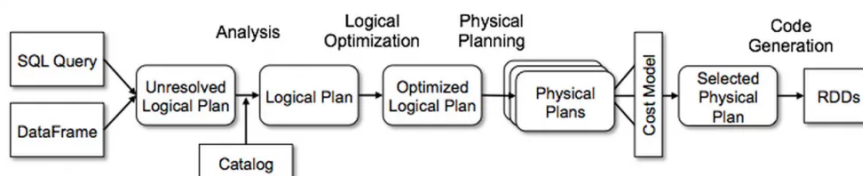
An execution plan is the set of operations executed to translate a query language statement (SQL, Spark SQL, Dataframe operations etc.) to a set of optimized logical and physical operations.

To sum up, it's a set of operations that will be executed from the SQL (or Spark SQL) statement to the DAG which will be send to Spark Executors.

[Top highlight](#)

*If you don't know what a DAG is, it stands for "Directed Acyclic Graph". A DAG is an acyclic graph produced by the DAGScheduler in Spark. As a graph, it is composed of vertices and edges that will represent RDDs and operations (transformations and actions) performed on them.*

On Spark, the optimizer is named "Catalyst" and can be represented by the schema below. It will produce different types of plans:



Operation names are:

- Analysis
- Logical Optimization
- Physical Planning
- Cost Model Analysis
- Code Generation

And those operations will produce various plans:

- Unresolved logical plan
- Resolved logical plan
- Optimized logical plan
- Physical plans

The goal of all these operations and plans is to produce automatically the most effective way to process your query.

## Generating plans

In order to generate plans, you have to deal with Dataframes regardless they come from SQL or raw dataframe. And the function you will use is (in Python) `explain()`.

For example, if you have these two dataframes:

```
itemsSchema = ("id integer, name string, price float")
ordersSchema= ("id integer, itemid integer, count integer")

items = spark.createDataFrame([[0, "Tomato", 2.0], \
                               [1, "Watermelon", 5.5], \
                               [2, "pineapple", 7.0]], \
                              schema=itemsSchema)

orders = spark.createDataFrame([[100, 0, 1], \
                                [100, 1, 1], \
                                [101, 2, 3], \
                                [102, 2, 8]], \
                               schema=ordersSchema)
```

You can manipulate them like this:

```
from pyspark.sql.functions import sum

y=(items.join(orders,items.id==orders.itemid, how="inner"))\
    .where(items.id==2)\
    .groupBy("name","price").agg(sum("count")\
    .alias("c"))
```

or like this:

```
items.createOrReplaceTempView("ITEMS")
orders.createOrReplaceTempView("ORDERS")

x=sql('''select ITEMS.name,
              ITEMS.price,
              SUM(ORDERS.count) as c
from ITEMS, ORDERS
where ITEMS.id=ORDERS.itemid
and ITEMS.id=2
group by ITEMS.name, ITEMS.price''')
```

In both cases, you will be able to call explain():

```
x.explain()
y.explain()
```

By default, calling explain with no argument will produce a physical plan explanation :

```
== Physical Plan ==
*(4) HashAggregate(keys=[name#765, price#766], functions=[finalmerge_sum(merge sum#817L) AS sum(cast(count#772 as bigint))#783L])
+- Exchange hashpartitioning(name#765, price#766, 200), true, [id=#4134]
   +- *(3) HashAggregate(keys=[name#765, knownfloatingpointnormalized(normalizenanandzero(price#766)) AS price#766], functions=[partial_sum(cast(count#772 as bigint)) AS sum#817L])
      +- *(3) Project [name#765, price#766, count#772]
         +- *(3) SortMergeJoin [id#764], [itemid#771], Inner
            :- Sort [id#764 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(id#764, 200), true, [id=#4123]
                  :- *(1) Filter (isnotnull(id#764) AND (id#764 = 2))
                     +- *(1) Scan ExistingRDD[id#764,name#765,price#766]
            +- Sort [itemid#771 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(itemid#771, 200), true, [id=#4127]
                  +- *(2) Project [itemid#771, count#772]
                     +- *(2) Filter ((itemid#771 = 2) AND isnotnull(itemid#771))
                        +- *(2) Scan ExistingRDD[id#770,itemid#771,count#772]
```

A spark physical plan explained

## Getting various plans

Before Apache Spark 3.0, there was only two modes available to format explain output.

- *explain(extended=False)* which displayed only the physical plan
- *explain(extended=True)* which displayed all the plans (logical and physical)

```
x.explain(extended=true)
```

```
== Analyzed Logical Plan ==
name: string, price: float, c: bigint
Aggregate [name#2222, price#2223], [name#2222, price#2223, sum(cast(count#2229 as bigint)) AS c#2239L]
+- Filter ((id#2221 = itemid#2228) AND (id#2221 = 2))
   +- Join Inner
      :- SubqueryAlias items
      : +- LogicalRDD [id#2221, name#2222, price#2223], false
      +- SubqueryAlias orders
      :- LogicalRDD [id#2227, itemid#2228, count#2229], false

== Optimized Logical Plan ==
Aggregate [name#2222, price#2223], [name#2222, price#2223, sum(cast(count#2229 as bigint)) AS c#2239L]
+- Project [name#2222, price#2223, count#2229]
   +- Join Inner, (id#2221 = itemid#2228)
      :- Filter (isnotnull(id#2221) AND (id#2221 = 2))
      : +- LogicalRDD [id#2221, name#2222, price#2223], false
      +- Project [itemid#2228, count#2229]
         +- Filter ((itemid#2228 = 2) AND isnotnull(itemid#2228))
         :- LogicalRDD [id#2227, itemid#2228, count#2229], false

== Physical Plan ==
*(4) HashAggregate(keys=[name#2222, price#2223], functions=[finalmerge_sum(merge sum#2274L) AS sum(cast(count#2229 as bigint))#2246L], output=[name#2222, price#2223, c#2239L])
+- Exchange hashpartitioning(name#2222, price#2223, 200), true, [id=#5183]
   +- *(3) HashAggregate(keys=[name#2222, knownFloatingpointNormalized(normalizeNaNandZero(price#2223)) AS price#2223], functions=[partial_sum(cast(count#2229 as bigint)) AS sum#2274L], output=[name#2222, price#2223, sum#2274L])
      +- *(3) Project [name#2222, price#2223, count#2229]
         +- *(3) SortMergeJoin [id#2221], [itemid#2228], Inner
            :- Sort [id#2221 ASC NULLS FIRST], false, 0
            : +- Exchange hashpartitioning(id#2221, 200), true, [id=#5172]
            : +- *(1) Filter (isnotnull(id#2221) AND (id#2221 = 2))
            : +- *(1) Scan ExistingRDD[id#2221,name#2222,price#2223]
            +- Sort [itemid#2228 ASC NULLS FIRST], false, 0
            :- Sort [itemid#2228 ASC NULLS FIRST], false, 0
            : +- Exchange hashpartitioning(itemid#2228, 200), true, [id=#5176]
            +- *(2) Project [itemid#2228, count#2229]
               +- *(2) Filter ((itemid#2228 = 2) AND isnotnull(itemid#2228))
               +- *(2) Scan ExistingRDD[id#2227,itemid#2228,count#2229]
```

Spark extended execution plans (logical to physical)

Starting from Apache Spark 3.0, you have a new parameter “*mode*” that produce expected format for the plan:

- *explain(mode="simple")* which will display the physical plan
- *explain(mode="extended")* which will display physical and logical plans (like “extended” option)
- *explain(mode="codegen")* which will display the java code planned to be executed
- *explain(mode="cost")* which will display the optimized logical plan and related statistics (if they exist)
- *explain(mode="formatted")* which will display a splitted output composed by a nice physical plan outline, and a section with each node details

```
x.explain(mode="formatted")
```

```

== Physical Plan ==
* HashAggregate (14)
+- Exchange (13)
   +- * HashAggregate (12)
      +- * Project (11)
         +- * SortMergeJoin Inner (10)
            :- Sort (4)
               : +- Exchange (3)
               :   +- * Filter (2)
               :   +- * Scan ExistingRDD (1)
            +- Sort (9)
               +- Exchange (8)
                  +- * Project (7)
                     +- * Filter (6)
                        +- * Scan ExistingRDD (5)

(1) Scan ExistingRDD [codegen id : 1]
Output [3]: [id#2525, name#2526, price#2527]
Arguments: [id#2525, name#2526, price#2527], MapPartitionsRDD[602] at applySchemaToPythonRDD at <unknown>:0, ExistingRDD, UnknownPartitioning(0)

(2) Filter [codegen id : 1]
Input [3]: [id#2525, name#2526, price#2527]
Condition : (isnotnull(id#2525) AND (id#2525 = 2))

(3) Exchange
Input [3]: [id#2525, name#2526, price#2527]
Arguments: hashpartitioning(id#2525, 200), true, [id=#6140]

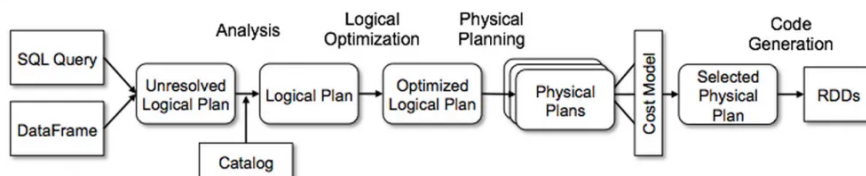
```

New spark 3.0 explain plan formatted output

*What is fun with this formatted output is not so exotic if you come, like me, from the rdbms world 🤔*

## Plans differences

As stated in the beginning of this post, various kinds of plans are generated after many operations processed by the Catalyst Optimizer:



### First step: Unresolved Logical plan generation

This plan is generated after a first check that verifies everything is correct on the syntactic field. Next, the semantic analysis is executed and will produced a first version of a logical plan where relation name and columns are not specifically resolved. This produced this kind of result:

```

== Parsed Logical Plan ==
'Aggregate ['name', 'price'], [unresolvedalias('name', None), unresolvedalias('price', None), sum('count') AS c#2709]
+- Filter (id#2661 = 2)
   +- Join Inner, (id#2661 = itemid#2668)
      :- LogicalRDD [id#2661, name#2662, price#2663], false
      +- LogicalRDD [id#2667, itemid#2668, count#2669], false

```

Spark Unresolved logical plan

### Next step: [Analyzed] Logical plan generation

When the unresolved plan has been generated, it will resolve everything that is not resolved yet by accessing an internal Spark structure mentioned as “Catalog” in the previous schema. In this catalog, which can be assimilated to a metastore, a semantic analysis will be produced to verify data structures, schemas, types etc. and if everything goes well, the plan is marked as “Analyzed Logical Plan” and will be formatted like this:

```
== Analyzed Logical Plan ==
name: string, price: float, c: bigint
Aggregate [name#2662, price#2663], [name#2662, price#2663, sum(cast(count#2669 as bigint)) AS c#2709L]
+- Filter (id#2661 = 2)
   +- Join Inner, (id#2661 = itemid#2668)
      :- LogicalRDD [id#2661, name#2662, price#2663], false
      +- LogicalRDD [id#2667, itemid#2668, count#2669], false
```

Spark resolved or analyzed logical plan

We can see here that, just after the “Aggregate” line, all the previously marked “unresolved alias” are now resolved and correctly typed ... specially the sum column.

### Next step: Optimized logical Plan

Once the Logical plan has been produced, it will be optimized based on various rules applied on logical operations (But you have already noticed that all these operations were logical ones: filters, aggregation etc.).

These logical operations will be reordered to optimize the logical plan.

When the optimization ends, it will produced this kind of output:

```
== Optimized Logical Plan ==
Aggregate [name#2746, price#2747], [name#2746, price#2747, sum(cast(count#2753 as bigint)) AS c#2763L]
+- Project [name#2746, price#2747, count#2753]
   +- Join Inner, (id#2745 = itemid#2752)
      :- Filter (isnotnull(id#2745) AND (id#2745 = 2))
         :- LogicalRDD [id#2745, name#2746, price#2747], false
      +- Project [itemid#2752, count#2753]
         +- Filter ((itemid#2752 = 2) AND isnotnull(itemid#2752))
            +- LogicalRDD [id#2751, itemid#2752, count#2753], false
```

Spark optimized logical plan

We can see in this plan, that predicates have been pushed down on the LogicalRDD to reduce the data volume processed by the join. We can mention too that filters are pushed to both data structure (one for the “items” dataframe, and one for the “orders” dataframe).

### Then, many physical plans are generated and finally a unique physical plan is selected

Laurent Leturgez highlighted

From the optimized logical plan, a plan that describes how it will be physically executed on the cluster will be generated. But before selecting a physical plan, the Catalyst Optimizer will generate many physical plans based on various strategies. Each physical plan will be estimated based on execution time and resource consumption projection and only one plan will be selected to be executed.

Based on our example, the selected physical plan is this one (which is the one that is printed when you use `explain()` with default parameters)

```

== Physical Plan ==
*(4) HashAggregate(keys=[name#10, price#11], functions=[finalmerge_sum(merge sum#72L) AS sum(cast(count#17
as bigint))#50L], output=[name#10, price#11, c#51L])
+- Exchange hashpartitioning(name#10, price#11, 200), true, [id=#299]
   +- *(3) HashAggregate(keys=[name#10, knownfloatingpointnormalized(normalizenanandzero(price#11)) AS pric
e#11], functions=[partial_sum(cast(count#17 as bigint)) AS sum#72L], output=[name#10, price#11, sum#72L])
      +- *(3) Project [name#10, price#11, count#17]
         +- *(3) SortMergeJoin [id#9], [itemid#16], Inner
            :- Sort [id#9 ASC NULLS FIRST], false, 0
               : +- Exchange hashpartitioning(id#9, 200), true, [id=#288]
                  : +- *(1) Filter (isnotnull(id#9) AND (id#9 = 2))
                     : +- *(1) Scan ExistingRDD[id#9,name#10,price#11]
            +- Sort [itemid#16 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(itemid#16, 200), true, [id=#292]
                  +- *(2) Project [itemid#16, count#17]
                     +- *(2) Filter ((itemid#16 = 2) AND isnotnull(itemid#16))
                        +- *(2) Scan ExistingRDD[id#15,itemid#16,count#17]

```

Spark physical plan

## Spark execution is Lazy, Execution plan generation is not a lazy thing !

Calling `explain()` function is an operation that will produce all the stuff presented above, from the unresolved logical plan to a selection of one physical plan to execute.

After that, and only after that, the physical plan is executed through one to many stages and tasks in a laziness way.

As a consequence, it won't be possible to generate an unresolved logical plan by typing something like the code below (which includes a schema error: `ids` instead of `id`)

```

>>> (items.join(orders,items.id==orders.itemid, how="inner"))\
...     .where(items.ids==2)\
...     .groupBy("name","price").agg(sum("count")\
...     .alias("c")).explain(mode="extended")

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "/usr/spark-3.0.0/python/pyspark/sql/dataframe.py", line 1401,
in __getattr__
    "'%s' object has no attribute '%s'" % (self.__class__.__name__,
name))
AttributeError: 'DataFrame' object has no attribute 'ids'

```

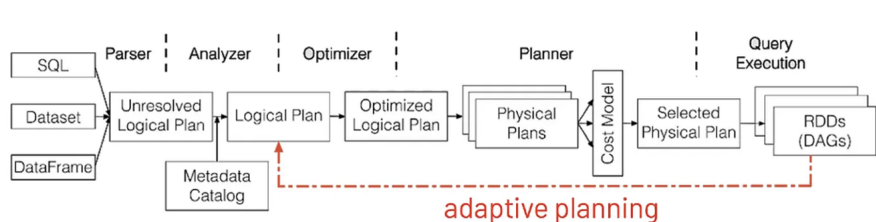
## And how AQE (Adaptive Query Execution) interact with this stuff ???

AQE is a new feature in Spark 3.0 which enables plan changes at runtime.

It collects statistics during plan execution and if Spark detects better plan during execution, it changes them at runtime.

If we put this on an update of the catalyst Optimizer schema, it will give

something like that:



However, any changes decided during DAG execution won't be displayed after calling `explain()` function. If you want to see these changes, you will have to explore Spark UI and tracking skew partitions splits, joins changes etc.

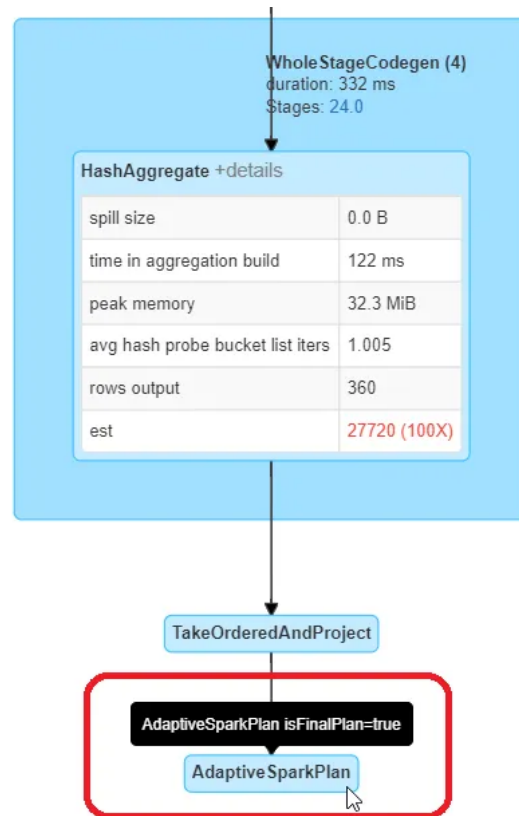
For example, if you enable AQE (which is not enabled by default), explain call will produce a hint in the physical plan display :

```

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Exchange rangepartitioning(total_sales#544L DESC NULLS LAST), true, 0
   +- HashAggregate(keys=[s_date#441], functions=[finalmerge_sum(merge sum#550L) AS sum(cast((s_quantity#440 * i_price#443) as big
int))#545L], output=[s_date#441, total_sales#544L])
      +- Exchange hashpartitioning(s_date#441, 200), true, [id=#2813]
         +- HashAggregate(keys=[s_date#441], functions=[partial_sum(cast((s_quantity#440 * i_price#443) as bigint)) AS sum#550L],
output=[s_date#441, sum#550L])
            +- Project [s_quantity#440, s_date#441, i_price#443]
               +- SortMergeJoin [cast(s_item_id#439 as bigint)], [i_item_id#442L], Inner
                  :- Sort [cast(s_item_id#439 as bigint) ASC NULLS FIRST], false, 0
                     : +- Exchange hashpartitioning(cast(s_item_id#439 as bigint), 200), true, [id=#2805]
                        : +- Project [s_item_id#439, s_quantity#440, s_date#441]
                           : +- Filter isNotNull(s_item_id#439)
                              : +- FileScan parquet aqe_demo.db.sales[s_item_id#439,s_quantity#440,s_date#441] Batched: true, DataFi
lters: [isNotNull(s_item_id#439)], Format: Parquet, Location: InMemoryFileIndex[dbfs:/user/hive/warehouse/ae_demo.db/db/sales], Part
itionFilters: [], PushedFilters: [IsNotNull(s_item_id)], ReadSchema: struct<s_item_id:int,s_quantity:int,s_date:date>
                  +- Sort [i_item_id#442L ASC NULLS FIRST], false, 0
                     +- Exchange hashpartitioning(i_item_id#442L, 200), true, [id=#2806]
                        +- Project [i_item_id#442L, i_price#443]
                           +- Filter isNotNull(i_item_id#442L)
                              +- FileScan parquet aqe_demo.db.items[i_item_id#442L,i_price#443] Batched: true, DataFilters: [isno
tNull(i_item_id#442L)], Format: Parquet, Location: InMemoryFileIndex[dbfs:/user/hive/warehouse/ae_demo.db/db/items], PartitionFilter
s: [], PushedFilters: [IsNotNull(i_item_id)], ReadSchema: struct<i_item_id:bigint,i_price:int>
  
```

In the explain output, you are hinted this physical plan is not the final plan, but if you have a look at the Spark UI, you will see that the SQL query plan is the final plan and optionally if the plan has been modified :





And ... that's all for today 😊

*Laurent*



### Published in datalex

99 followers · Last published Apr 1, 2024

experts in data, cloud architecture & SaaS—data lover

Follow



### Written by Laurent Leturgez

396 followers · 42 following

Specialist Solutions Architect @ Databricks - Lille, France

Follow

## Responses (7)



Aaditya Adhikari

What are your thoughts?



Ayoublaarouchi

Sep 10, 2024



straight to the point, easy to follow and understand.



[Reply](#)



Subscriptions Smida Marwen

Mar 11, 2024



Very strait forward and well explained, thank you



[Reply](#)



Vinayak Shettigar

Oct 24, 2023



Useful post.Thank you!!!



[Reply](#)

[See all responses](#)

## More from Laurent Leturgez and datalex

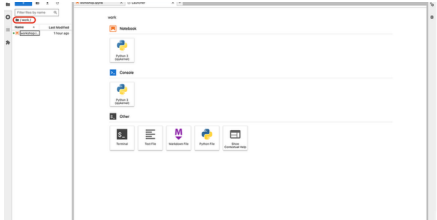


In datalex by Laurent Leturgez

### 5 reasons to choose Delta Lake format (on Databricks)

5 reasons to prefer the Delta Lake format to parquet or ORC when you are using

Sep 27, 2021 163 4

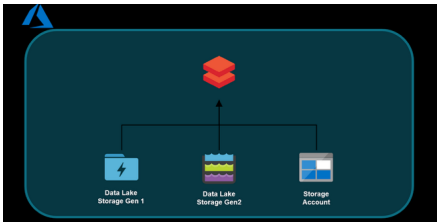


In datalex by Alexandre Bergere

### Running PySpark on Jupyter Notebook with Docker on Mac: A

In the realm of big data processing, PySpark has emerged as a potent tool for harnessing

Mar 5, 2024 6 1

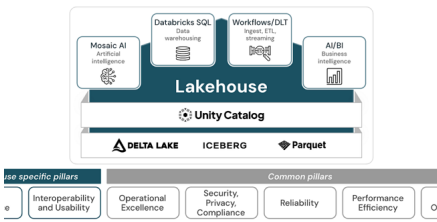


In datalex by Alexandre Bergere

### Azure Databricks—Connect Azure storage to Databricks

When you're working with Azure Databricks, you need some storage for your amount of

Aug 11, 2023 8



In Data Warehouse migrations to by Laurent Letu

### Top 10 Tips for DBAs Migrating to Databricks

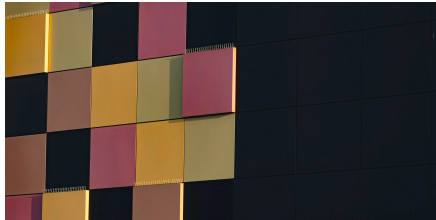
As organizations increasingly embrace cloud-based data platforms, Database

May 6 8

See all from Laurent Leturgez

See all from datalex

## Recommended from Medium



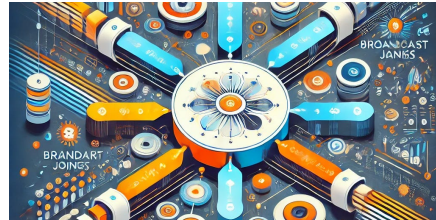
 Vijay Gadhave

## How Apache Spark Uses CPU, Memory, and Storage

Note: If you're not a medium member, [CLICK HERE](#)

★ Jan 6 🖱 3

🔖 ...



 Diogo Santos

## Mastering Apache Spark Shuffle Optimization: Techniques for

Explore the power of bucketing, repartitioning, and broadcast joins to

★ Jan 26 🖱 64

🔖 ...



 Sahil Sharma

## Mastering SCD in Data Engineering: Key Tips for Interview

Learn how to implement efficient and scalable Slowly Changing Dimensions (SCD)

★ Jan 9 🖱 7 💬 1

🔖 ...



 Abhishek Jadhav


## Data Engineering Projects That Employers Value

By Abhishek Jadhav

Feb 4 🖱 55

🔖 ...



 Mayurkumar Surani

## 🚀 Mastering PySpark: Your Complete Guide to 46 Essential

Collection of PySpark Functions

★ Jun 3 🖱 34

🔖 ...



 In Towards Data Engineeri by Ángel Álvarez Pasc

## Apache Spark WTF???— That Thing You Do! (Part II)

Here we go again! I warned you it wasn't going to take long before the second part of

Jan 13 🖱 13

🔖 ...

[See more recommendations](#)

---

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Rules](#) [Terms](#) [Text to speech](#)