Different Types of “Join Strategies” in “Apache Spark”

Oindrila Chakraborty · Follow
17 min read · Oct 6, 2023

Listen Share More

What is “Join Selection Strategy”?

- When “**Any Type**” of “**Join**”, like the “**Left Join**”, or, the “**Inner Join**” is “**Performed**” between “**Two DataFrames**”, “**Apache Spark**” “**Internally**” decides which “**Algorithm**” will be used to “**Perform**” the “**Join**” **Operations** between the “**Two DataFrames**”.

- That particular “**Algorithm**” that is “**Responsible**” for “**Planning**” the “**Join**” **Operation** between the “**Two DataFrames**”, is called as the “**Join Selection Strategy**”.

Why Learning About “Join Selection Strategies” is Important?

- To “**Optimize**” a “**Spark Job**” that “**Involves**” a “**Lot of Joins**”, the “**Developers**” need to be very much aware about the “**Internal Algorithm**” that “**Apache Spark**” will “**Choose**” to “**Perform**” “**Any**” of the “**Join**” **Operations** between “**Two DataFrames**”.

- The “**Developers**” need to know about the “**Join Selection Strategies**” so that the “**Wrong Join Selection Strategy**” is “**Not Used**” in the “**Join**” **Operation** between “**Two DataFrames**”.

- An “**Incorrect Join Selection Strategy**” will “**Increase**” the “**Execution Time**” of the “**Join**” **Operation**, and, the “**Join**” **Operation** becomes a “**Heavy Operation**” on the “**Executors**” as well.

In Which Phase the “Join Selection Strategy” is “Selected”?

- “**Apache Spark**” decides which “**Algorithm**” will be used to “**Perform**” the “**Join**” **Operation** between “**Two DataFrames**” in the “**Physical Planning**” **Phase**, where “**Each Node**” in the “**Logical Plan**” has to be “**Converted**” to “**One**”, or, “**More**” “**Operators**” in the “**Physical Plan**” using the so-called “**Join Selection Strategies**”.

How “Many Types” of “Join Selection Strategies” are There in the “Apache Spark”?

- Following are the “**Join Selection Strategies**” that “**Apache Spark**” can “**Choose**” from in the time of a “**Join**” **Operation** between “**Two DataFrames**” -
  1. **Sort Merge Join**
  2. **Shuffle Hash Join**
  3. **Broadcast Hash Join**
  4. **Cartesian Join**
  5. **Broadcasted Nested Loop Join**

- In this article, I will explain about the first three types of “**Join Selection Strategies**”.

How “Apache Spark” Decides Which “Join Selection Strategy” to “Choose”?

- “**Apache Spark**” decides the “**Join Selection Strategy**” to use in a “**Join**” **Operation** based on the following criteria -

- “**Size**” of the “**Data**” of “**Each**” of the “**DataFrame**” involved in a “**Join**” **Operation**

- The “**Amount**” of “**Memory**” available to the “**Driver Node**”, and, the “**Worker Nodes**”
- “**Presence**” of “**Skewness**” in the “**Data**” of “**Each**” of the “**DataFrame**” involved in a “**Join**” **Operation**
- “**Characteristics**” of the “**Join Key**”, like — if the “**DataFrames**” involved in a “**Join**” **Operation** are “**Partitioned**” using the “**Join Key Column**”, or, the “**Join Key Columns**” that are “**Present**” in “**Both**” of the “**DataFrames**” involved in a “**Join**” **Operation** are “**Already Sorted**” etc.

Dataset Used -

Create Lists for “Person”.

```
# Create a List Containing the "Column Names" for "Person"
personColumns = ["Id", "First_Name", "Last_Name", "AddressId"]

# Create a List Containing the "Data" for "Person"
personList = [\
                (1001, "Oindrila", "Chakraborty", "A001"),
                (1002, "Soumyajyoti", "Bagchi", "A002"),
                (1003, "Oishi", "Bhattacharyya", "A004"),
                (1004, "Sabarni", "Chakraborty", "A003"),
                (1005, "Ayan", "Dutta", "A002"),
                (1006, "Dhrubajyoti", "Das", "A004"),
                (1007, "Sayantan", "Chatterjee", "A004"),
                (1008, "Ritwik", "Ghosh", "A001"),
                (1009, "Puja", "Bhatt", "A001"),
                (1010, "Souvik", "Roy", "A002"),
                (1011, "Souvik", "Roy", "A003"),
                (1012, "Ria", "Ghosh", "A003"),
                (1013, "Soumyajit", "Pal", "A002"),
                (1014, "Abhirup", "Chakraborty", "A004"),
                (1015, "Sagarneel", "Sarkar", "A003"),
                (1016, "Anamika", "Pal", "A002"),
                (1017, "Swaralipi", "Roy", "A002"),
                (1018, "Rahul", "Roychowdhury", "A003"),
                (1019, "Paulomi", "Mondal", "A004"),
                (1020, "Avishek", "Basu", "A002"),
                (1021, "Avirupa", "Ghosh", "A004"),
                (1022, "Ameer", "Sengupta", "A003"),
                (1023, "Megha", "Kargupta", "A002"),
                (1024, "Madhura", "Chakraborty", "A002"),
                (1025, "Debankur", "Dutta", "A002"),
                (1026, "Bidisha", "Das", "A001"),
                (1027, "Rohan", "Ghosh", "A004"),
                (1028, "Tathagata", "Acharyya", "A003")
              ]
```

Create Lists for “Address”.

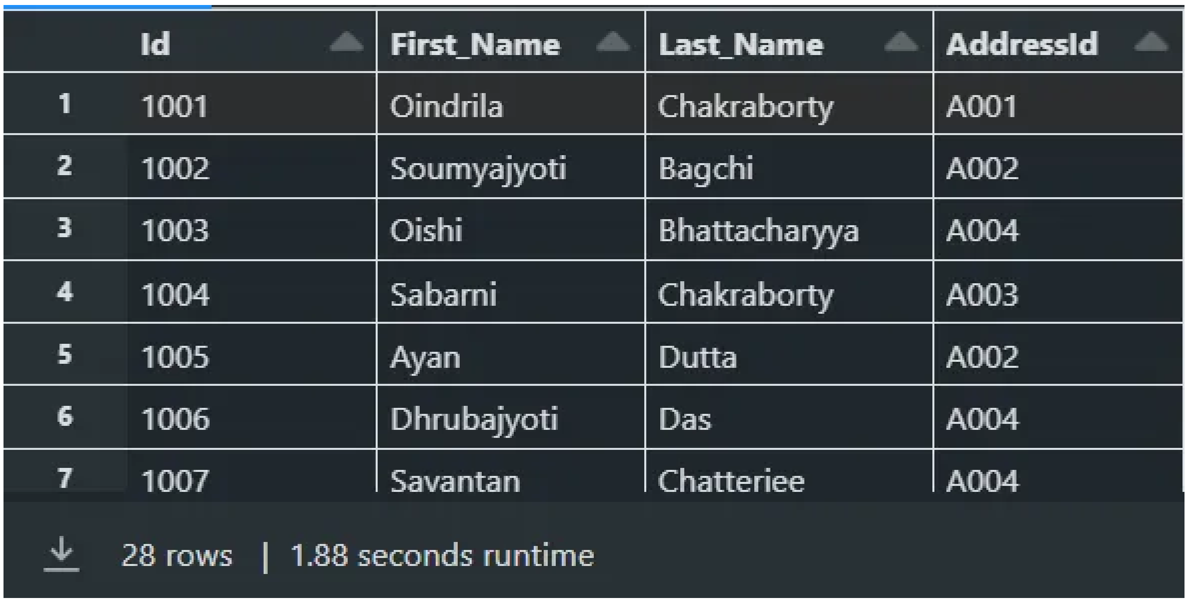
```
# Create a List Containing the "Column Names" for "Address"
addressColumns = ["AddressId", "Address"]

# Create a List Containing the "Data" for "Address"
addressList = [\
                  ("A001", "India"),
                  ("A002", "US"),
                  ("A003", "UK"),
                  ("A004", "UAE")
              ]
```

Create a DataFrame for “Person”.

```
dfPerson = spark.createDataFrame(personList, schema = personColumns)
dfPerson.printSchema()
display(dfPerson)
```

Output -



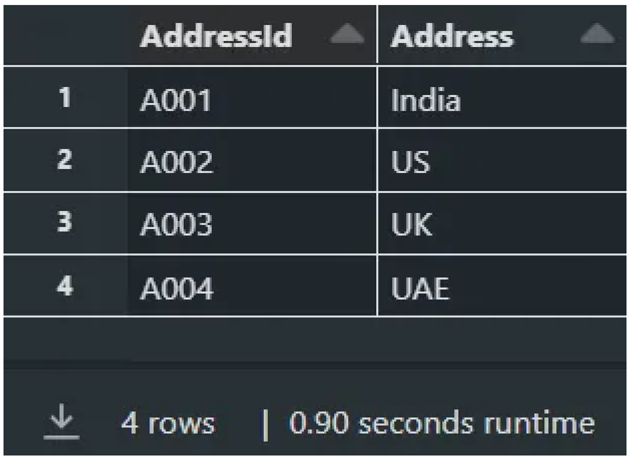
	Id	First_Name	Last_Name	AddressId
1	1001	Oindrila	Chakraborty	A001
2	1002	Soumyajyoti	Bagchi	A002
3	1003	Oishi	Bhattacharyya	A004
4	1004	Sabarni	Chakraborty	A003
5	1005	Ayan	Dutta	A002
6	1006	Dhrubajyoti	Das	A004
7	1007	Savantan	Chatterjee	A004

28 rows | 1.88 seconds runtime

Create Another DataFrame for “Address”.

```
dfAddress = spark.createDataFrame(addressList, schema = addressColumns)
dfAddress.printSchema()
display(dfAddress)
```

Output -



	AddressId	Address
1	A001	India
2	A002	US
3	A003	UK
4	A004	UAE

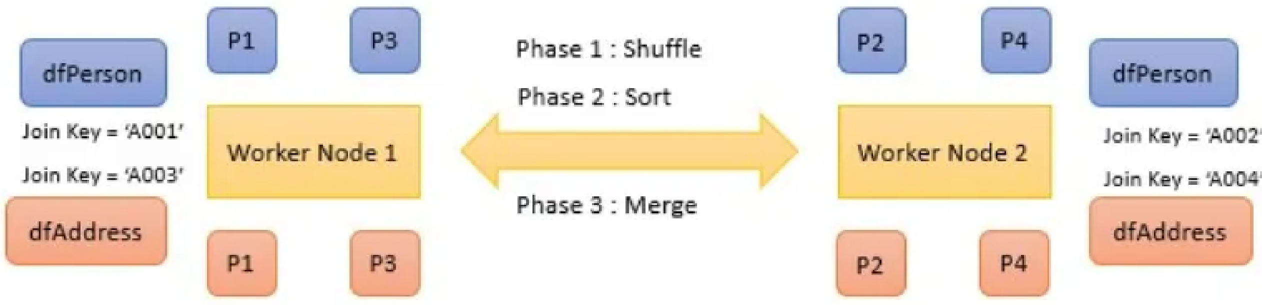
4 rows | 0.90 seconds runtime

1. Sort Merge Join

- The “**Sort Merge Join**” is the “**Default Join Selection Strategy**” when a “**Join**” **Operation** is “**Performed**” between “**Two DataFrames**”.

- The “**Sort Merge Join**” goes through the following “**Three Phases**” -
  1. **Shuffle**
  2. **Sort**
  3. **Merge**

- Consider, in the “**Cluster**”, there are “**Two Worker Nodes**”, and, “**Both**” of the “**DataFrames**” are “**Initially Distributed**” into “**4 Partitions**”. Then, the “**Sort Merge Join**” is “**Applied**” to “**Join**” the “**DataFrame**” of “**Person**” with the “**DataFrame**” of “**Address**”.



Phase 1 : “Shuffle”

A “**DataFrame**” is nothing but a “**Collection**” of “**Partitions**” that are “**Distributed**” across “**All**” the “**Cores**” of “**Every Worker Node**” in a “**Cluster**”.

When a “**DataFrame**” is “**Created**”, the “**Number of Partitions**” for that “**DataFrame**” is decided by various “**Parameters**”, like — the “**Default Block Size**”, i.e., “**128 MB**”.

When a “**Join**” **Operation** is “**Performed**” between “**Two DataFrames**”, the “**Driver Node**” sends some “**Piece of Code**”, involving the “**Join**” **Operation**, in the “**Form**” of “**Tasks**” to the “**Worker Nodes**”.

“**Each Task**” will “**Pick Up**” “**One Partition**” of “**Both**” of the “**DataFrames**” “**At a Time**”, and, that particular “**Task**” will “**Work On**” the “**Data**” that is “**Present**” in those particular “**Partitions**”.

The “**Join**” **Operation** is “**Performed**” based on a “**Key Column**”. It might happen that “**All**” the “**Data**”, pertaining to a particular “**Value**” of the “**Join Key Column**” is “**Not Present**” in a “**Single Partition**”, but is “**Scattered Around**” the “**Different Worker Nodes**” in the “**Cluster**”.
Example -

- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A001**” is “**Not Present**” in a “**Single Partition**” of the “**DataFrame**” of “**Person**”, but is “**Present**” in the “**Multiple Partitions**”.
  “**Each**” of these “**Partitions**” might be “**Present**” in “**Different Worker Nodes**”.

- The “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A001**” is “**Present Only**” in the “**Partition No: 2**” of the “**DataFrame**” of “**Address**”.
  So, “**Partition No: 2**” of the “**DataFrame**” of “**Address**” can “**Not**” be “**Present**” in “**Different Worker Nodes**”. Suppose, the “**Partition No: 2**” of the “**DataFrame**” of “**Address**” is “**Present**” in the “**Third Worker Node**”. Hence, the “**Join**” **Operation** can “**Not Occur**”.

The “**Data**” in “**Each Partition**”, for “**Both**” of the “**DataFrames**”, needs to be “**Shuffled**” so that “**All**” the “**Data**”, having a particular “**Value**” of the “**Join Key Column**” can be “**Stored**” in the “**Same Partition**” for the respective “**DataFrames**”, and, the “**Resultant Partitions**” of “**Both**” of the “**DataFrames**” can be “**Stored**” in the “**Same Worker Node**” depending on the “**Values**” of the “**Join Key Column**”, on which the “**Filtering**” happened.
Example -

- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A001**” should be “**Present**” in the “**Partition No: 1**” for “**Both**” of the “**DataFrames**” in the “**Worker Node : 1**”.

- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A002**” should be “**Present**” in the “**Partition No: 3**” for “**Both**” of the “**DataFrames**” in the “**Worker Node : 2**”.

- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A003**” should be “**Present**” in the “**Partition No: 1**” for “**Both**” of the “**DataFrames**” in the “**Worker Node : 1**”.

- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A004**” should be “**Present**” in the “**Partition No: 3**” for “**Both**” of the “**DataFrames**” in the “**Worker Node : 2**”.

Phase 2 : “Sort”

Once, the “**Shuffle**” **Phase**” is “**Over**”, in “**All**” of the “**Resultant Partitions**” of “**Both**” of the “**DataFrames**”, the “**Data**” is “**Not Sorted**” based on the “**Values**” of the “**Join Key Column**”.

In the **Second Phase**, i.e., “**Sort**” **Phase**”, “**All**” the “**Data**” in “**All**” of the “**Resultant Partitions**” of “**Both**” of the “**DataFrames**” are “**Sorted**” based on the “**Values**” of the “**Join Key Column**”.

The “**Resultant Sorted Partitions**”, after the **Second Phase** is “**Over**”, of “**Both**” of the “**DataFrames**” are still “**Present**” in the “**Same Working Nodes**” respectively.
Example -

“**Partition No: 1**” for the “**DataFrame**” of “**Person**” has “**Data**”, having “**A001**”, and, “**A003**” as the “**Value**” of the “**Join Key Column**”.
In the “**Sorting**” **Phase**, the “**Data**” of the “**Partition No: 1**” is “**Sorted**” -

- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A001**” is “**Kept Together**” in the “**Worker Node : 1**”.
- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A003**” is “**Kept Together**” in the “**Worker Node : 1**”.

“**Partition No: 3**” for the “**DataFrame**” of “**Person**” has “**Data**”, having “**A002**”, and, “**A004**” as the “**Value**” of the “**Join Key Column**”.
In the “**Sorting**” **Phase**, the “**Data**” of the “**Partition No: 3**” is “**Sorted**” -

- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A002**” is “**Kept Together**”.
- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A004**” is “**Kept Together**”.

Phase 3 : “Merge”

Once, the “**Sort**” **Phase**” is “**Over**”, “**All**” of the “**Resultant Partitions**” of “**Both**” of the “**DataFrames**” are “**Sorted**” based on the “**Values**” of the “**Join Key Column**”.

In the **Third Phase**, i.e., “**Merge**” **Phase**”, a “**Partition**” from the “**First DataFrame**” is “**Merged**” with a “**Partition**” from the “**Second DataFrame**” based on the “**Same Value**” of the “**Join Key Column**” in the “**Same Working Node**”.

Example -

- “**All**” the “**Data**” of the “**Partition No: 1**” for the “**DataFrame**” of “**Person**” having “**A001**” as the “**Value**” of the “**Join Key Column**” needs to be “**Merged**” with “**All**” the “**Data**” of the “**Partition No: 1**” for the “**DataFrame**” of “**Address**” having “**A001**” as the “**Value**” of the “**Join Key Column**” in the “**Worker Node : 1**”.

- “**All**” the “**Data**” of the “**Partition No: 3**” for the “**DataFrame**” of “**Person**” having “**A002**” as the “**Value**” of the “**Join Key Column**” needs to be “**Merged**” with “**All**” the “**Data**” of the “**Partition No: 3**” for the “**DataFrame**” of “**Address**” having “**A002**” as the “**Value**” of the “**Join Key Column**” in the “**Worker Node : 2**”.
- “**All**” the “**Data**” of the “**Partition No: 1**” for the “**DataFrame**” of “**Person**” having “**A003**” as the “**Value**” of the “**Join Key Column**” needs to be “**Merged**” with “**All**” the “**Data**” of the “**Partition No: 1**” for the “**DataFrame**” of “**Address**” having “**A003**” as the “**Value**” of the “**Join Key Column**” in the “**Worker Node : 1**”.
- “**All**” the “**Data**” of the “**Partition No: 3**” for the “**DataFrame**” of “**Person**” having “**A004**” as the “**Value**” of the “**Join Key Column**” needs to be “**Merged**” with “**All**” the “**Data**” of the “**Partition No: 3**” for the “**DataFrame**” of “**Address**” having “**A004**” as the “**Value**” of the “**Join Key Column**” in the “**Worker Node : 2**”.

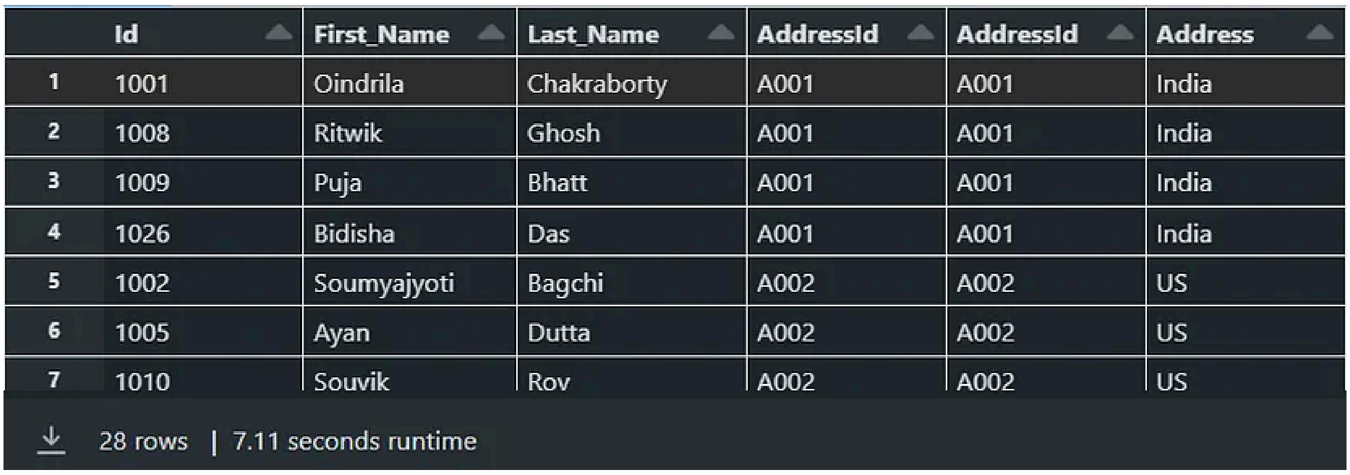
For What “Type” of “Join” the “Sort Merge Join” can be Used as the “Join Selection Strategy”?

- The “**Sort Merge Join**” can be used “**Only**” for the “**Equi Joins**”, i.e., those “**Join Operations**”, where “**Equals Operator**” (“==”) is used.
- The “**Sort Merge Join**” “**Works**” for “**All Types**” of “**Joins**”, including “**Full Outer Join**”.

“Join” the “Person DataDrame” With the “Address DataFrame” on “AddressId”.

```
dfPersonWithAddress = dfPerson.join(dfAddress, dfPerson.AddressId == dfAddress.
dfPersonWithAddress.printSchema()
display(dfPersonWithAddress)
```

Output -



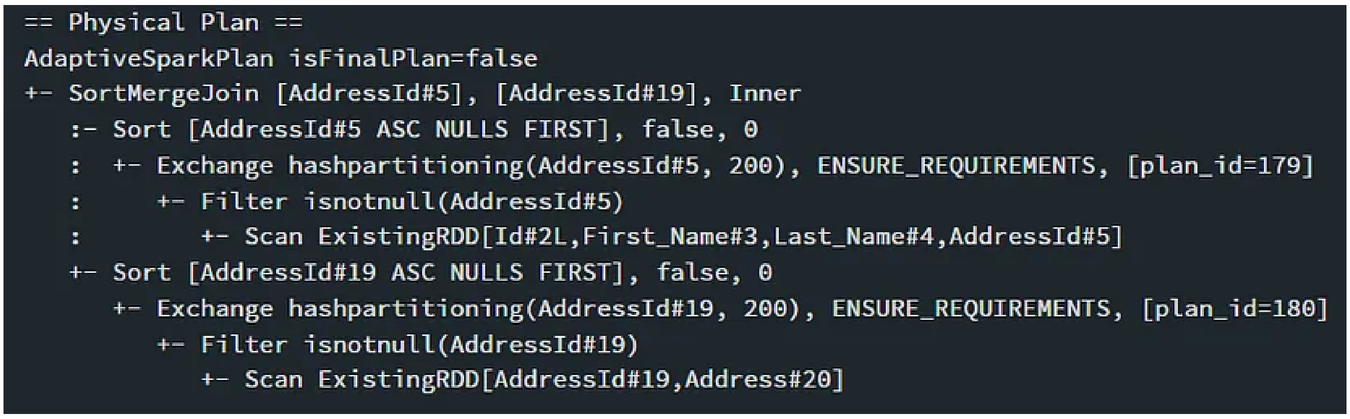
	Id	First_Name	Last_Name	AddressId	AddressId	Address
1	1001	Oindrila	Chakraborty	A001	A001	India
2	1008	Ritwik	Ghosh	A001	A001	India
3	1009	Puja	Bhatt	A001	A001	India
4	1026	Bidisha	Das	A001	A001	India
5	1002	Soumyajyoti	Bagchi	A002	A002	US
6	1005	Ayan	Dutta	A002	A002	US
7	1010	Souvik	Roy	A002	A002	US

28 rows | 7.11 seconds runtime

View the “Execution Plan” of the “DataFrame” Created by the “Join” Operation.

```
dfPersonWithAddress.explain()
```

Output -



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- SortMergeJoin [AddressId#5], [AddressId#19], Inner
   :- Sort [AddressId#5 ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(AddressId#5, 200), ENSURE_REQUIREMENTS, [plan_id=179]
   :     +- Filter isnotnull(AddressId#5)
   :        +- Scan ExistingRDD[Id#2L,First_Name#3,Last_Name#4,AddressId#5]
   +- Sort [AddressId#19 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(AddressId#19, 200), ENSURE_REQUIREMENTS, [plan_id=180]
         +- Filter isnotnull(AddressId#19)
            +- Scan ExistingRDD[AddressId#19,Address#20]
```

In the “Physical Plan”, it can be seen that the “Join” performed is “Sort Merge Join” indeed.

> *Even Though, “Both” of the “Person DataFrame”, and, the “Address DataFrame” are “Small”, still “Apache Spark” selected the “Sort Merge Join” as the “Join Selection Strategy” in the “Physical Plan” Phase during the “Join” Operation, because, the “Sort Merge Join” is the “Default Join Selection Strategy” when a “Join” Operation is “Performed” between “Two DataFrames”.*

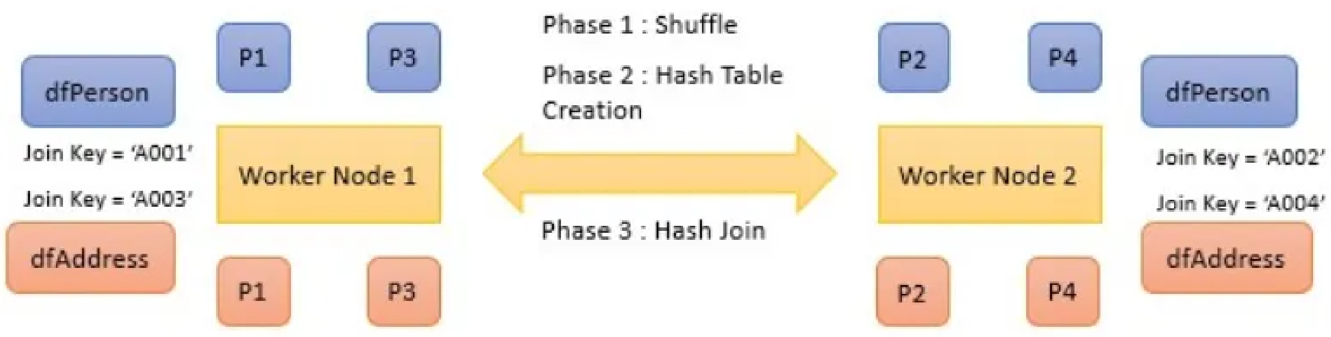
How to "Disable" the "Sort Merge Join" as the "Default Join Selection Strategy"?

- The "**Sort Merge Join**" is the "**Default Join Selection Strategy**" when a "**Join**" **Operation** is "**Performed**" between "**Two DataFrames**".
- To "**Disable**" the "**Sort Merge Join**" as the "**Default Join Selection Strategy**", the "**Spark Configuration Option**", i.e., "**spark.sql.join.preferSortMergeJoin**" is "**Set**" to "**false**".

```
spark.conf.set("spark.sql.join.preferSortMergeJoin", "false")
```

2. Shuffle Hash Join

- The "**Shuffle Hash Join**" goes through the following "**Three Phases**" -
  1. Shuffle
  2. Hash Table Creation
  3. Hash Join
- Consider, in the "**Cluster**", there are "**Two Worker Nodes**", and, "**Both**" of the "**DataFrames**" are "**Initially Spitted**" into "**4 Partitions**". Then, the "**Shuffle Hash Join**" is "**Applied**" to "**Join**" the "**DataFrame**" of "**Person**" with the "**DataFrame**" of "**Address**".



Phase 1: "Shuffle"

A "**DataFrame**" is nothing but a "**Collection**" of "**Partitions**" that are "**Distributed**" across "**All**" the "**Cores**" of "**Every Worker Node**" in a "**Cluster**".

When a "**DataFrame**" is "**Created**", the "**Number of Partitions**" for that "**DataFrame**" is decided by various "**Parameters**", like — the "**Default Block Size**", i.e., "**128 MB**".

When a "**Join**" **Operation** is "**Performed**" between "**Two DataFrames**", the "**Driver Node**" sends some "**Piece of Code**", involving the "**Join**" **Operation**, in the "**Form**" of "**Tasks**" to the "**Worker Nodes**".

"**Each Task**" will "**Pick Up**" "**One Partition**" of "**Both**" of the "**DataFrames**" "**At a Time**", and, that particular "**Task**" will "**Work On**" the "**Data**" that is "**Present**" in those particular "**Partitions**".

The "**Join**" **Operation** is "**Performed**" based on a "**Key Column**". It might happen that "**All**" the "**Data**", pertaining to a particular "**Value**" of the "**Join Key Column**" is "**Not Present**" in a "**Single Partition**", but is "**Scattered Around**" the "**Different Worker Nodes**" in the "**Cluster**".
Example -

- "**All**" the "**Data**", pertaining to the "**Join Key Column**" with the "**Value**" as "**A001**" is "**Not Present**" in a "**Single Partition**" of the "**DataFrame**" of "**Person**", but is "**Present**" in the "**Multiple Partitions**".
"**Each**" of these "**Partitions**" might be "**Present**" in "**Different Worker Nodes**".
- The "**Data**", pertaining to the "**Join Key Column**" with the "**Value**" as "**A001**" is "**Present Only**" in the "**Partition No: 2**" of the "**DataFrame**" of "**Address**".
So, "**Partition No: 2**" of the "**DataFrame**" of "**Address**" can "**Not**" be "**Present**" in "**Different Worker Nodes**". Suppose, the "**Partition No: 2**" of the "**DataFrame**" of "**Address**" is "**Present**" in the "**Third Worker Node**". Hence, the "**Join**" **Operation** can "**Not Occur**".

The "**Data**" in "**Each Partition**", for "**Both**" of the "**DataFrames**", needs to be "**Shuffled**" so that "**All**" the "**Data**", having a particular "**Value**" of the "**Join Key Column**" can be "**Stored**" in the "**Same Partition**" for the respective "**DataFrames**", and, the "**Resultant Partitions**" of "**Both**" of the "**DataFrames**" can be "**Stored**" in the "**Same Worker Node**" depending on the "**Values**" of the "**Join Key Column**", on which the "**Filtering**" happened.
Example -

- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A001**” should be “**Present**” in the “**Partition No: 1**” for “**Both**” of the “**DataFrames**” in the “**Worker Node : 1**”.
- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A002**” should be “**Present**” in the “**Partition No: 3**” for “**Both**” of the “**DataFrames**” in the “**Worker Node : 2**”.
- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A003**” should be “**Present**” in the “**Partition No: 1**” for “**Both**” of the “**DataFrames**” in the “**Worker Node : 1**”.
- “**All**” the “**Data**”, pertaining to the “**Join Key Column**” with the “**Value**” as “**A004**” should be “**Present**” in the “**Partition No: 3**” for “**Both**” of the “**DataFrames**” in the “**Worker Node : 2**”.

Phase 2 : “Hash Table Creation”

Once, the “**Shuffle**” **Phase**” is “**Over**”, a “**Hash Table**” will be “**Created**” based on the “**Smaller DataFrame**” from the “**Two DataFrames**” that are involved in the “**Join**” **Operation.**

In “**Each Working Node**”, a “**Hash Table**” will be “**Created**” based on “**Each**” of the “**Partitions**” of “**Smaller DataFrame**” that are “**Present**” in that particular “**Working Node**”.
Example -

In the “**Working Node : 1**”, for the “**Smaller DataFrame**”, i.e., the “**DataFrame**” of “**Address**”, the following “**Two Hash Tables**” will be “**Created**” -

- A “**Hash Tables**” will be “**Created**” for the “**Partition No : 1**” of the “**DataFrame**” of “**Address**”
- Another “**Hash Tables**” will be “**Created**” for the “**Partition No : 3**” of the “**DataFrame**” of “**Address**”

In the “**Working Node : 2**”, for the “**Smaller DataFrame**”, i.e., the “**DataFrame**” of “**Address**”, the following “**Two Hash Tables**” will be “**Created**” -

- A “**Hash Tables**” will be “**Created**” for the “**Partition No : 2**” of the “**DataFrame**” of “**Address**”

- Another “**Hash Tables**” will be “**Created**” for the “**Partition No : 4**” of the “**DataFrame**” of “**Address**”

Phase 3 : “Hash Join”

Once, the “**Hash Table**” is “**Created**” for “**Each**” of the “**Partitions**” of the “**Smaller DataFrame**” that are “**Present**” in a particular “**Working Node**”, “**Each**” of the “**Hash Table**” will be “**Joined**” with the “**Respective Partition**” of the “**Larger DataFrame**” in the “**Same Working Node**”.
Example -

In the “**Working Node : 1**” -

- The “**Hash Tables**” that is “**Created**” for the “**Partition No : 1**” of the “**DataFrame**” of “**Address**” will be “**Joined**” with the “**Partition No : 1**” of the “**DataFrame**” of “**Person**”
- The “**Hash Tables**” that is “**Created**” for the “**Partition No : 3**” of the “**DataFrame**” of “**Address**” will be “**Joined**” with the “**Partition No : 3**” of the “**DataFrame**” of “**Person**”

In the “**Working Node : 2**” -

- The “**Hash Tables**” that is “**Created**” for the “**Partition No : 2**” of the “**DataFrame**” of “**Address**” will be “**Joined**” with the “**Partition No : 2**” of the “**DataFrame**” of “**Person**”
- The “**Hash Tables**” that is “**Created**” for the “**Partition No : 4**” of the “**DataFrame**” of “**Address**” will be “**Joined**” with the “**Partition No : 4**” of the “**DataFrame**” of “**Person**”

Why the “Shuffle Hash Join” is “Termed” as “Expensive Operation”?

- The “**Shuffle Hash Join**” is an “**Expensive Operation**”, because, it uses both the “**Shuffling**”, and, the “**Hashing**”, which are “**Individually**” “**Expensive Operations**”.
  Also, “**Maintaining**” a “**Hash Table**” requires “**Memory**”, and, “**Computation**”.

When the “Shuffle Hash Join” Can “Work” as the “Join Selection Strategy”?

“**Apache Spark**” will “**Not Choose**” the “**Shuffle Hash Join**” as the “**Join Selection Strategy**” by default, because, the “**Shuffle Hash Join**” is an “**Expensive Operation**”.

The “**Shuffle Hash Join**” can be used as the “**Join Selection Strategy**” if the following criteria are met -

- When the “**Sort Merge Join**” **Operation** is “**Disabled**” as the “**Default Join Selection Strategy**”.
  To “**Disable**” the “**Sort Merge Join**” as the “**Default Join Selection Strategy**”, the “**Spark Configuration Option**”, i.e., “**spark.sql.join.preferSortMergeJoin**” is “**Set**” to “**false**”.
- When the “**Broadcast Hash Join**” **Operation** can “**Not**” be used as the “**Join Selection Strategy**”, because, “**Both**” of the “**DataFrames**” involved in a “**Join**” **Operation** are “**Above**” the “**Range**” of the “**Default Broadcast Threshold Limit**”, i.e., “**10 MB**”.
  It is also possible to “**Forcibly Disable**” the “**Broadcast Hash Join**” by “**Setting**” the “**Value**” of the “**Broadcast Threshold Limit**” to “**-1**”.
- If the “**Sizes**” of “**Both**” of the “**DataFrames**” are “**Already Known**”, then “**Explicitly Pass**” the “**Join Hint**” as “**SHUFFLE_HASH**” for the “**Smaller DataFrame**” during the “**Join**” **Operation.**

“Forcibly Disable” the “Broadcast Hash Join” by Setting “Broadcast Threshold Limit” to “-1”.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

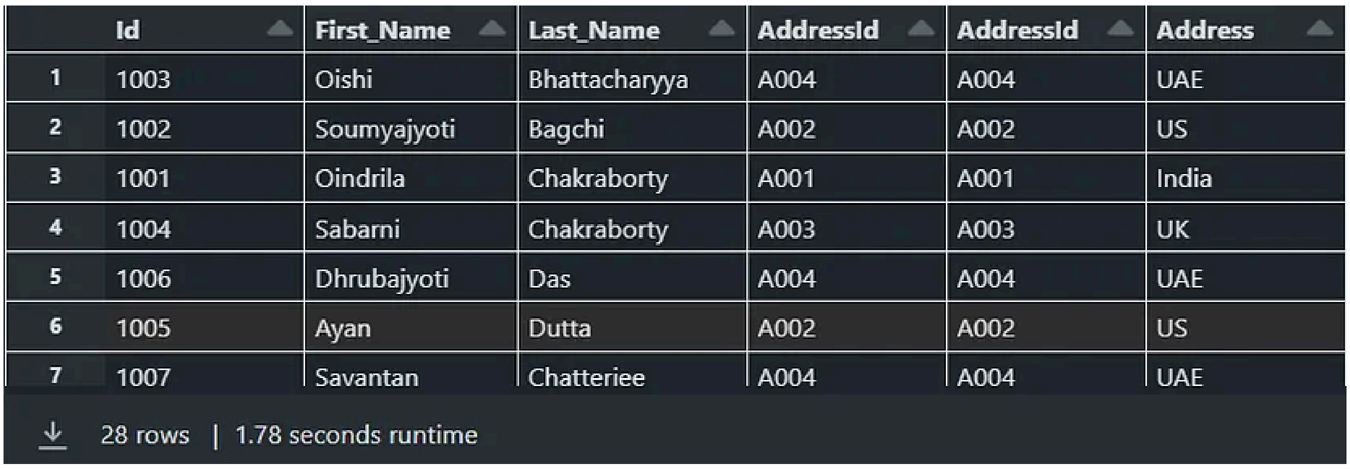
“Disable” the “Sort Merge Join” as the “Default Join Selection Strategy”.

```
spark.conf.set("spark.sql.join.preferSortMergeJoin", "false")
```

“Explicitly Pass” the “Join Hint” as “SHUFFLE_HASH” for the “Smaller DataFrame” While “Joining” the “Person DataDrame” With the “Address DataFrame” on “AddressId”.

```
dfPersonWithAddress = dfPerson.join(dfAddress.hint("SHUFFLE_HASH"), dfPerson.Ac
dfPersonWithAddress.printSchema()
display(dfPersonWithAddress)
```

Output -



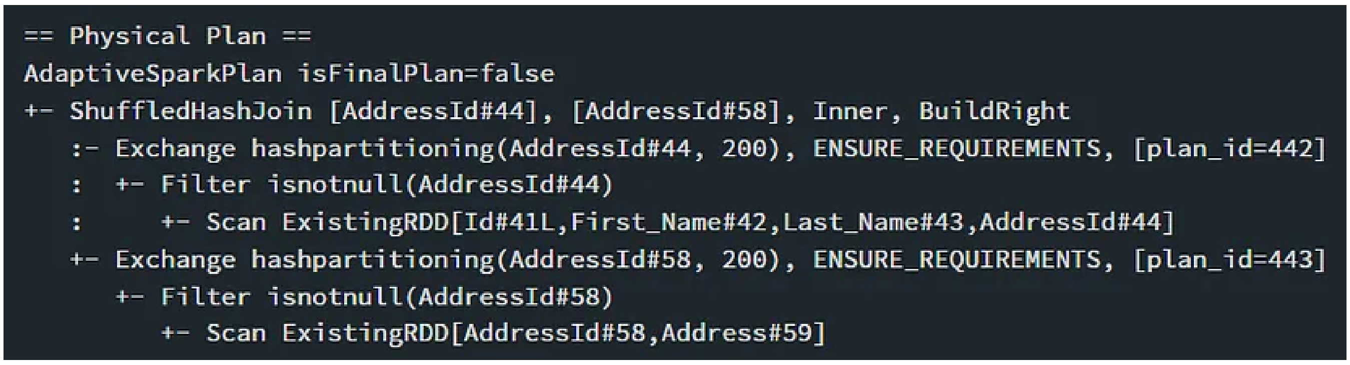
	Id	First_Name	Last_Name	AddressId	AddressId	Address
1	1003	Oishi	Bhattacharyya	A004	A004	UAE
2	1002	Soumyajyoti	Bagchi	A002	A002	US
3	1001	Oindrila	Chakraborty	A001	A001	India
4	1004	Sabarni	Chakraborty	A003	A003	UK
5	1006	Dhrubajyoti	Das	A004	A004	UAE
6	1005	Ayan	Dutta	A002	A002	US
7	1007	Savantan	Chatterjee	A004	A004	UAE

28 rows | 1.78 seconds runtime

View the “Execution Plan” of the “DataFrame” Created by the “Join” Operation.

```
dfPersonWithAddress.explain()
```

Output -



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- ShuffledHashJoin [AddressId#44], [AddressId#58], Inner, BuildRight
   :- Exchange hashpartitioning(AddressId#44, 200), ENSURE_REQUIREMENTS, [plan_id=442]
   :  +- Filter isnotnull(AddressId#44)
   :     +- Scan ExistingRDD[Id#41L,First_Name#42,Last_Name#43,AddressId#44]
   +- Exchange hashpartitioning(AddressId#58, 200), ENSURE_REQUIREMENTS, [plan_id=443]
      +- Filter isnotnull(AddressId#58)
         +- Scan ExistingRDD[AddressId#58,Address#59]
```

In the “Physical Plan”, it can be seen that the “Join” performed is “Shuffle Hash Join” indeed.

> *Even Though, “Both” of the “Person DataFrame”, and, the “Address DataFrame” are “Small”, “Apache Spark” did not select the “Broadcast Hash Join” as the “Join Selection Strategy” in the “Physical Plan” Phase during the “Join” Operation, because, the “Broadcast Hash Join” is “Forcibly Disabld” by Setting the “Value” of the “Broadcast Threshold Limit” to “-1”.*
>
> *“Apache Spark” also did not select the “Sort Merge Join” as the “Join Selection Strategy” in the “Physical Plan” Phase during the “Join” Operation, because, the “Sort Merge Join” is is “Forcibly Disabld” as the “Default Join Selection Strategy”.*
>
> *“Apache Spark” selected the “Shuffle Hash Join” as the “Join Selection Strategy” in the “Physical Plan” Phase during the “Join” Operation, because, during the “Join” Operation, it was mentioned in the “Hint” of the “Smaller DataFrame”, i.e., “DataFrame” of “Address”.*

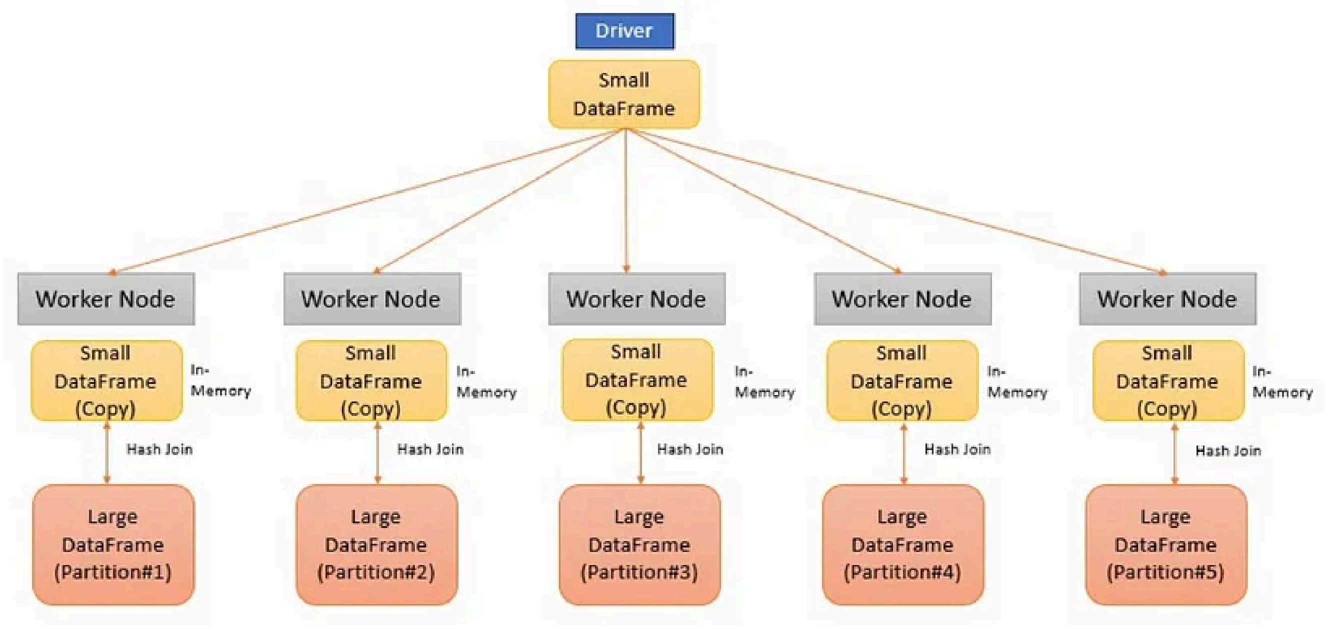
For What “Type” of “Join” the “Shuffle Hash Join” can be Used as the “Join Selection Strategy”?

- The “**Shuffle Hash Join**” can be used “**Only**” for the “**Equi Joins**”, i.e., those “**Join Operations**”, where “**Equals Operator**” (“==”) is used.
- The “**Shuffle Hash Join**” can be used for “**All Types**” of “**Join**”, only “**Except**” for the “**Full Outer Join**”.

3. Broadcast Hash Join

- When a “**Join**” **Operation** is “**Performed**” between “**Two DataFrames**”, if the “**Size**” of “**Any One**” or “**Both**” the “**DataFrames**” lie “**Within**” the “**Range**” of the “**Broadcast Threshold Limit**”, then a “**Read-Only Copy**” of the “**DataFrame**”, which is having a “**Size**” that is “**Within**” the “**Range**” of the “**Broadcast Threshold Limit**”, is “**Made Available**” in the “**Memory**” of “**All**” the “**Worker Nodes**” in the “**Cluster**”.

- Then, a “**Hash Join**” will be “**Performed**” between the “**Read-Only Copy**” of the “**DataFrame**” that is “**Made Available**” in “**Each**” of the “**Worker Nodes**”, and, the “**Partition**” of the “**Large DataFrame**” that is “**Present**” in the “**Respective Worker Nodes**”.


Driver
Small DataFrame
Worker Node
Small DataFrame (Copy)
In-Memory
Hash Join
Large DataFrame (Partition#1)
Large DataFrame (Partition#2)
Large DataFrame (Partition#3)
Large DataFrame (Partition#4)
Large DataFrame (Partition#5)

“Default Size” of the “Broadcast Threshold Limit”

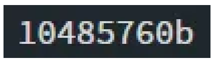
- Using the “**Spark Configuration Option**”, i.e., “**spark.sql.autoBroadcastJoinThreshold**”, it is possible to “**Find Out**” the “**Maximum Size**” of a “**DataFrame**” that is “**Allowed**” to be “**Broadcasted**”.

- The “**Default Value**” of the “**Broadcast Threshold Limit**” is “**10 MB**”, which means that “**Any DataFrame**”, which is “**Below 10 MB**” in “**Size**” is “**Allowed**” to be “**Broadcasted**”.

Display the “Default Size” of the “Broadcast Threshold Limit”.

```
print(spark.conf.get("spark.sql.autoBroadcastJoinThreshold"))
```

Output -



```
10485760b
```

Can the “Default Size” of the “Broadcast Threshold Limit” be “Changed” to “Any Other Value”?

- Using the “**Spark Configuration Option**”, i.e., “**spark.sql.autoBroadcastJoinThreshold**”, it is possible to “**Change**” the “**Size**” of the “**Broadcast Threshold Limit**” from its “**Default Size**” to “**Any Desired Size**” as per the “**Requirement**” as well.

Change “Size” of the “Broadcast Threshold Limit” from its “Default Value” to “Any Desired Value”.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 99999999)
```

How to “Disable” the “Broadcast Threshold Limit”?

- By default, the “**Broadcast Threshold Limit**” is “**Enabled**”.

- To “**Disable**” the “**Broadcast Threshold Limit**”, the “**Spark Configuration Option**”, i.e., “**spark.sql.autoBroadcastJoinThreshold**” is “**Set**” to “**-1**”.

“Disable” the “Broadcast Threshold Limit”.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

Why “Lesser Shuffling” Occurs in a “Broadcast Hash Join”?

- When a “**Broadcast Hash Join**” **Operation** is “**Performed**” between “**Two DataFrames**”, “**Lesser Shuffling**” **Occurs**, because, the “**Two DataFrames**” that are “**Involved**” in the “**Join**” **Operation** are “**Present**” in “**Each**” of the “**Worker Nodes**”, as the “**Read-Only Copy**” of the “**Small DataFrame**” is “**Made Available**” in “**Each**” of the “**Worker Nodes**”, where the “**Respective Partition**” of the “**Large DataFrame**” is also “**Present**”.

Than the “Broadcast Threshold Limit” and “Broadcast Hash Join” is Used?

- When a “**Broadcast Hash Join**” **Operation** is “**Performed**” between “**Two DataFrames**”, where the “**Size**” of “**Both**” the “**DataFrames**” are “**Larger**” than the “**Broadcast Theshold Limit**”, and, a “**Read-Only Copy**” of “**Any One**” of the “**DataFrames**” is “**Made Available**” in the “**Memory**” of “**All**” the “**Worker Nodes**” in the “**Cluster**”, then “**Out of Memory**” **Error** will be “**Thrown**” by the “**Apache Spark**”.

How to “Explicitly Define” the “Broadcast Hash Join” as the “Join Selection Stratey” During the “Join” Operation?

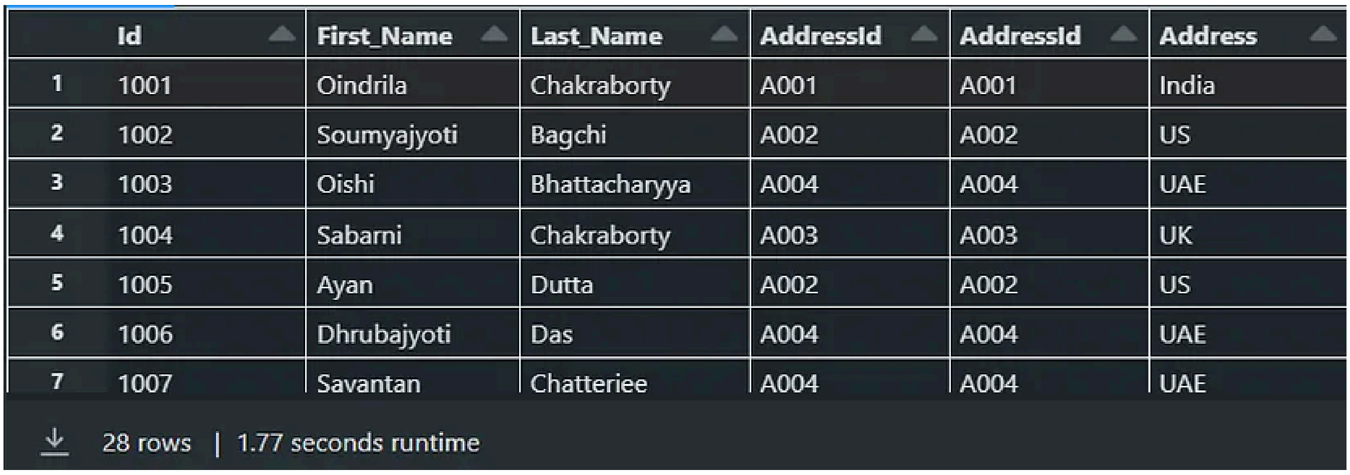
- It is also possible to “**Explicitly Define**” the “**Broadcast Hash Join**” as the “**Join Selection Strategy**” During the “**Join**” **Operation** by using the “**Broadcast Variable**” to “**Send**” the “**Smaller DataFrame**” to “**Each Worker Node**” of the “**Cluster**”.
- To do so, the “**Smaller DataFrame**” is “**Passed**” to the “**broadcast**” **Function** during the “**Join**” **Operation.**

“Join” the “Person DataDrame” With the “Address DataFrame” on “AddressId” Using “Broadcast Variable”.

```
from pyspark.sql.functions import broadcast

dfPersonWithAddress = dfPerson.join(broadcast(dfAddress), dfPerson.AddressId ==
dfPersonWithAddress.printSchema()
display(dfPersonWithAddress)
```

Output -



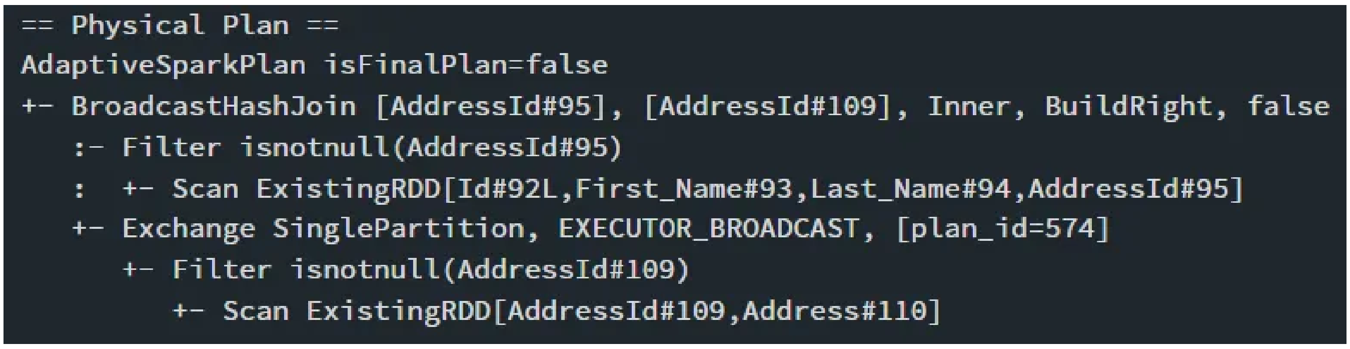
	Id	First_Name	Last_Name	AddressId	AddressId	Address
1	1001	Oindrila	Chakraborty	A001	A001	India
2	1002	Soumyajyoti	Bagchi	A002	A002	US
3	1003	Oishi	Bhattacharyya	A004	A004	UAE
4	1004	Sabarni	Chakraborty	A003	A003	UK
5	1005	Ayan	Dutta	A002	A002	US
6	1006	Dhrubajyoti	Das	A004	A004	UAE
7	1007	Savantan	Chatteriee	A004	A004	UAE

28 rows | 1.77 seconds runtime

View the “Execution Plan” of the “DataFrame” Created by the “Join” Operation.

```
dfPersonWithAddress.explain()
```

Output -



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- BroadcastHashJoin [AddressId#95], [AddressId#109], Inner, BuildRight, false
   :- Filter isnotnull(AddressId#95)
   :  +- Scan ExistingRDD[Id#92L,First_Name#93,Last_Name#94,AddressId#95]
   +- Exchange SinglePartition, EXECUTOR_BROADCAST, [plan_id=574]
      +- Filter isnotnull(AddressId#109)
         +- Scan ExistingRDD[AddressId#109,Address#110]
```

In the “Physical Plan”, it can be seen that the “Join” performed is “Broadcast Hash Join” indeed.

> *Even Though, “Both” of the “Person DataFrame”, and, the “Address DataFrame” are “Small”, “Apache Spark” would not have selected the “Broadcast Hash Join” as the “Join Selection Strategy” in the “Physical Plan” Phase during the “Join” Operation, because, the “Shuffle Sort Join” is the “Default Join Selection Strategy”.*
>
> *To make “Apache Spark” select the “Broadcast Hash Join” as the “Join Selection Strategy” in the “Physical Plan” Phase during the “Join” Operation, the “Smaller DataFrame”, i.e., the “DataFrame” of “Address”, is “Sent” using the “Broadcast Variable” “Explicitly”.*

Data · Data Engineering · Data Science · Azure · Databricks

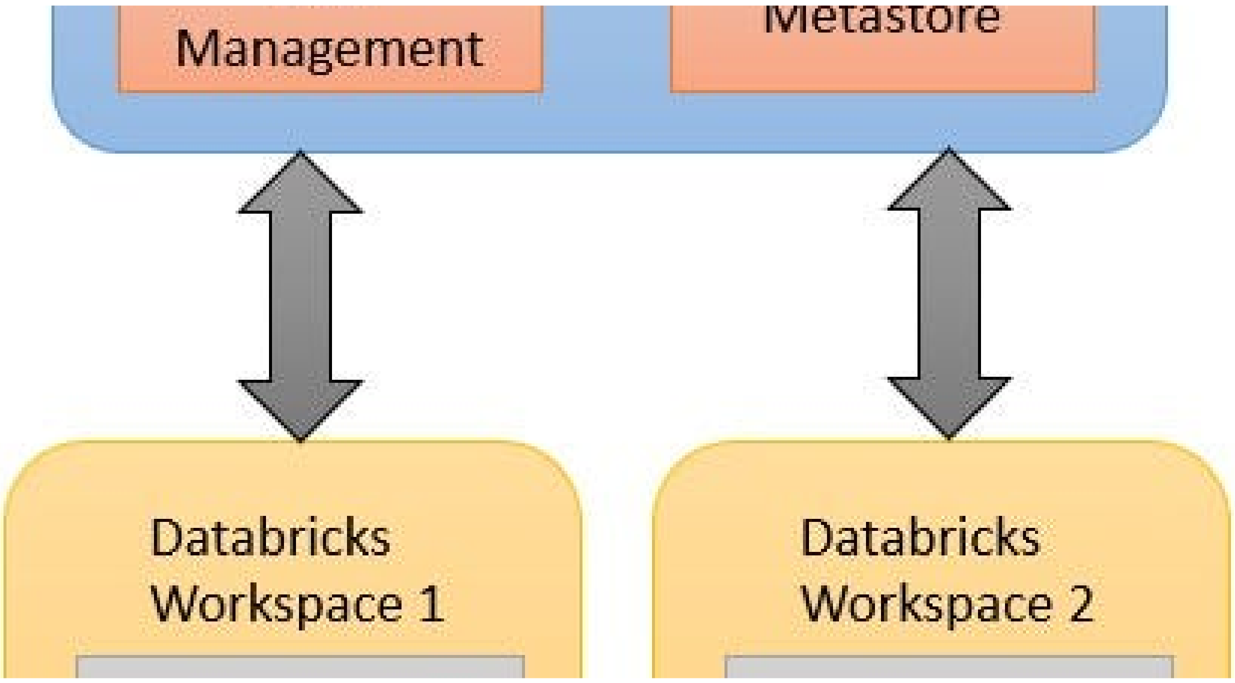
Follow

Written by Oindrila Chakraborty

625 Followers

I have 11+ experience in IT industry. I love to learn about the data and work with data. I am happy to share my knowledge with all. Hope this will be of help.

More from Oindrila Chakraborty



Oindrila Chakraborty

Introduction to “Unity Catalog” in Databricks

What is “Unity Catalog”?

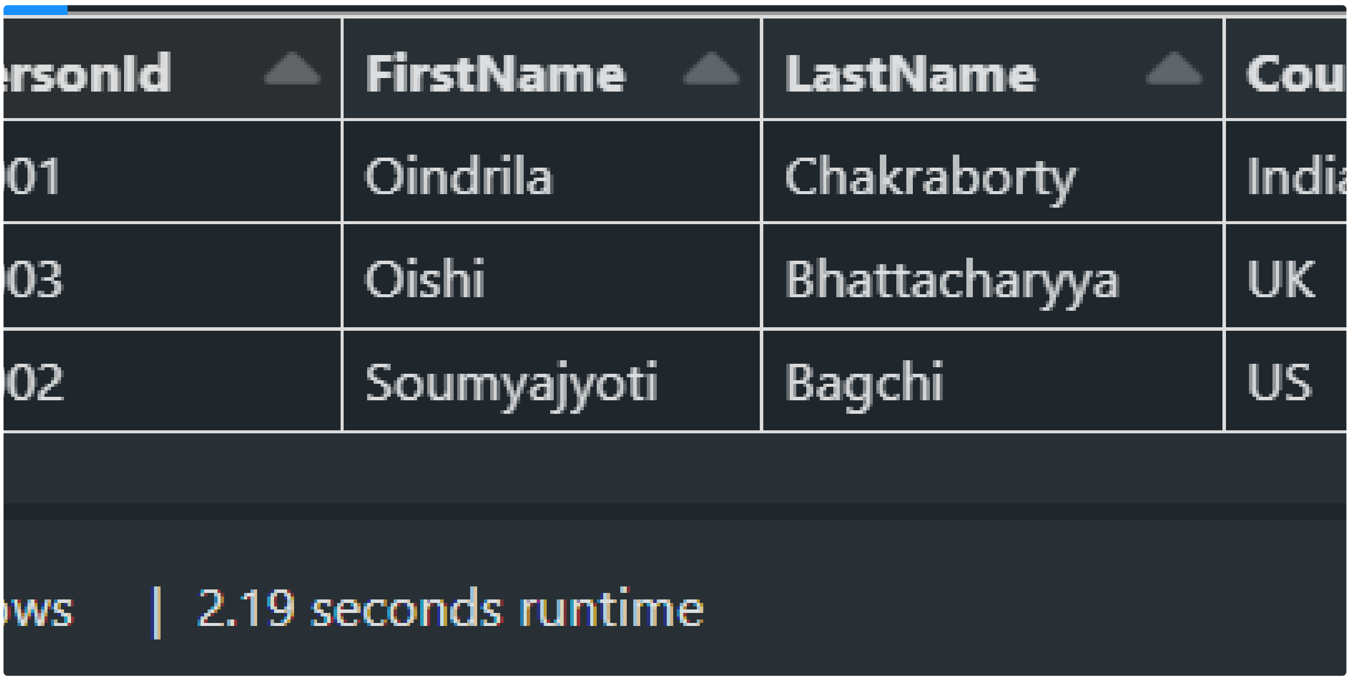
Aug 15, 2023 93 3

Oindrila Chakraborty

How to “Optimize” the “Delta Tables” in Databricks

How “Delta Tables” Become “Slow” to “Query”?

Sep 15, 2023 60 1



rsonId	FirstName	LastName	Cou
01	Oindrila	Chakraborty	Indi
03	Oishi	Bhattacharyya	UK
02	Soumyajyoti	Bagchi	US

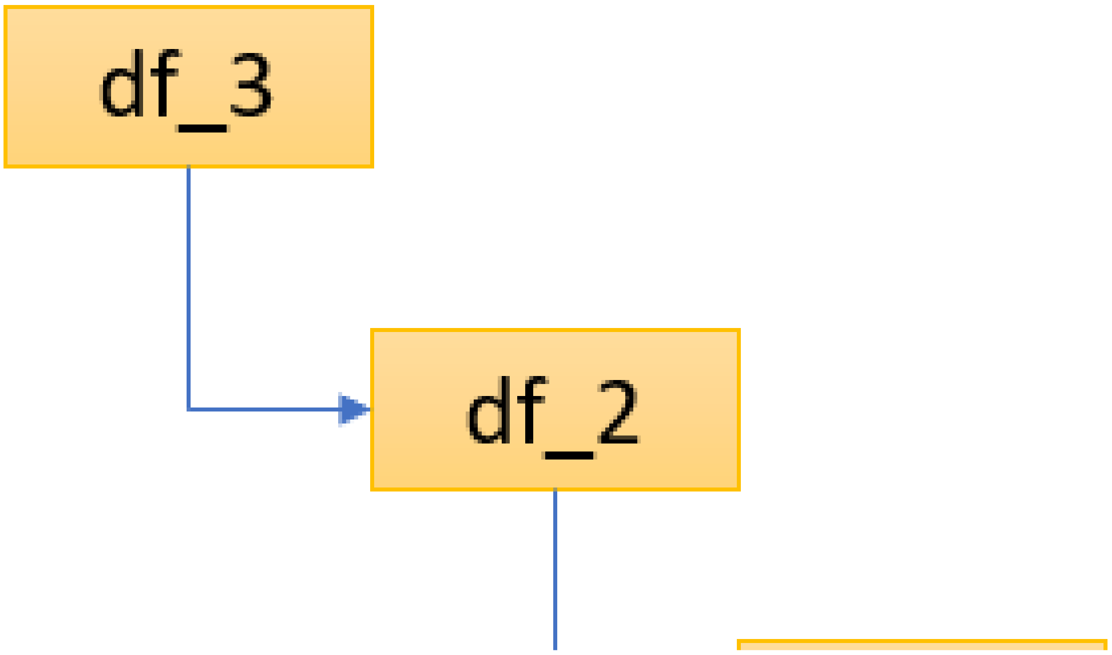
ws | 2.19 seconds runtime

Oindrila Chakraborty

Perform “SCD Type 1” Using “MERGE” Operation on Delta Table Using “SPARK SQL” and “PySpark” in...

What is “Slowly Changing Dimension”?

Oct 7, 2023 31 5



Oindrila Chakraborty

Cache vs. Persist in Databricks

What are “Cache” and “Persist”?

May 7 50 4

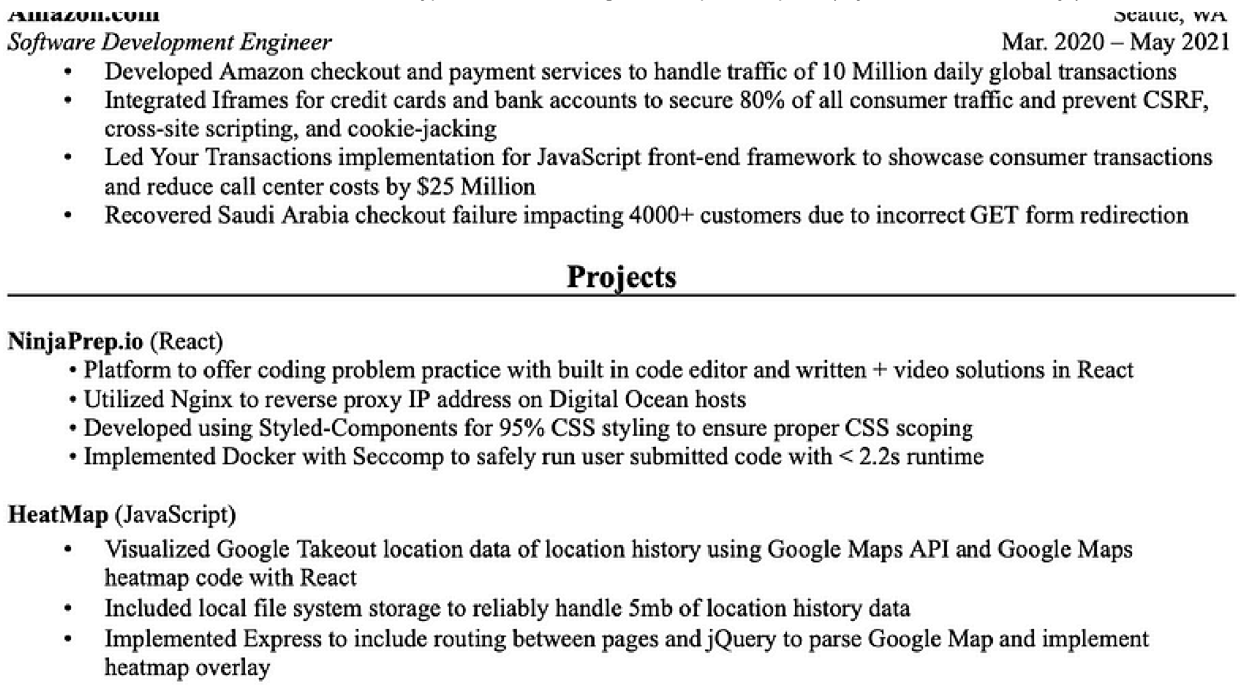
See all from Oindrila Chakraborty

Recommended from Medium

Sai Prabhanj Turaga

Difference between Datamart, Datawarehouse and Deltalake

May 1 30



Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)

- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)

- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

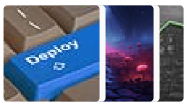
Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

Jun 1 23K 459

Lists

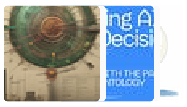


Predictive Modeling w/ Python

20 stories · 1580 saves

Practical Guides to Machine Learning

10 stories · 1917 saves



data science and AI

40 stories · 259 saves

Coding & Development

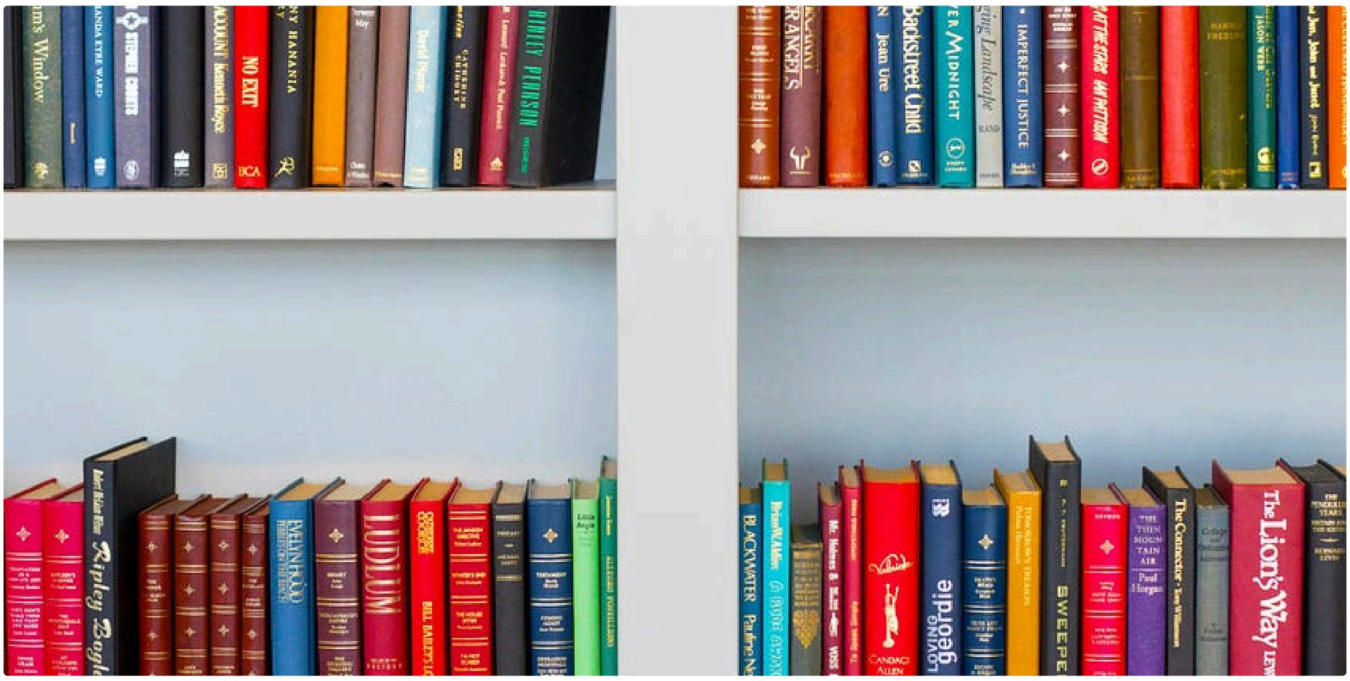
11 stories · 836 saves

Deepa Vasanthkumar

Spark Logical and Physical Plan Generation

In Spark, when you submit a SQL query or DataFrame transformation, it goes through several stages of processing before execution. Let us...

Apr 9 124

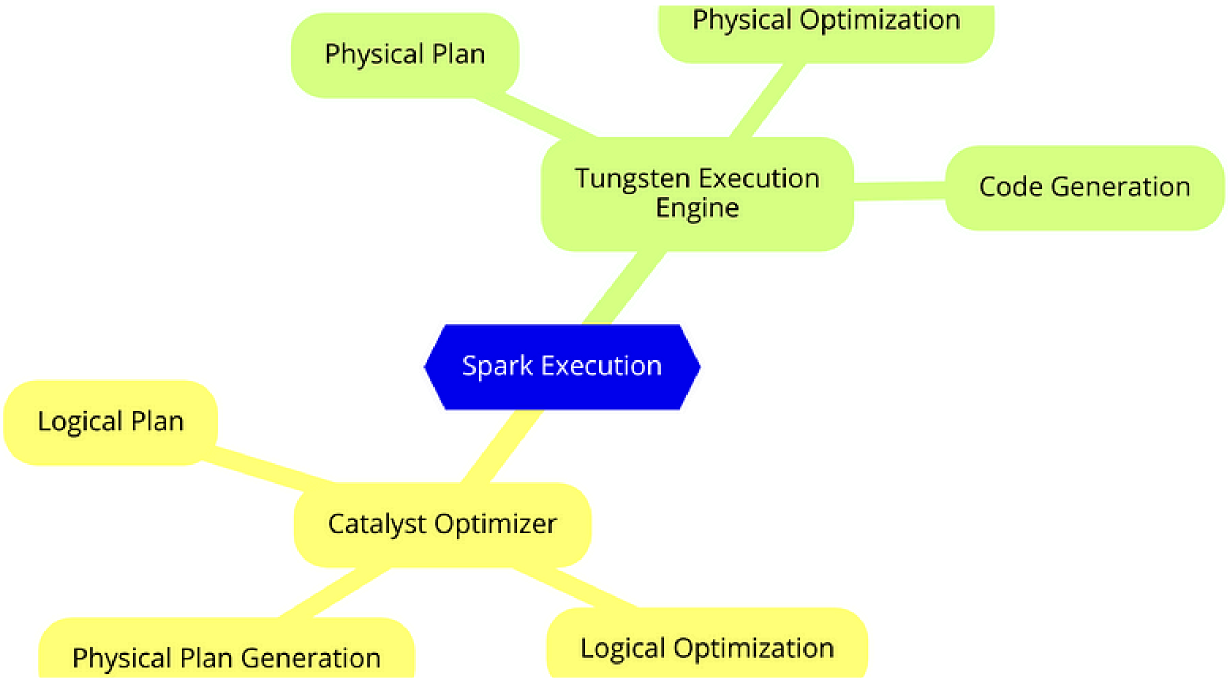


Subham Khandelwal in Dev Genius

PySpark—Run Multiple Jobs in Parallel

Understand How to Execute multiple Jobs in Parallel or Concurrently in PySpark

Sep 2 54 2

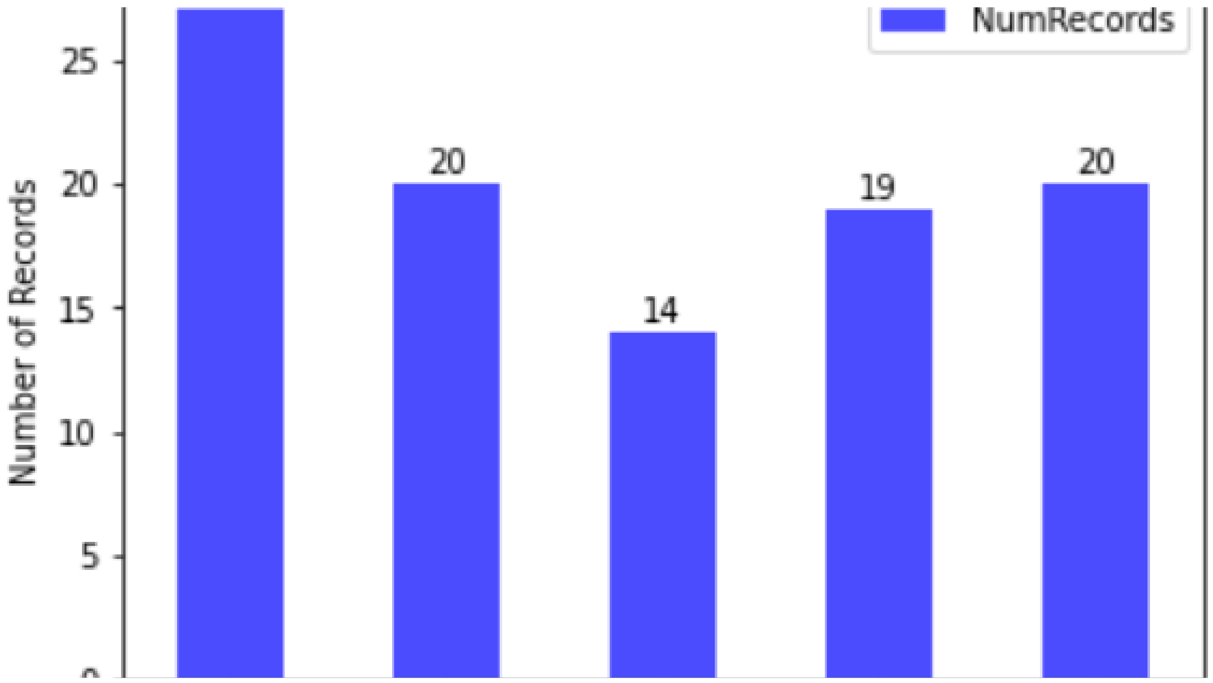


Prem Vishnoi(cloudvala)

Exploring Apache Spark’s Catalyst Optimizer and Tungsten Execution Engine

Apache Spark includes several sophisticated components to optimize and execute queries efficiently.

Jun 3 4



Yash Kothari

Custom Partitioning in Pyspark

In Apache Spark, the partitioner plays a crucial role in determining how data is distributed across the nodes in a cluster during…

May 6 3

See more recommendations